The Loyal Opposition Comments on Plan Domain Description Languages

Jeremy Frank and Keith Golden and Ari Jonsson
Computational Sciences Division
NASA Ames Research Center, MS 269-2
frank@email.arc.nasa.gov
Moffett Field, CA 94035

Abstract

In this paper we take a critical look at PDDL 2.1 as designers and users of plan domain description languages. We describe planning domains that have features which are hard to model using PDDL 2.1. We then offer some suggestions on domain description language design, and describe how these suggestions make modeling our chosen domains easier.

The Loyal Opposition

PDDL has served as the underpinnings of the Planning Competition, and has had an enormous impact on the planning community as a whole. PDDL and STRIPS have served as a *lingua franca* for deterministic planning domains, making it possible for researchers to compare techniques on the same problems and enabling meaningful comparisons of those techniques. The result has been considerable progress in planning algorithms.

Recently, however, the problems of interest to the planning community have changed. Goal achievement by itself is no longer enough, and the form of the goals has changed as well. Temporal planning requires meeting goals that include temporal constraints and resource constraints. While it is possible to model planning domains with temporally extended state and resources using purely propositional modeling languages, it is expensive in modeler effort and results in huge domain descriptions. By contrast, software domains such as data processing, web services integration and and information integration generally require incomplete information, incompletely known dynamic universes and sensing actions, but do not, generally, require rich models of time.

In this paper, we take a critical look at the design of PDDL 2.1. Our perspective is that of researchers in planning and scheduling who have considerable experience in designing and using planning domain description languages. We will describe planning domains that pose problems for PDDL 2.1. We will then describe what we believe are the core set of features for modeling planning domains. We will show how these core features simplify the modeling of the domains of interest.

PDDL and its Discontents

In this section, we describe some domain models and their PDDL 2.1 representations, and then discuss reasons why the PDDL 2.1 model is problematic.

Temporal Constraints and Instantaneous Events

PDDL 2.1 requires that propositions hold for a non-zero amount of time before they are used as preconditions for actions. However, PDDL's underlying representation of states is not based on intervals, and so this makes it dif£cult to specify the interaction between multiple events that modify the same proposition. It will also be dif£cult to see what domain axioms resulted in the separation of two events that modify the same proposition in the £nal plan.

Part of the reason for this is that sometimes actions require additional constraints to decide whether the action sequence is legal, and PDDL 2.1's semantics forbid such plans when there is a possibility of plan execution failure. The following example appears in (Fox & Long 2003) . The model includes action A with preconditions $P \vee Q$ and effect R, and action B with preconditions P and effect $\neg P$ and S. These two actions are considered mutually exclusive because a plan in which A and B execute concurrently may fail. We might have A and B execute at the same time in a state where $\neg Q$ holds, for example. PDDL assumes that the executions of A and B are actually ordered, and that the order is arbitrary. Thus, if the state happens to be one where $\neg Q$ holds, it may be that B happens £rst, making A fail both its preconditions.

Modeling this scenario by asserting the intervals of time over which P and Q hold, and forcing actions to declare how long the precondition must hold, clari£es the situation. For example, if A requires P to hold for 1 unit of time prior to A's execution, then in order for the plan to be legal, B can't happen at a time that makes P true for less than a unit of time. This makes it possible to both post and check the constraints that must hold for both A and B to happen concurrently. If P has held for long enough, then A and B can happen at the same instant, even in a state where $\neg Q$ holds.

A related problem is that it is hard to express certain temporal constraints in PDDL 2.1. For example, suppose that an action Take-Picture requires a proposition stable to hold for a period lasting from at least 5 seconds before to at least 5 seconds after the Take-Picture ends. Furthermore, the camera expends energy only during the time of the exposure, which lasts 24 seconds. During the intervening 5 seconds before and after the exposure, the camera could be involved in other activities such as changing the £lters, but can't slew (as this results in vibration).

PDDL 2.1 does not allow direct expression of the constraints on stability. As we said previously, there is no way

```
(:durative-action take-picture
:parameters (?s - satellite ?c - camera ?o - observation)
:duration (= ?duration 24)
:condition (and (at start (pointing-at (?s ?o)))
   (at start (on ?c))
   (at start (stable ?s))
   (over all (on ?c))
   (over all (stable ?s))
:effect (and (captured-but-not-stabilized ?o)
   (at start (decrease (energy ?s 5)))
   (at end (increase (energy ?s 5)))))

(:durative-action stabilize-before
:parameters (?s - satellite)
:duration (= ?duration 5)
:condition (over all (stable ?s))
:effect (and(at end (stable ?s))))

(:durative-action stabilize-after
:parameters (?s - satellite ?o - object)
:duration (= ?duration 5)
:condition (and(at-start (captured-but-not-stabilized ?o))
   (at start (stable ?s))
   (over all (stable ?s)))
:effect (and (at end (not(captured-but-not-stabilized ?o)))
   (at end (captured ?o))))
```

Figure 1: Temporal Constraints in PDDL 2.1

in PDDL 2.1 to write a condition that requires a proposition to hold for an interval of time. Thus, we can't even require that Stable hold for 5 seconds before Take-Picture can start. We can write an action Stabilize-Before that lasts for 5 seconds, with an effect Stable, after which Take-Picture can be performed. Take-Picture also asserts Stable throughout the 24 second interval, and properly asserts that the instrument must be on, and affects the power properly. Now, however, we must ensure that stability is held for another 5 seconds. We can do so by asserting a condition captured-but-not-stabilized for the observation. The only way to assert captured, then, is to execute the stabilize-after action. The only remaining problem is that these actions should be executed back-to-back to enforce our original constraint, and PDDL 2.1 has no way of ensuring that this happens.

Notice that PDDL 2.1 does not allow direct assertions that states hold for a period of time, but rather uses action duration to indirectly affect the amount of time that propositions hold during a plan. More generally, PDDL preserves the notion of preconditions and effects from STRIPS, which simply doesn't make sense when considering plans with overlapping concurrent activities. The stability condition described above is more than a persistent precondition, in that it outlasts the end of the action. PDDL has no mechanisms to specify such requirements.

Exogenous Events

Consider a domain containing a spacecraft that collects data which can be transmitted back to Earth only at speci£c times. PDDL 2.1 forces this to be modeled using constraints on the times that communication actions can begin and end. This must be accomplished by using a conditional effect that enumerates, in the condition, the possible start times for the communication windows. For instance, suppose that communication windows are open from times 1 to 5, and from times 7 to 10. The model in Figure 2 describes how this works. Notice also that we have a function compute-open-window-duration that determines how long the action takes based on the start time.

While this enforces the correct behavior in the sense that communication windows are open at the correct times, it doesn't actually have the desired semantics; if a use-window action occurs at a time when no window is available, the action occurs and uses time, but the communication window is not open for use, when in fact we would like the action to fail. Furthermore, this approach requires rewriting the model for each new situation. In particular, the conditions and the function compute-open-window-duration now must be revised for each new set of communication windows desired.

The problem is that PDDL does not allow the user to specify specify *exogenous events* that are known to be in the plan. For a treatment of exogenous events in temporal planning we refer the reader to (Smith & Jonsson 2002). Exogenous events require no explanation, and may establish states that can be used by other actions. Often, a null action is assumed to have established all the propositions in the initial state; the ability to include many such exogenous actions at different times is therefore a natural extension to planning domain initial states. Including exogenous events will make domain models more general; for example, the constraints in the satellite domain governing communication activities can be removed, and a general condition establishing the existence of a communication window can be used instead. The initial

```
(:durative-action use-window
:parameters (?s - ground-station)
:duration (= ?duration (compute-open-window-duration(?s ?start))
:condition (at start (closed ?s)))
:effect (and (when (or (and (> (?start 1)) (< (?start 5)))
    (and (> (?start 7)) (< (?start 10))))
    (over-all (in-use ?s))
    (at end (not(in-use ?s))))))
```

Figure 2: Exogenous Events in PDDL 2.1

state will contain a listing of all communication windows.

Continuously Varying Quantities

Suppose we have an aircraft that can be refueled in ¤ight. We would like to model the amount of fuel in the fuel tank while the aircraft is both consuming fuel and being refueled. As pointed out in (Fox & Long 2003), PDDL 2.1 offers two options for modeling this domain; discrete durative actions and continuous durative actions. Discrete durative actions force all modi£cation of the fuel amount to occur at the end of the action. Under some circumstances this assumption eliminates plans that are legal. Suppose the plane has 5 units of fuel left when the refueling action begins, and consumes 1 unit of fuel per unit of time. Suppose further that the refueling operation adds 20 units of fuel but takes 10 units of time. Modeling with discrete updates, the plane would be out of fuel before the refueling action £nishes.

Continuous durative actions allow the fuel state to be continuously updated, and so the situation described above can be avoided, since the net fuel change is to add one unit of fuel per unit of time during refueling. This increased power comes at a high price. The function governing the amount of fuel must be evaluated at arbitrary points during the action, and this imposes strict requirements on concurrent actions updating the same numeric quantity in the plan.

The requirement for continuous updating of variables results in complex and overly restrictive semantics. It is unnecessary to allow such unrestricted access to continuously varying quantities. First, (Fox & Long 2003) indicate that plan validation only checks the values of continuously varying quantities at £nitely many points, which implies that high order nonlinear functions cannot be validated. Thus, the power of the approach is not actually used. It is not clear whether such high order variations are simply forbidden, or whether model correctness is sacri£ced in such cases. Second, modelers usually have a good idea of the conditions under which values of variables are needed, and can build the models to correctly account for these situations. As pointed out in (Fox & Long 2003), an alternative model is possible in which the correct value of the fuel was only guaranteed to be visible at the start and end of the action. The action of refueling would change the rate of fuel consumption, necessitating a new action, Flying-and-Refueling. After the completion of the refueling, if further ¤ying was needed, then Flying would resume.

As discussed in (Fox & Long 2003), a model such as this one is weaker than the model using continuous durative action, in the sense that more states are needed, more parameters may be needed to propagate values through Flying-while-Refueling, and there is no way to access the value of fuel in the middle of the Flying or Flying-while-refueling actions. But the need for doing so has been eliminated, because the Refueling action now changes the state to Flying-and-Refueling, and all that is needed is the value of the fuel level at the end and beginning of the actions to ensure the fuel value is propagated correctly. The arguments in (Fox & Long 2003) indicate that all possible concurrent actions affecting fuel may need to be considered. While this is true, the approach taken in PDDL 2.1 actually infers these concurrent actions during the plan validation phase, and assumes that checking the endpoints of the actions is suf£cient to validate the constraints. In situations where this is not suf£cient, the modeler *must* take the burden on, to the extent they see £t when modeling the application. Thus, the PDDL 2.1 approach makes the commitment to model £delity for the modeler, and is inappropriate for more complex cases.

Resources

PDDL 2.1 uses numeric expressions to model resources. We have seen examples of resources modeled this way in Figure 3. However, this approach makes it dif£cult to do some very useful reasoning about resources. Techniques like edge £nding (Baptiste & Pape 1996) and resource envelopes (Muscettola 2002; Laborie 2003) require an explicit notion of activities using resources in order to work. In addition, modeling resources solely through numeric expressions tends to hide information from humans reading models, as well as forcing modelers to hide obvious resources in the model by using the numerical expressions. While techniques like TIM (Long & Fox 2000) can be used to infer the presence of resource behavior in planning domains, we feel there are signi£cant advantages to explicitly declaring these parts of domain models. Note that complex numeric expressions may still be necessary to determine the actual amount of resource consumption or production; for example, a model of solar panel power production will require complex numerical constraints to determine the actual impact on the resource. However, an explicit declaration of resources and explicit declaration of resource use by actions can be bene£cial.

In£nite and Dynamic Domains

In order to ensure that the number of actions and propositions is £nite, PDDL permits only a £nite number of objects, which must be explicitly enumerated, and does not allow arguments to actions or predicates to include numeric expressions (numbers being the only non-£nite domains permitted

```
(:durative-action ¤y
:parameters (?x - airplane ?y - waypoint ?z - waypoint)
:duration (= ?duration (travel-time ?y ?z))
:condition (and (at start (at (?x ?y)))
    (over all (in¤ight ?x))
    (over all (>= (fuel-level ?x) 0)))
:effect (and (at start (not (at ?x ?y)))
    (at end (at (?x ?z)))
    (at start (in¤ight ?x))
    (at end (not (in¤ight ?x)))
    (decrease (fuel-level ?x) (* #t (fuel-consumption-rate ?x)))))

(:durative-action midair-refuel
:parameters (?x - airplane)
:condition (in¤ight ?x)
:effect (increase (fuel-level ?x) (* #t (refuel-rate ?x))))
```

Figure 3: Flying and Refueling in PDDL 2.1

in PDDL). It also forbids functions that return objects, which could be used to introduce in£nitely many new objects. The justi£cation for these restriction is that many planners rely on being able to enumerate the actions and propositions in a planning problem.[1]

Since PDDL is designed for the planning competitions, tailoring it to the limitations of the planners competing is reasonable. However, from the perspective of modeling certain real-world domains, having such a requirement encoded in the language de£nition is problematic. For example, (Fox & Long 2003) point out that it is impossible to write a PDDL action to ¤y at a certain altitude. Indeed, an action to drive at a given speed or a given distance would also be ruled out for the same reason.

This requirement also makes PDDL unsuitable for modeling software domains. Software domains include information integration, web services, data processing, and other domains where the agent interacts in a software environment. These domains are typically characterized by a large, incompletely known and often dynamic universe. Since PDDL 2.1 was not designed to handle sensing, we will defer the discussion of incomplete information. Instead, we focus on dynamic universes. In PDDL 2.1, actions that create new objects must be modeled by enumerating all objects that might appear during planning ahead of time, either explicitly or implicitly. For example, in the Settlers domain, newly created machines are modeled using an integer counter. Such an approach is inadequate for describing software domains. For example, consider the following command, which creates a new archive of the £les in directory `~/papers`:

```
zip papers.zip ~/papers
```

This action fails to conform to PDDL restrictions in two ways; £rst, it creates a new object, the archive `~/papers.zip`. Second, one of its arguments is a string, which is not a £nite type. Actually, both arguments are strings, but one of them, `~/papers`, designates an existing object, a directory, and the number of directories is £nite. Following the general advice in (Fox & Long 2003), we could use the directory as an argument, rather than referring directly to its pathname, so that argument would be £nite. However, the other argument designates a £le yet to be created, so there is no existing object for which it is an attribute.

One might insist that such open-ended choices in action selection create an unreasonable burden for the planner, but nothing could be further from the truth. Either the choice will be constrained by the problem speci£cation, in which case there may be no choice at all, or it will not, in which case the choice doesn't matter; any random string will suf£ce. From a constraint reasoning perspective, it is a trivial problem.

String and numeric arguments are ubiquitous in software domains. In addition to £le creation, many image processing commands take numeric arguments, such as thresholds, values for scaling, rotating, brightening, compression factors, etc., and positions for cropping or overlapping images. None of these values are attributes of existing objects, but they are controls that the planner should be able to set, because whether, and how well, the plan achieves the goal depends on the values of those numeric arguments. For example, a user may want to scale an image to just £t on her screen, maintaining the same orientation and aspect ratio. The appropriate scale value depends on the horizontal and vertical extents of the image and of the screen. Another user may want to combine two images which are at different resolutions. Doing so will require scaling one of the images so that its resolution matches the other; the appropriate scale value depends on the resolutions of the two images, and possibly additional constraints, such as memory and the resolution of the £nal image.

One could imagine handling the object creation by listing, in advance, all of the objects that could be created. For example, we could have a few hundred "blank" £le objects, and instantiate them as needed when new objects are created. However, this is not just inelegant and inef£cient; it is also inadequate. Consider the reverse of the archive creation action above: archive extraction:

[1] A philosophical argument is also offered: that there are only £nitely many objects in the world; we agree that this is technically true, but for all practical purposes it is false, and the number of possible actions in many worlds of interest is essentially in£nite.

```
unzip papers.zip
```

This action will create new copies of every £le in the archive `papers.zip`, preserving the original directory structure from `~/papers`, but rooted in the current directory. Since a single action can create an unbounded number of new objects, listing all of the new objects up front is clearly infeasible.

(Fox & Long 2003) raise the concern in connection with in£nite domains that extensional interpretations of quanti£ed preconditions are no longer possible. This is not a problem in the examples we have discussed because, although domains are dynamic, in any given state, any object domain is £nite and can be directly determined from the execution trace that led to it. In fact, there is no way to introduce in£nite numbers of new objects unless actions can have an in£nite number of effects, which impossible to describe unless we already allow quanti£cation over in£nite domains. In other words, there is no way to get quanti£cation over in£nite domains unless we already have it.

A related issue is that an extensional interpretation of universally quanti£ed *goals* may not be possible, because the universe at the time the goal is achieved is not known at planning time, even if the agent has complete information and all actions are deterministic. Indeed, there could be many goal states with quite different universes, depending on the execution traces followed to reach the goal. However, this can be solved quite simply by interpreting the Herbrand universe with respect to the initial state, as is done in (Golden & Weld 1996).

Domain Description Languages: An Ontological Approach

The essence of modeling is abstraction, and the essence of abstraction is simpli£cation – omitting details so that the model is simpler than the thing being modeled. Choosing the right abstraction for a problem makes the problem much easier to solve. Domain description languages should enable modelers to abstract away details of planning domains that they feel are irrelevant to the task at hand.

There is an implicit agenda in the expansion of PDDL to gradually encompass more and more features that are needed for various planning domains. One possible conclusion of this is a grand-uni£ed domain description language (GUDDL). As we have pointed out in this paper, some of the problems with PDDL 2.1 stem from an attempt to shoehorn time and resources into a STRIPS-based language. We anticipate similar problems as other features are encompassed. Domain modelers will tend to reject a language with unneeded features if the presence of those features proves to be a burden, either in increased computational complexity or increased modeling dif£culty.

STRIPS and earlier versions of PDDL impose one set of abstractions: instantaneous action, the STRIPS assumption concerning persistence of states, and so on. Planning frameworks like CAIP (Frank & Jónsson 2003) impose different abstractions, such as the failure to distinguish state and action. PDDL 2.1 offers yet another set of abstractions, the ability to assert only local temporal constraints and tying temporally extended states to actions with duration, and the freedom from modeling the interaction of some concurrent actions that modify the same quantities. Incomplete information, offers additional options for abstraction. Some representations opt for a list of all possible states, others for a probability distribution over all possible states, in which the underlying representation is propositional. Still others reject the propositional abstraction, to allow sensors that return (possibly continuous) values, but give up explicit case analysis afforded by enumerating states.

It is unlikely that a modeler will model a behavior in two different ways in the same model. Having a language that supports different abstractions of the same underlying concept also makes the language clumsy to use and makes model validation more dif£cult. While some of the abstractions are hierarchical, forcing a planning domain to support the most concrete leads to both inef£ciency and frustration on the part of domain modelers who don't use the power of the language. Furthermore, it is unlikely that different abstractions will be needed in the same model [2] A domain modeler is unlikely to want to put both continuously changing quantities and discretely changing quantities into the same model, for example. That same modeler, however, may want unary resources in a model which also has continuously changing quantities. In order to unify the languages, the resulting language will have to be less abstract, and the language will become more unwieldy.

A Common Core

As an alternative to a single language for all planning domains, we propose a common core for use in many planning domain description languages. The core must contain the essential elements of all planning domains, and provide a common set of concepts that can be used to develop many planning languages. These languages can have different syntax, and different underlying implementations that play to the strengths of the particular additional components, but depend on the same set of underlying ideas.

We believe that all planning domains require the following components:

1. A notion of *state*. States must be allowed to contain *numerical arguments*, but are fundamentally discrete statements about the world.
2. A notion of *objects* or *attributes*, which take on states.
3. A notion of the conditions governing *state transitions*. States may either end on their own or be terminated by an event or action. The rules governing these state transitions must be encoded in the domain description. Sometimes these transitions may be uncertain, and sometimes they may be conditional, but they must be described.
4. A notion of the *requirements* states impose on plans. States must be explained; either an event establishes them or they are exogenous. Furthermore, states generally impose conditions on the plan. Again, sometimes the requirements may be conditional.

Modelers must describe the set of objects that exist in the world, and enumerate the states they can

[2]A potential exception to this is uni£ed agent models, in which models of different levels of abstraction must be coordinated. However, it is unlikely that the same planner will work with the different levels of abstraction.

take on. For example, a satellite object may take on the states Take-Picture, Idle, Communicate or Slew. This is a generalization of the idea that propositions are either true or not true at any given instant. By declaring the set of states an object can take on, the modeler also declares a number of mutual exclusion constraints. That is, objects can be in only one state, and the set of possible states is enumerated in the model. Notice how the semantics of negation are affected. In the example of the satellite domain, if a satellite is not Idle then by closure it must be in one of the other states.

A state takes the form P$(x_1...x_n)$ where P is a predicate and x_i are parameter variables. We will refer to the additional variables o_P, s_P and e_P, the object, start and end variables of the state. Extending the set of variables this way makes it easy to post precedence constraints among states, and also to make decisions about what object of a class of objects takes on a required state. The constraint $P_s < P_e$ is implicitly understood to hold for all states. We view these additional variables as being implicit parameters to all states.

Domain Axioms provide the means to explain states, assert the conditions that states impose on plans, and describe the rules of state transitions. This is accomplished using *constraints* on the variables of states. Domain axioms take the form: P$(x_1...x_n)$ $\wedge G(X) \rightarrow$ Q$(y_1...y_j)$ $\wedge H(Z)$ where $X \subset \{x_1...x_i\}$ and $Z \subset \{x_1...x_i, y_1...y_j\}$, G is a set of conditions and H is a set of constraints. If a state P is in the plan, then some other states must be in the plan, and some constraints must hold among the variables representing those states. Notice that the states that must be in the plan are not necessarily new states; they can be states established some other way. Thus, planners must decide whether to reuse existing states or establish new state instances. The conditions G allow us to specify that some of the variables in P must take on certain values for the axiom to apply. The constraints in H allow us to impose limitations on the possible ground states Q that can be in the plan along with P.

The conditions can be used to dictate the transitions between states. Constraints can be posted among the parameters to limit the legal sets of predicates as well as imposing ordering constraints among the states in the plan. As an example, suppose that in the satellite domain a Take-Picture can be followed by another Take-Picture or an Idle state. The rules

Take-Picture$(p,s) \wedge$ eq$(s,$take-picture$) \rightarrow$
  Take-Picture(q,t),eq(e_{tp1}, s_{tp2})
Take-Picture$(p,s) \wedge$ eq$(s,$idle$) \rightarrow$
  Idle(),eq(e_{tp}, s_i)

ensure these conditions hold.

Constraints are also a natural way to model both disjunctive preconditions and conditional effects. For example, the rules

Take-Picture$(p,s) \wedge$ eq$(p,$idle$) \rightarrow$
  Idle(), eq(s_{tp}, e_i)
Take-Picture$(p,s) \wedge$ eq$(p,$warmup$) \rightarrow$
  Warmup(), eq(s_{tp}, e_w)

indicate that two possible preconditions can hold for a Take-Picture action, either an Idle action or a Warmup action. This easily enables back chaining from exogenous events.

Constraints also replace numerical expressions in PDDL 2.1. Let us consider the in-¤ight refueling model of Figure 3. This would be modeled in the following way: the action

Fly(s,i,c,y) $\wedge$ eq$(s,$refuel$) \rightarrow$
  Fly-and-refuel(t,j,d,b),
fuel-cons(s_f, e_f, i, c),
  eq(j,c), eq(y,b)

computes the fuel consumption for the Fly action, which determines how much fuel is available when the refueling begins. We post equality constraints to ensure that the destination of the original Fly operation persists in the new action. The Fly-and-Refuel action looks similar:

Fly-and-Refuel(s,i,c,y) $\wedge$ eq$(s,$fly$) \rightarrow$
  Fly(t,j,d,b), fuel-prod(s_f, e_f, i, c),eq(j,c),eq(y,b)

In this case, we post the constraint that fuel is produced instead of consumed, but otherwise the state axioms look very similar.

Exogenous events are simply assertions that actions take place in a plan. One way of thinking about exogenous events is that they are simply a set of simple domain axioms that always hold. Since they are volatile in the same way that goals and initial states are, they properly belong in the initial state £le, It is very convenient, however, that we can express them using the same underlying concepts that we use to express the domain axioms. Returning to the communication windows example, we can express the assertion that a communication window is in a plan as follows:

TRUE $\rightarrow$ Comm-Window()

and constraints that a communication window is followed immediately by a closed communication window are written

TRUE $\rightarrow$
  Comm-Window(), No-Comm-Window(),
eq(e_c, s_n)

Pros and Cons of a Constraint-Based Representation

A number of aspects of PDDL 2.1 make it dif£cult to build planners that work by means other than progression. For example, suppose that a goal is to ¤y an airplane to a city, but nothing in the goal speci£es the remaining fuel. A progression planner can simulate the Fly action with the current fuel and check for action success. However, a regression planner must £gure out how to invert the functions in the domain axioms to determine the minimum amount of fuel needed to perform the action in the city of origin. Constraints make this easier, since they are simply relations on the legal values of the variables. In a sense, however, this moves the problem to the underlying support system to enforce the relation correctly. However, this is not required. Domains can be written that involve only successor state axioms, or only involve explanatory axioms. Thus, if a modeler knows that only progression planning is needed, only the successor axioms need to be put in the model.

As with PDDL, generic states can be introduced without £xing the entity that takes on that state. Since this is just another variable, it can be constrained just like any other variable. However, as we said earlier, mutual exclusion is enforced on objects as a part of the semantics of objects. For PDDL, this must be done by other means, either using hand-coded domain axioms or propositions or numeric expressions to simulate unary resources.

PDDL uses the STRIPS axiom to ensure that propositions that are not negated persist in time. This is a little harder to do using our framework. Since properties of objects are manifested as parameters in the states, we need to ensure they are propagated from state to state using equivalence constraints, as we saw in the `Fly-and-Refuel` example.

PDDL actions can change the value of many propositions at once. Synchronizing state changes using the concepts we describe is also simple. We can write a rule that forces many objects to change their states all "simultaneously".

An advantage of our approach over PDDL is that we can write rules that require unconditional state changes. However, in some cases, we are actually forced to do so; an example is "idle" states where we would like to persist some state information.

We have eliminated explicit actions from our representation. Part of the reason for this is that, when states have duration, there is a blurring of the distinction between temporally extended states and actions with duration. In many domains, some properties that appear "static" are really "active"; a spacecraft pointing at Earth is performing many functions in order to do so, for example. Finally, some states may only hold for a short time, as opposed to continuing inde£nitely. Since actions can be mimicked using parameters of states, and since most propositional planners assume actions to be instantaneous, we feel this imposes no great burden. As we discuss later, actions can be introduced at the syntactic level if desired. However, the underlying semantics is concerned only with state transitions.

Extensions

How can the core be extended? We describe three principal extensions: states with temporal extent, uncertainty, and dynamic domains. All of these extensions are very natural and the fundamental concepts we have described above make it easy to create languages that support these features.

State Duration and Metric Temporal Constraints

States can be extended to have duration, and constraints then govern duration and the temporal relationship between states. As an example consider the `Take-Picture` state. Suppose its duration is 24 seconds. Then we have the following rule: `Take-Picture`$(p, s) \rightarrow$

`addeq`$(s_{tp}, 24, e_{tp})$

States are now more properly called *intervals*. Note that this is a very natural extension given the representation described previously; we merely add more constraints on the start and end variables of states.

More generally, we can post any constraints in Allen's algebra. For example, consider the satellite domain in which the `Take-Picture` state required the satellite to be `Stable` for 5 seconds before and after the action. Consider the domain description in Figure 1. Compare it to the following:

`Take-Picture`$(p, s) \rightarrow$
`Stable`(t), `eq`(s, t),
`addeq`$(s_S, -5, s_T)$, `addeq`$(e_S, 5, e_T)$

We can concisely express the constraint that a `Take-Picture` state requires a `Stable` state that "contains" it, and express the exact constraints that must hold between the temporal variables of the states. Furthermore, we can also ensure not only that some state occurs in the plan, we can ensure that it happens at a particular time.

If states have duration, we can no longer employ the STRIPS axiom, States do not necessarily persist inde£nitely; we must write the successor axioms and frame axioms for all states. However, this does not impose a serious burden on the modeler in most cases. A domain axiom can indicate that a particular state can last inde£nitely, but their successors must be enumerated in case the state is terminated. For example, consider the `Idle` state in the satellite example. In the event that a state terminates, we must describe what states can follow it. Termination is accomplished by assigning or constraining the duration of the state. De£ning successors can be done a number of ways, but an easy way is to use a parameter of the state to de£ne the possible successors, then use conditions as we have described in previous examples. The rules would look like this:

Uncertainty

Uncertainty can be added in several different ¤avors to accommodate the needs of the domain. For example, in a contingency planning context, one might only wish to provide the set of possible outcomes. Those wishing a description more like MDPs can provide probability distributions over action transitions. If we revisit the satellite domain, we see that the rules need to be augmented in these cases. Suppose that trying to take a picture may fail because the shutter does not open. We can do so by introducing a special set of world-choice variables for each state, which are "set" by the world. For example, suppose the Take-Picture action either results in a Camera-Ready state or a Camera-Broken state, conditional on an outcome, $!o$, which the planner has no control over:

`Take-Picture`(p, s)
$\wedge$ `eq`$(s,$`take-pic`$)$, `eq`$(!o,$`ready`$)$ $\rightarrow$
`Camera-Ready()`, `eq`(e_{tp}, s_r)
`Take-Picture`(p, s)
$\wedge$ `eq`$(s,$`take-pic`$)$, `eq`$(!o,$`broken`$)$ $\rightarrow$
`Camera-Broken()`, `eq`(e_{tp}, s_b)

A richer representation of uncertainty allows us to specify a probability distribution over possible outcomes. We can augment the above example by associating probabilities with the different values of the outcome variable $!o$. Notice we need only do this for successor state transitions, not explanatory axioms.

A more complex task is to handle continuous probability distributions over the outcomes of actions. Uncertainty can be represented in terms of unknown values of variables. For example, uncertainty over the start time of an event can be expressed as an interval representation for the start-time variable. Again, we must take care to distinguish between uncertainty, where the world chooses, and temporal ¤exibility, where the agent chooses. More sophisticated representations can add probability distributions over values in the interval. For example, if the `Take-Picture` action results in an uncertain amount of onboard storage use, we can imagine extending the set of constraints to involve continuous probability distributions as constraints over quantities. Sensing actions can constrain the parameters of the distributions, for example. However, the fundamental notion of constraints over variables in the states still holds.

More importantly for software domains, we can represent uncertainty over the value of an object attribute, such as the pathname or size of a £le. Here we see a signi£cant advantage over propositional representations, because these attributes have in£nite domains; representing the possibilities as a list of worlds is impossible. Instead, we leave the domain of the variable open, to indicate that it could have any value, or partially open, to indicate that it is restricted to a particular subset of values.

In addition to uncertainty over the attributes, we can represent uncertainty over the objects themselves using the same representation. It is not necessary to list all objects that could exist in the world; it is suf£cient to represent the actions that can discover new objects and dynamically introduce new variables as needed to describe new objects as they are discovered. We can represent sensors that return arbitrary numbers of new objects by making the world-choice variables $!o$ universally quanti£ed (Golden & Weld 1996; Golden & Frank 2002).

Dynamic Domains

Dynamic domains arise both in the context of sensing, when a new object in the world is detected, and object creation, when an action in the plan leads to the creation of new objects whose states must be reasoned about. We can handle sensing and object creation using a similar approach. A newly created value is similar in most respects to a newly sensed value, the differences being that, in the case of object creation, the world changes and the planner has some control over the outcome. We can represent a new object, such as the output of a data-producing action, as a variable whose value is a skolem function of the corresponding action. As in the case of sensing, if the number of objects that will be created is unknown (because it depends on an unknown number of inputs, for example), we can represent the effect using universal quanti£cation, where one variable is used to represent a set of objects.

A simple extension to the form of domain axioms enables this. Recall that domain axioms can lead to the creation of a new state for an object, if an existing state isn't appropriate. Thus, there is already precedent for constraints that hold to justify the existence of a new state. We can extend the form of domain axioms to enable the creation of new objects as well. For example, consider the zip £le creation action. Let us suppose that the states a zip £le can be in are `Idle`, `Compressing`, `Uncompressing`, `Moving`, and the properties of interest of zip £les are its size, whether or not it is compressed (represented by a boolean) and its location. We can write this as follows:

`Zip`$(l,p) \rightarrow$
  `new Zip-File, Idle`(m,s,c)`,eq`$(c,$`false`$)$
  `zipsizeof`(s,p)`,eq`(m,l)`,eq`(s_i,e_z)

The keyword `new` indicates that the zip £le we are asserting properties of is created and is like the approach used in (Golden 2002). The semantics of this can ensure that no state before the time of creation can be asserted. However, we can also impose the usual constraints on the `Idle` state of the £le, along with asserting the £les initial size, location, and compressed state.

Syntax

The fundamental construct we have used in the descriptions above is that the presence of a state in a plan implies some other states must exist, and that there are some constraints. We can wrap these ideas in a number of convenient syntactic constructs. We will describe a variety of these in this section.

We will begin with simple domains where states do not have duration and metric temporal constraints are not used. We can use syntax that posts ordering constraints on the states directly:
$\mathtt{P} < \mathtt{Q}$ and translate this into the constraints on variables. If temporal constraints and states with duration are used, we can use the Allen's algebra names or other convenient labels to express temporal constraints. In the case of the constraint that the satellite must be stable while taking pictures, this constraint is written

`Take-Picture`$(p,s) \rightarrow$
  `contained-by[5][5] Stable`(t)

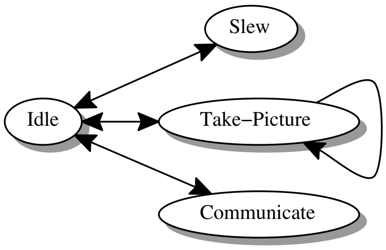
Equivalence constraints can be posted by simply using the same variable names in the parameter lists of the states.

For those who want to build models with a distinction between state and action, this can be accomplished. Actions would depend on objects being in particular states, and would ensure that some objects have new states. A simple transformation would augment each state with action parameters and the axioms can be rewritten to ensure that the proper constraints are posted among the variables of different states.

As we said previously, since properties of objects are represented by parameters of states, a mechanism is needed to propagate values to states where they are involved in constraints. However, syntax can conceal these details from the modeler. For example, objects can be created with a £xed set of variables, and the states can use these variable names in constraints. The underlying reasoning system can then decide whether new variable instances are needed and post the appropriate constraints. In the satellite domain, `Idle` states normally would propagate the amount of data in the onboard storage unit to the next `Take-Picture` or `Communicate` state. However, the action need not name the variable representing the data amount, and the underlying system would simply use the variable representing the last computed quantity in the constraint involving the next state. Notice that this syntax is similar to the PDDL 2.1 syntax, but with a different interpretation.

The astute reader will notice that, since mutual exclusion is enforced on object states, that state machines or timed automata are a good representation for many planning domains. The transformation between these representations and our fundamental language of states and constraints is also very straightforward. Rules relating the states of different objects are represented by synchronizations across different state machines.

As we said previously, domain axioms can be thought of as implications that always hold. However, another way of



Idle → Slew
Idle → Take−Picture
Idle → Communicate
...

Figure 4: A simple state machine representation of the satellite domain. The rules implied by the state machine appear below the £gure.

thinking about them is as partial plans. As such, we can assert that actions take place and have constraints among their variables, without deciding when they take place, or even whether they are ordered or not. Syntax describing partial plans can take a wide variety of forms.

Finally, resource declarations can be added to a language to augment numerical constraints, enable technologies like edge £nding and envelope calculations, and to add descriptive clarity to model de£nitions. With an explicit resource declaration, we can replace axioms designed to enforce mutual exclusion with a unary resource shared by many actions, as well as numerical expressions meant to simulate resources.

For example, consider the case of a unary resource, stability, used by 5 different actions: three Take-Picture actions (one for each of three instruments on a satellite), communication, and slewing. All the actions but the slewing action require stability, and slewing makes the satellite unstable. We might write this model as follows:

`Resource unary stability`
`Take-Picture(`p, s`)` $\rightarrow$
  `uses Stability (`$s_t - 5, e_t + 5$`)`
`Communicate()` $\rightarrow$
  `uses Stability (`s_t, e_t`)`
`Slew()` $\rightarrow$
  `uses Stability (`s_t, e_t`)`

Each state now declares how it uses the resource. The usage time can be constrained using mathematical functions of the start and end times of the activity in any way the domain modeler sees £t.

Now consider a mutli-capacity reusable resource such as power. Resources now must declare their resource impact as well as the time span during which the resource is affected. As an example:

`Resource multi reusable power 5`
`Take-Picture(`p, s`)` $\rightarrow$
  `uses power (`$s_t - 5, e_t + 5, 5$`)`

Finally, we consider renewable resources, where each activity can consume the resource or produce the resource. In this case, we must allow for the possibility that an activity could change the resource in different ways at different timepoints, in general.

`Resource multi renewable power 5`
`Take-Picture(`p, s`)` $\rightarrow$
  `uses power (`$s_t - 5, 5$`)`

This looks quite similar to the declaration of a function in PDDL, but there is an important difference: it is easier to understand that the resource is a complex, ¤exibly scoped constraint that can be reasoned about as a single entity. This simpli£es modeling as well as revealing the reasons for mutual exclusion of actions. The computational burden is wholly shifted to the implementation, where it can be ef£ciently handled in any way the implementer sees £t. All of these declarations can be converted into simple arithmetic constraints, or they could be used as the input to edge £nding, envelope calculations, or other sophisticated techniques.

A £nal syntax issue is that of functional representations versus object based representations. PDDL 2.1 uses a functional representation, and allows objects to be passed as arguments to functions. Other planning domain languages use a notion of objects with attributes, where attributes can be accessed using syntax that resembles that in object oriented programming languages. Neither of these approaches is fundamentally incompatible with a constraint-based representation such as the one we have proposed. The two approaches offer different representational transparency in the model and in the way in which planners access the information, but can represent the same things.

A Challenge for the Community

In this paper, we do not advocate a single planning domain description language. Even though the fundamental concepts we have described appear quite general and powerful, it would be easy to create a single, very clumsy language supporting many features using these concepts. However, we believe that using these concepts as a starting point will make it easier for language designers to extend the basic language in a wide variety of ways and create good languages for accomplishing many modeling tasks.

Several existing plan domain description languages make use of some of the ideas presented here. Numerous languages have more ¤exible temporal representations (Jónsson *et al.* 2000; Smith & Jonsson 2002), use constraints rather than functions (Frank & Jónsson 2003), and use dynamic domains (Golden & Frank 2002). All of these languages have their pros and cons. Language designers should be sensitive to the strengths and weaknesses of these languages for the various purposes they are used for, and consider how the language is likely to be used. The challenge for the planning community is not to search for one language that £ts all needs, but to search for the core elements of languages that are most suitable for modeling different planning domains.

References

Baptiste, P., and Pape, C. L. 1996. Edge-£nding constraint propagation algorithms for disjunctive and cumulative scheduling. In *Proceedings of the Fifteenth Workshop of the U.K. Planning Special Interest Group*.

Fox, M., and Long, D. 2003. Pddl 2.1: An extension of pddl for expressing temporal planning domains. *Journal of Arti£cial Intelligence Research*.

Frank, J., and Jónsson, A. 2003. Constraint based attribute and interval planning. *Journal of Constraints*.

Golden, K., and Frank, J. 2002. Universal quanti£cation in a constraint-based planner. In *Proceedings of the Sixth International Conference on Arti£cial Intelligence Planning and Scheduling*.

Golden, K., and Weld, D. 1996. Representing sensing actions: The middle ground revisited. In *Proc. 5th Int. Conf. Principles of Knowledge Representation and Reasoning*, 174–185.

Golden, K. 2002. Dpadl: An action language for data processing domains. In *Proceedings of the 3rd NASA Intl. Planning and Scheduling workshop*, 28–33.

Jónsson, A. K.; Morris, P. H.; Muscettola, N.; Rajan, K.; and Smith, B. 2000. Planning in interplanetary space: Theory and practice. In *Proceedings of the Fifth International Conference on Arti£cial Intelligence Planning and Scheduling*.

Laborie, P. 2003. Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. *Arti£cial Intelligence* 143(2).

Long, D., and Fox, M. 2000. Recognizing and exploiting generic types in planning domains. In *Proceedings of the Sixth International Conference on Arti£cial Intelligence Planning and Scheduling*.

Muscettola, N. 2002. Computing the envelope for stepwise-constant resource allocations. In *Proceedings of the Eighth International Conference on the Principles and Practices of Constraint Programming*.

Smith, D., and Jonsson, A. 2002. The logic of reachability. In *Proceedings of the Sixth International Conference on Arti£cial Intelligence Planning and Scheduling*.