A Domain Description Language for Data Processing

Keith Golden
NASA Ames Research Center
MS 269-2
Moffett Field, CA 94035
kgolden@email.arc.nasa.gov

Abstract

We discuss an application of planning to data processing, a planning problem which poses unique challenges for domain description languages. We discuss these challenges and why the current PDDL standard does not meet them. We discuss DPADL (Data Processing Action Description Language), a language for describing planning domains that involve data processing. DPADL is a declarative, object-oriented language that supports constraints and embedded Java code, object creation and copying, explicit inputs and outputs for actions, and metadata descriptions of existing and desired data. DPADL is supported by the IMAGEbot system, which we are using to provide automation for an ecological forecasting application. We compare DPADL to PDDL and discuss changes that could be made to PDDL to make it more suitable for representing planning domains that involve data processing actions.

1 Introduction

Earth-observing satellites return terabytes of data per day, providing global daily coverage across multiple spectral bands at a variety of resolutions. These observations can be used in countless ways: to monitor changes in Earth's climate, assess the health of forests and farms, and track critical short-term events, such as severe storms. However, doing all this in a timely manner is a significant challenge, which will require greater levels of automation. To go from raw "level 0" satellite data to high-level observations or predictions such as "decreased vegetation growth" or "high fire risk" requires many data-processing steps, from filtering out noise to running simulations. There are often many data sources to choose from, and many ways to process the data to produce the desired data product. These choices involve tradeoffs along many dimensions, including data quality, temporal and spatial resolution and coverage, timeliness, CPU usage, storage and bandwidth.

We use planning technology to automate this data processing. We represent data-processing operations as planner actions, descriptions of desired data products as planner goals, and use a planner to generate data-flow programs that output the requested data. We are working with Earth scientists to provide planner-based automation to an ecological forecasting system called the Terrestrial Observation and Prediction System, or TOPS (Nemani *et al.* 2002) (http://www.forestry.umt.edu/ntsg/Projects/TOPS/). We have developed a planner-based softbot (**soft**ware ro**bot**), called IMAGEbot, to generate and execute data-flow programs (plans) in response to data requests. The data-processing operations supported by IMAGEbot include image processing, text processing, managing file archives and running scientific models. Some aspects of the planner are described in (Golden & Frank 2002).

In the course of developing IMAGEbot, we considered available domain description languages, especially PDDL, for representing data processing actions, but found them unsuitable. We discuss the features of these domains that are problematic for PDDL and the changes to the PDDL that would be needed to handle them. To deal with these issues, we developed a new language called DPADL, for Data Processing Action Description Language. We considered basing our language on PDDL, which is attractive in that it has become a standard for much of the planning community, but decided instead to base the syntax on a different widely-used language: Java. This decision was driven by practical considerations, such as our desire for the language to be usable by software developers, the appropriateness of an object-oriented language to describe the complex data structures that arise in data-processing domains, and the fact that Java is the language that both TOPS and IMAGEbot are written in, to name a few.

In the remainder of the paper, we discuss language features relevant to representing data-processing domains, how those features are implemented in PDADL, and what issues there would be in including those features in PDDL:

- First-class objects (Section 2): Data files often have complex data structures. The language should provide the vocabulary for describing these structures.
- Constraints (Section 4): Determining the appropriate parameters for an action can be more difficult than determining which action schemas belong in the plan. Parameter values can depend on other actions or objects in the plan. The language should provide the ability to specify such constraints where they are needed.
- Integration with a run-time environment (Section 4): Sensing and acting in a complex software environment requires "hooks" into that environment, both to obtain information and to initiate operations.
- Metadata goals (Section 5) and inputs (Section 7): The inputs and outputs of data-processing plans are data, so

the language should be expressive enough to describe requested and available data . Since a data file contains information about past states of the world, metadata should be able to describe how the content of the data depends on the past state of the world.

- Object creation and copying (Section 6.3): Many programs create new objects, such as files, sometimes by copying or modifying other objects. The language must provide a way of describing such operations.
- Data-flow plans (Section 8): Since the purpose of plans is to process data, they should take the form of data-flow programs, in which outputs of one action are fed into inputs of another.

At the end of each section, we present a BNF grammar covering the language elements described in that section. For example, the top-level production rule for a domain description is:

DOMAIN ::=	(TYPE | FUNCTION | ACTION | GOAL | STATE | <INLINE_CODE>)+ <EOF>

where symbols in SMALL CAPS are non-terminals, symbols in <ANGLE_BRACKETS> are terminals, and keywords are underlined.

2 First-class objects

Data files (and other entities in a software environment) typically have a complex, hierarchical structure, which can be described in terms of object composition. Representing these data structures explicitly as first-class citizens not only makes domains simpler to encode and understand, but provides valuable information to the planner. Thus, we decided that DPADL should be an object-oriented language. Although the identification of objects and object attributes is important, an object-oriented syntax, such as our Java-based syntax, is less so; the same information could be expressed in a PDDL-style relational syntax, just not as concisely.

DPADL allows the definition of new types corresponding either to structures (objects) or primitive types, such as integers or strings. The keyword for introducing a new type declaration is **`type`**. Here and elsewhere in the paper, DPADL text is rendered in `typewriter` font, and keywords are **`bold`**. We use ellipses (...) to indicate that text has been omitted for the sake of brevity. For example,

```
static type Filename extends String
```

introduces a new type, `Filename`, which is a subtype of `String`, a predefined type. The predefined types are `int`, `unsigned`, `float`, `String`, `Object` and `boolean`. The keyword **`static`** means that no instance of `Filename`, once created, can ever be changed.[1] A type that is not static is **`fluent`**.

Subtypes of `Object` may be used to represent Java objects. For example,

```
static type Tile extends Object
            mapsto tops.modis.Tile
```

means that the type `Tile` corresponds to the Java class `tops.modis.Tile`. As we discuss in Sections 4 and 6.4, the agent can manipulate Java objects in the course of constraint reasoning or action execution by executing in-lined Java code.

Alternatively, when there is a small number of instances of a type, we can define it by listing all possible instances. This is similar to enumerated types in C/C++, but without the restriction to integral values.[2] Listing values in this way is useful for constraint reasoning, since the domain of a variable corresponding to such a type can be initialized with the set of possible values.

```
static type ImageFormat =
{"JPG", "GIF", "TIFF", "PNG", "XCF", ...};
```

As in C/C++, enumerated values can have symbolic names attached to them.

```
static type ProjectionType =
{LAZEA=11, GOODE_HOMOL=24, ROBINSON=21, ...};
```

Like classes in C++ and Java, types can have attributes. For example, file attributes include pathname and owner:

```
type File extends Object {
  static key Path pathname;
  User owner;
  ...
}
```

The keyword **`key`** is used to indicate that `pathname` is a unique identifier for a file, so two files that have the same pathname must in fact be the same file. This is not correct if we access files on multiple machines, in which case we should use the host machine as an additional key.

In addition to the subtype relation, designated using **`extends`**, we can specify that one type **`implements`** another, meaning it inherits all the attributes of the other type but is not an instance of that type.[3] This is useful in cases where two objects share the same structure but cannot be used interchangeably. For example, a file archive, such as a tar file, contains records that reflect all the properties and contents of individual files, but are not themselves files. We say that `TarFile.Record` **`implements`** `File`. This is especially useful when used in conjunction with **`copyof`** (Section 6.3), since a record in a tar file can be a copy of a file, or vice versa.

When referring to an attribute of an object, we use a Java-like syntax. For example, `f.filename` refers to the `filename` attribute of the object represented by the variable `f`. Attributes can take arguments. For example, `pic.pixelValue(x,y)` refers to the value of the pixel at the x,y coordinates of the image `pic`. Although the syntax resembles that of Java method calls, `pixelValue(x,y)` is simply a parameterized attribute, and can be used in exactly the same contexts. For example,

[1] This is a departure from the Java meaning of `static`.

[2] In PDDL, all types other than numbers are effectively enumerated types, since all objects of each type must be explicitly declared. Since the Closed-World Assumption is not at all reasonable for data processing domains, DPADL does not impose this requirement.

[3] This is a departure from the Java meaning of `implements`.

```
pic2.pixelValue(x,y) = pic1.pixelValue(y, x+5);
```

describes an effect that transposes an image to the left by 5 pixels.

The object-oriented notation is convenient, but not essential. Any object description can be translated into an equivalent relational form by translating each attribute description into a relation in which the first argument is a reference to the object, the second argument is the value of the attribute, and the remaining arguments are the arguments of the attribute. From the example above, we would define a PDDL relation (pixelValueR ?image ?pvalue ?x, ?y). Additionally, the action descriptions or domain axioms would need to be modified to enforce the fact that

- Two objects are equal if and only if their key attributes are equal and
- An attribute can have only one value, so (pixelValue IMAGE-56, BLACK, 10, 10) is mutually exclusive with (pixelValue IMAGE-56, ?v, 10, 10), for all ?v $\neq$ BLACK.

Alternatively, we could provide additional syntax to convey the same information while maintaining a relational representation, as was done in the SADL language (Golden & Weld 1996).

Explicitly identifying objects is not just useful to the domain developer, but also to the planner. For example, the planner can reduce search by exploiting the fact that attributes of static objects don't change once the object is created. Additionally, Section 6.3 discusses the role attributes play when objects are copied.

TYPE	::=	(static | fluent)? type ((<IDENTIFIER> **=** **{** MEMBERS **}**) | (TYPESPEC)) (TYPEBODY | **;**)
MEMBERS	::=	((<IDENTIFIER> **=**)? LITERAL) (**,** MEMBERS)?
TYPESPEC	::=	PRIMTYPE | (<IDENTIFIER> extends TYPENAME (implements TYPENAME)* (mapsto <CLASSNAME>)?
TYPEBODY	::=	**{** (MEMDEF | CTRSPEC | TYPE)* **}**
MEMDEF	::=	(static | fluent)? key? TYPENAME <IDENTIFIER> (PARAMS)? (MEMBODY | **;**)
MEMBODY	::=	**{** (CTRSPEC)* **}**
PRIMTYPE	::=	int | unsigned | float | String | Object | boolean
TYPENAME	::=	<IDENTIFIER> | PRIMTYPE
QUALTYPE	::=	TYPENAME (**.** <IDENTIFIER>)*

3 Functions and relations

The object-attribute notation is just a special case of a functional notation, which DPADL also supports. Functions, like types, may be static or fluent. The value of a fluent function changes over time, whereas the value of a static function does not. For example,

```
fluent float temp(float lon, float lat);
```

declares a function that takes two real values, representing longitude and latitude, and returns a real value representing the temperature at that location. Functions, like attributes, may have zero arguments, in which case the parentheses are omitted. For example,

```
fluent Date currentDate;
```

specifies that `currentDate` is a fluent function taking no arguments.

Functions over objects have been mentioned as a possible future extension to PDDL (Fox & Long 2003). While that would make it easier to describe data-processing domains in PDDL, we should note that functions in DPADL are merely a notational convenience; they allow us to avoid explicitly stating the mutual exclusions to specify that, for example, a file can have only one size, but semantically they are no different from relations in which one of the arguments is restricted to a single value. In particular, they do not play the same role that functions play in first-order predicate logic. DPADL does not support domain axioms, which could be used to generate an arbitrary number of object references through repeated function composition.

To indicate that a function is undefined for particular arguments, we use the keyword **`null`** to represent invalid values. The type of **`null`** is a subtype of all types, but **`null`** will not match any value except itself.

FUNCTION	::=	(static | fluent) TYPENAME (<IDENTIFIER> | <OPERATOR>) (**(** PARAMS? **)**)? (**;** | **{** (CTRSPEC)* **}**)
PARAMS	::=	(PARAMDEF (**,** PARAMS)?) | :rest PARAMDEF
PARAMDEF	::=	QUALTYPE <IDENTIFIER>

4 Constraints

In data-processing domains, we need to be able to express thresholds, intervals over space or time, mathematical functions, and more complex calculations. In DPADL, these are all represented using constraints. PDDL supports numeric functions, which are used to specify how quantities change over time. PDADL constraints can serve the same role, but are more flexible; they can perform arbitrary calculations or sense information from the environment. However, they are also more limited than PDDL functions, in that they cannot represent and reason about quantities that change continuously over time, such as fuel (Fox & Long 2003).

Formally speaking, a constraint is simply a relation that holds over a set of variables, so we could view any functions, object attributes or types as constraints. However, thus far, we have only shown how to *declare* functions, attributes and types, not (with the exception of enumerated types) how to *define* them. To reason about constraints, we need definitions, not just declarations. For example, consider the following declaration.

```
float foo(float x);
```

Given the value of `x`, we know there must be some value `y = foo(x)`, but we have provided no way to determine what that value is. Viewing `foo` as a constraint is valid but pointless.

We provide two alternative ways of specifying the definition of a constraint; it may be selected from a library of predefined constraint definitions or defined in terms of arbitrary Java code embedded in the type and function declarations. The constraint reasoning system supports constraints over all primitive types as well as Java objects. It can also handle constraints involving universal quantification, as discussed in (Golden & Frank 2002).

Constraint definitions can only be given for statics. Any function defined as a constraint must be determined only by that constraint; no action may affect it. This restriction provides a clear division of labor between causal reasoning and constraint reasoning.

4.1 Type constraints

Formally, a type is a unary relation that is true for all instances of the type and false for all non-instances. But in the type declarations of Section 2, we did not define what those relations were. It is fine to say `Filename` **`extends`** `String`, but given a String, how do we know if it is a valid filename?

One possibility might be to define `Filename` as an enumerated type; that is, we list all valid filenames. The obvious problem with this is that there are, for all practical purposes, infinitely many of them. A better option is to specify a regular expression that concisely specifies all valid filenames:

```
static type Filename extends String {
   constraint Matches(true, this, "~[/]+");
}
```

means that filenames must contain at least one character, and they cannot contain the character '/'. In Unix, this is, in fact, the only practical limitation on filenames. `Matches` is a constraint from the constraint library requiring a string to a match a regular expression. All string constraints are actually defined in terms of operations on regular expressions, so Matches is, in a sense, the simplest. The keyword **`this`** designates an instance of the type being defined, in this case a filename.

Constraints can also be defined in terms of inlined Java code, as discussed in the next section.

4.2 Attribute constraints

We can define attributes as constraints as well. One reason for doing this is to support *procedural attachment*: specifying program code that provides the definition of the attribute. For example, if we have a DPADL object that corresponds to a Java object, we must specify what methods to call on the Java object to determine the values of the attributes as declared in DPADL:

```
static type Tile extends Object
            mapsto tops.modis.Tile {
  key String uniqueId {
    constraint {
      value(this) = $this.getUID()$;
      this(value) = $Tile.findTile(value)$;
    }
...
```

The attribute `uniqueId` is declared as a **`key`** of a (mosaic) `Tile`, meaning there is a one-to-one mapping between tiles and their unique identifiers. Given a tile, we should be able to obtain its unique identifier, and given a unique identifier, we should be able to obtain the corresponding tile. The embedded Java code provides instructions for performing these mappings. The `uniqueId` attribute of a `Tile` can be determined by calling the `getUID` method on the `Tile`, and a `Tile` object corresponding to a given `uniqueId` can be determined by calling the method `findTile`, with the `uniqueId` as an argument. The text preceding the "=" is a "signature" specifying the return value and parameters of the following Java code. The keyword **`value`** refers to the value of the attribute being defined, in this case `uniqueId`. The keyword **`this`** refers to an object of the type being defined, in this case `Tile`. Thus, **`value`**`(`**`this`**`)` means that given an object of type `Tile`, we can obtain the value of the `uniqueId` attribute by executing the following Java code (delimited by `$`). Conversely, **`this`**`(`**`value`**`)` means that given a `uniqueId`, we can find the corresponding `Tile`.

The above constraint will only be enforced if there is a singleton domain for some tile or ID variable. It is also possible to define constraints that work for non-singleton domains, by indicating that an argument or return value represents an interval (delimited by `[]`) or a finite set (delimited by `{}`). For example, one attribute of a `Tile` is that it `covers` a given longitude, latitude. Given a particular longitude and latitude, the constraint solver can invoke a method to find a single tile that covers it, but it can do even better. Given a rectangular region, represented by intervals of longitude and latitude, it can invoke a method to find a set of tiles covering that region.

```
boolean covers(float lon, float lat) {
  constraint {
    ...
    // returns the set of tiles covering
    // a given lon/lat range.
    {this}([lon], [lat], d=day, y=year,
           p=product, value)
      = {$ if(value)
             return tm.getTiles(lon.max,
                                lat.min,
                                lon.min,
                                lat.max,
                                d, y, p);
           else return null; $};
  }
}
```

In this example, the signature is more complicated. `{`**`this`**`}(...)` means that the return value of the Java code is a set (specified by `{...}`) of Tiles (specified by **`this`**). The first two arguments, lon and lat, are surrounded by `[...]`, indicating that the variable domains should be intervals. The next three arguments, d, y, and p are defined as being equal to the Tile attributes day, year and product, not shown in this example. Finally, **`value`** is the boolean value of the `covers` relation, true if and only if the tile covers the specified lon/lat.

The Java code is also more complex. Unlike the previous example, it has a conditional and an explicit return call. If **value** is true, then it returns the result of the method getTiles. Since lon and lat are intervals, we refer to their maximum and minimum values to specify the bounding box of interest. If **value** is false, it returns **null**, meaning the set of tiles could not be determined, since there is no method for returning the tiles outside of a bounding box.

4.3 Function constraints

Functions, like attributes, can have constraints associated with them, the only difference being that the constraints cannot reference the keyword **this**, because there is no object to reference. Infix mathematical operators are also functions, and they can be defined for any type, using a syntax similar to that used for C++ operator overloading. For example to specify that the "+" operator can be used to concatenate strings, as in Java, we can write

```
static String operator+ (String s1,
                         String s2) {
  constraint Concat(value, s1, s2);
}
```

where Concat is a constraint from the constraint library, specifying that the first argument is the concatenation of the remaining arguments.

4.4 Restrictions

Minimum requirements on the inlined code used to define constraints are:

1. The code may not do anything other than calculate the domain of a variable and return it. That is, it may not have any side-effects.
2. If the code is called multiple times with the same arguments, it will always return the same calculated domain. This requirement precludes using constraints to represent values that change during the course of planning.
3. If the domains corresponding to one or more of the arguments is reduced, then the calculated domain will be a subset of the original domain.

If these requirements are not met, then the results are undefined. With them, we can view each constraint as some unknown relation and the procedures as sensors that provide limited information about the extension of the relation.

```
CTRSPEC  ::= constraint ( <IDENTIFIER>
             { ARGS } _;_ ) | { ( JAVACTR )+ } )
JAVACTR  ::= ( CTRARG CTRARGS _=_
             <INLINE_CODE> _;_ )
             | ( [ CTRARG ] CTRARGS _=_
             [ <INLINE_CODE> _,_
             <INLINE_CODE> ] ; )
             | ( { CTRARG } CTRARGS _=_
             { <INLINE_CODE> } _;_ )
CTRARG   ::= <IDENTIFIER> | value | this
CTRARG2  ::= (CTRARG | [ CTRARG ] | { CTRARG })
             ( _=_ ADDITIVE )?
CTRARGS  ::= { CTRARG2 [ _,_ CTRARGS ] }
```

5 Goals

Goals are used primarily to describe data products that the system should produce. Data product descriptions should specify at least the following:

- Data semantics: the information represented by the data. That is, what facts about the world can be inferred from the data contents.
- Data syntax: how the information is coded in the data. For example, what pixel values in an image are used to represent the information.
- Time: what time the information pertains to. For example, we need to be able to distinguish between rainfall last week and rainfall last year.

Time is an optional argument of all fluents. The mapping between semantics and syntax is specified using the keyword **when**. For example, to request a file that contains gridded temperature values over a particular region, using the LAZEA projection and a particular mapping (tempEncoding) from temperatures to pixel values, we could write:

```
forall int x, int y, float lon, float lat,
       float t;
when(tempEncoding(temperature(lon, lat)) == t
     && proj(LAZEA, x, y, lon, lat)
     && 0 <= x < MAXX && 0 <= y < MAXY) {
  file.pixelValue(x, y) == t;
}
```

We will call the expression inside the parentheses following the keyword **when** the left-hand side (LHS) of the goal, and we will call the expression in the braces the right-hand side (RHS).

A key aspect of DPADL is that all data descriptions are purely causal; we describe how the data content of the file causally depends on the (earlier) state of the world. An advantage of this representation is that standard temporal projection techniques can be used to to determine how a succession of data-processing operations affect the data content of the final output.

A **when** condition describes an implication, but an implication between conditions that hold at two different times. The LHS implicitly refers to the time the goal is posted (unless an earlier time is specified), and the RHS refers to the final state (whenever the goal achieved). Because the agent cannot change the past, the only way to achieve the goal is to make sure the RHS is satisfied, subject to the conditions given by the LHS.

More formally, a **when** goal can be be described as follows. Let s_0 be the initial state and let Σ be the set of states indistinguishable from s_0 consistent with the agent's knowledge in the initial state (i.e., the set of all possible worlds). Let π be a plan consisting of a sequence of actions, and let $do(\pi, s)$ be the state reached from s by executing π. The goal **when**$(\Phi(\vec{x}))$ $\{\Psi(\vec{x})\}$ is achieved by π if $\forall \vec{x} \forall s \in \Sigma((s \models \Phi(\vec{x})) \Rightarrow (do(\pi, s) \models \Psi(\vec{x})))$.

The only formal difference between conditions in the LHS and conditions in the RHS is the time that they refer to, but this provides a sufficient foundation for describing data

goals, since the important characteristic of data is that it stores information about the past. Thus, we use temporal goals to describe how the past information of the world determines the future content of the data:

- The semantics of the desired data (e.g., temperature) is specified in terms of fluents in the LHS of the goal, because it concerns properties of the world that hold when the goal is specified (or earlier), properties that are not affected by the agent in pure data-processing domains.
- The data syntax (e.g., pixelValue) is specified in terms of static predicates in the RHS, because it concerns properties of data that may not exist at the time the goal is given, properties that must be affected by the agent to produce the requested data. Optionally, predicates describing syntax could also appear on the LHS, to represent goals of converting file formats, etc. For example, we might specify a goal of making all the red pixels in an image blue: **when**`(input.color == RED) {output.color == BLUE;}`
- Constraints (e.g., `0 <= x < MAXX`) are specified in the LHS of the goal because, being static, they must hold in the initial state and cannot be affected by the agent. Since variables involved in constraints can appear in the RHS, this is not a practical limitation.

To use these conventions in PDDL, we would need to extend PDDL to specify goals that refer to an earlier state of the world in addition to the final state. PDDL 2.1 can refer to time, but only the start and end times of actions. It would also be necessary to relax the CWA, if these domains are to be remotely interesting.

GOAL	::=	goal <IDENTIFIER> (PARAMS?) { ((output | forall | exists) PARAMS ;)* OREXP }
OREXP	::=	CONDEXP+ (|| CONDEXP+)*
CONDEXP	::=	(when (ANDEXP) { CONDEXP* } (else { CONDEXP* })?) | EQUAL ;
ANDEXP	::=	EQUAL (&& (EQUAL))*
EQUAL	::=	RELATION ((== | !=) RELATION)*
RELATION	::=	ADDITIVE ((< | > | <= | >=) ADDITIVE)*
ADDITIVE	::=	MULTIPL ((+ | -) MULTIPL)*
MULTIPL	::=	UNARY ((* | / | %) UNARY)*
UNARY	::=	(+ | - | !)? PRIMEXP
PRIMEXP	::=	(ANDEXP) | (FUNEXP | this) (. FUNEXP)* | LITERAL
FUNEXP	::=	<IDENTIFIER> ((ARGS))?
LITERAL	::=	<INTEGER_LITERAL> | <FLOATING_POINT_LITERAL> | <CHARACTER_LITERAL> | <STRING_LITERAL> | null | true | false
ARGS	::=	ADDITIVE (, ARGS)?

6 Actions

Actions can include data sources (which provide data based on the state of the world) and filters (which provide data based on their inputs), so preconditions and effects describe inputs and outputs as well as the state of the world. Additionally, actions must be executable, so the procedure for executing an action (i.e., Java code) is part of the action description.

ACTION	::=	action <IDENTIFIER> (PARAMS) { ((input | forall) PARAMS ;) | (output OUTPUTS ;) | PRECOND | EFFECT | EXEC)* }
OUTPUTS	::=	PARAMDEF (copyof <IDENTIFIER>)? (, OUTPUTS)?

6.1 Inputs, outputs and parameters

As in PDDL (McDermott 2000), actions are parameterized, and parameters are typed. In addition to ordinary parameters, two kinds of variables are recognized as unique and are treated somewhat differently; namely, inputs and outputs.

Outputs represent objects (e.g., files) generated as a result of executing the action. An output does not exist before the corresponding action is executed, and is always distinct from all other objects.

Inputs represent objects that are required by the action but are not required to exist after the action has been executed. Inputs may come from outputs of other actions or they may be preexisting objects. In the former case, all preconditions describing attributes of a given static input must be supported by the same action, since only one action can have produced the output, and once it is created, no action can change it.

Ordinary parameters are essentially like the parameters passed to method or function calls in C or Java; they refer to primitive values or objects that may exist before the action is executed and may persist afterward.

In addition to parameters, inputs and outputs, actions can refer to universally quantified variables and introduce variables corresponding to new objects with the **new** keyword, discussed in Section 6.3.

To extend PDDL to handle input and output parameters, it would be necessary to allow for object creation (which requires a dynamic universe), and to allow the values of certain variables to be unbound at planning time, provided it can be proven that they will be bound at execution time.

6.2 Preconditions

Preconditions describe the conditions that must be true of the world and of the inputs in order for the action to be executable. Thus, action preconditions need to reference the input variables and the prior world state, but cannot reference the output variables, which describe objects that don't exist in the prior state.

Low-level actions, such as filters, can be described purely in terms of the syntactic properties of the input files. For example, an image-processing operation doesn't care whether the pixels of the input image represent temperatures in Montana or a bowl of fruit. All that matters are the values of the pixels. Thus, the preconditions for these actions should refer only to properties of the data that hold in the prior state. Similarly, simple sensors (data sources) depend only on the immediate state of the world, so their preconditions should

only refer to conditions of the world that hold in the prior state.

However, some high-level actions, such as ecological models, expect their inputs to represent certain information about past states of the world, such as temperature or precipitation, so it is appropriate for the preconditions of these actions to specify the information content of their inputs, not just the structure. The descriptions the information content of these inputs will be in terms of states other than the prior state. For example, an ecological model might require a file containing temperature data from last Tuesday. In other words, preconditions, like goals, can include metadata descriptions, which are described in exactly the same way, using the keyword **when**.

The LHS of a **when** precondition, like the LHS of a goal, refers to past states. The RHS, however, rather than referring to the final state, refers to the start of execution of the corresponding action. Conventions for describing data inputs in preconditions are the same as the conventions for describing goals: The LHS specifies the semantics of the data file and the RHS specifies the syntax. Any constraints must appear in the LHS.

Preconditions are introduced with the keyword **precond**, and introduce a condition, which may be disjunctive.

PRECOND ::= precond OREXP

6.3 Effects

Effects, introduced with the keyword **effect**, are used to describe the outputs generated by an action. Outputs depend on the state of the world (in the case of sensory actions) or the inputs (in the case of filters), so effects need to be able to reference both the prior state and next state and both the input and output variables.

EFFECT ::= effect (WHENEXP)+

Conditional effects Like goals and preconditions, conditional effects are introduced using the keyword **when**, but here the LHS refers to the prior state (and input variables), not the initial state. The RHS describes the next state and output variables, so the combination of the two describes how the output depends on the input (or on the state of the world). This is no different than conditional effects in PDDL.

As with goals, there are conventions for describing data effects.

- data sources are described using conditional effects, in which conditions on the LHS are either constraints or fluents describing the state of the world and conditions on the RHS are statics describing the syntax of the output data.
- Filters are described using conditional effects, in which conditions on the LHS are either constraints or statics describing the syntax of the input data, and conditions on the RHS are statics describing the syntax of the output data.

In order to restrict the language to only describe data-processing domains, we do not allow fluents to appear on the RHS of any effect. This means that actions cannot change the world except by creating objects (e.g., files) that satisfy certain conditions based on the current (or past) state of the world. This restriction could easily be lifted, allowing us to describe arbitrary planning domains, but imposing it allows the use of specialized planning algorithms that take advantage of unique properties of pure data-processing domains. An alternative would be to run a simple preprocessor that checks whether a domain is a data-processing domain and runs a specialized planner if it is.

Every atomic RHS expression involves setting the (possibly boolean) value of a function or attribute or creating a new object. A static attribute can only be set if it is an attribute of a newly created object. Since we restrict predicates on the RHS of effects to static predicates, that means all that actions can do is produce data; they cannot change the world or alter preexisting data.

For example, to describe a threshold action, which sets output pixels to either BLACK or WHITE, depending on whether the corresponding input pixels are below or above a given threshold `thresh`, we can write:

```
action threshold (unsigned thresh) {
  input Image in;
  output Image out copyof in;
  forall unsigned x, unsigned y;
  effect when ((x < in.xSize)
               && (y < in.ySize) {
    when (in.valueAt(x, y) <= thresh) {
        out.valueAt(x, y) = BLACK;
    } else {
        out.valueAt(x, y) = WHITE;
    }
  }
}
```

The keyword **else** has the same meaning as in C or Java. The keyword **copyof** is explained below.

WHENEXP ::= (when { ANDEXP } { (WHENEXP)* } (else { (WHENEXP)* })?) | CONSEQNT
CONSEQNT ::= ASSIGNMNT | NEWDECL
ASSIGNMNT ::= CFUN (. CFUN)* (= (EQUAL | NEWEXP))? ;
CFUN ::= <IDENTIFIER> ((CARGS))?
CARGS ::= (ADDITIVE | NEWEXP) (, CARGS)?

Object creation and copying Output variables implicitly describe newly created objects, but it is sometimes necessary to explicitly refer to object creation in action effects. For example, an output may be a complex object, such as a file archive or a list, with an unbounded number of complex sub-elements. Since each of those sub-elements is (possibly) newly created, we need some way of describing their creation. We do so using the keyword **new**.

Additionally, newly created objects may be copies of other objects, possibly with minor changes. Listing all the ways the new objects are the same as the preexisting objects can be cumbersome and error-prone, so we would like to simply indicate that one is a copy of the other, and then

specify only the ways in which they differ. We do so using the **copyof** keyword.

Suppose we have an action whose input, `in`, is a collection of JPEG files and whose output, `out`, is a new collection, in which the files from the input are compressed with quality of 0.75.

```
forall Image orig;
when(in.contains(orig)) {
  out.contains
     (new Image copyof orig {
       quality = min(orig.quality, 0.75); });
}
```

When an object is copied, all attributes of the original object are inherited by the copy, unless explicitly overridden. For example, the new Image is identical to the original in every way, except in quality, which is set to 0.75. Note that this is one way in which attributes of objects are different from other relations on objects. `in.contains(orig)` is an attribute of `in`, but not an attribute of `orig`, so after `orig` is copied, `in.contains(copy)` is not true but, for example, `copy.format == JPEG` is true.

The copy and the original need not be the same type, as long as they inherit from or implement a common parent type. All attributes common to both types are copied.

Formally, we can describe new objects as Skolem functions of the actions, inputs and quantified variables that they depend on. The semantics of **copyof** can be specified in a manner similar to Reiter's solution to the frame problem (Reiter 1991). Let a be an action with an effect "**new** T n **copyof** i," and let p represent an attribute common to the type of i and type T. Let $n = sk(a,i,v)$ be a Skolem function of action a, input i and variables v appearing in a. We will write $p(n)$ to designate the value of the attribute p of object n. Let Π^a be the precondition of action a, let $do(a,s)$ be the state reached by executing action a in state s. Without loss of generality, assume that for each possible value v of attribute p, there is a single conditional effect of the form "**when** $(\gamma^v_{p(n)}(a))$ `{n.p = v;}`." If a has no direct effects concerning $p(n)$, then $\gamma^v_{p(n)}(a,s)$ is false for all v. If a unconditionally sets $p(n) = x$, then $\gamma^x_{p(n)}(a,s)$ = true. Because the effects of a are assumed to be consistent, $\gamma^v_{p(n)}(a,s)$ can be true for at most one value of v.

The successor state axiom for $p(n)$ is:

$$\Pi^a(s) \Rightarrow p(n, do(a,s)) = \begin{cases} v & \text{if } \gamma^v_{p(n)}(a,s) \\ p(i) & \text{otherwise} \end{cases}$$

That is, assuming a is executable ($\Pi^a(s)$), the value of $p(n)$ after a is executed is v if a has an effect that sets it to v. Otherwise, it is the value of $p(i)$. A similar axiom must be given for each attribute common between i and n.

The advantage of **copyof** is purely syntactic, since it could be replaced by a large number of conditional effects, one for each attribute of the object being copied. However, since the number of attributes can be quite large, the reduction in the size and apparent complexity of action descriptions can be substantial. This is exactly analogous to the advantage of the STRIPS assumption as a solution to the Frame Problem, in that we avoid specifying conditions that stay the same. The only difference is that the properties that "persist" are actually copied from one object to another. Using conditional effects would also make it harder for a planner to distinguish effects that result in progress toward some goal from those that simply propagate a condition from one file to another. Although this could be determined using domain analysis, that would be making life harder for both the planner and domain modeler with no apparent advantage for either.

```
NEWDECL    ::= new QUALTYPE <IDENTIFIER>
               (copyof <IDENTIFIER> )?
               ( ({ ( ATTRIBUTES )* } ) | ; )
NEWEXP     ::= new QUALTYPE
               ( copyof <IDENTIFIER> )?
               ( ( { ATTRIBUTES * } ) | ; )
ATTRIBUTES ::= FUNEXP = ( EQUAL ;
               | NEWEXP )
```

6.4 Execution

The action descriptions include instructions for actually executing the action. These instructions are written in Java, which enables us to write actions that correspond to any operation that can be performed by the Java runtime environment, including invoking methods on objects or making system calls. All parameters and inputs corresponding to Java objects or primitives may be referenced in the Java code, and outputs must be initialized.

We pose the requirement that the results of execution are accurately reflected by the stated effects of the action. There is, of course, no way to verify this requirement, but that's the case for execution in any planning domain.

```
EXEC ::= exec <INLINE_CODE> ;
```

7 States

A typical component of planning problems is a specification of the "initial state," from which the goal must be achieved. In PDADL, a significant amount of state information is communicated through the execution of inlined code during constraint reasoning, which can be used to "query the world" to determine the current state. However, static state information is also useful, especially metadata descriptions for stable data sources. The language provides the ability to define multiple named states through the **state** keyword. States may be thought of as dumbed-down actions that have no preconditions and can only be "executed" in the initial state. As with goals, metadata descriptions are specified using the **when** keyword. As with goals, the LHS can refer to the current state or earlier, but the RHS refers to the current state, not the final state. The conventions for describing the semantics and syntax of data are the same as they are for goal descriptions.

The RHS of metadata state conditions can only contain static predicates describing the data and the LHS can only contain fluents and constraints. Recall that the LHS of goals could contain static predicates, which allowed us to express goals that relate the contents of one data file to the contents of another. Metadata formulas in the initial state can only relate data contents to the current or past state of the world. A

consequence of this restriction is that the predicates appearing on the LHS are completely disjoint from the predicates appearing on the RHS.

In addition to metadata, state conditions can also include unconditional fluent literals describing simple facts such as the names and locations of files.

```
STATE ::= state <IDENTIFIER> {
          ( forall PARAMS _; )
          | WHENEXP )+ }
```

8 Plans

A DPADL plan is a triple $< \mathcal{N}, \mathcal{A}, \mathcal{C} >$, where $\mathcal{N}$ and $\mathcal{A}$ are a set of nodes and arcs in the form of a directed acyclic graph (DAG). The nodes represent actions. The goal is represented as a node with only incoming arcs, and the initial state is represented as a node with only outgoing arcs. An arc $A \in \mathcal{A}$ is a tuple $< p, o_p, c, i_c >$, in which $p \in \mathcal{N}$ is the producer or source node, o_p is an output variable of p, $c \in \mathcal{N}$ is the consumer or target node and i_c is an input variable of c. We refer to the arcs in $\mathcal{A}$ as "I/O links," because they link the output of the producer to the input of the consumer. $\mathcal{C}$ is a constraint network, which is a triple $< V, D, C >$, where V is a set of variables appearing in actions $n \in \mathcal{N}$, D is a set of domains of those variables, representing their possible values, and C is a set of constraints, each of which defines a relation on some subset of the variables in V.

A plan is valid if

- All of the variables in V corresponding to action parameters are grounded (i.e., have singleton domains in D), $\mathcal{C}$ is solved. See (Golden & Frank 2002) for a discussion of how the constraint network is solved.
- All of the constraints corresponding to goals or preconditions are in C.
- Each input i_n of each action $n \in \mathcal{N}$ has a corresponding arc $< p, o_p, n, i_n >$, such that the constraint $i_n = o_p$ is in C and every precondition (excluding constraints) associated with i_n is supported by p. A disjunctive precondition is supported if one of the disjuncts is supported, a conjunctive precondition is supported if all of the conjuncts are supported, and a precondition of the form **"when** (Φ) $\{\psi\}$," where ψ is a literal and Φ is conjunctive, is supported by p if
  - It is a constraint in C **or**
  - $\Phi \models \psi$ **or**
  - There is a corresponding effect **"when** (Φ') $\{\psi'\}$" in p, such that $\psi' \models \psi$,[4] subject to the constraints in C, and either $\Phi \models \Phi'$ or the subgoal "**when**(Φ) $\{\Phi'\}$" is supported **or**
  - There is no effect **"when** (Φ') $\{\psi'\}$" in p, such that $\psi' \models \psi$ or $\psi' \models \neg\psi$, but there is an effect **"when** (Φ'') $\{o_p$ **copyof** $i_p\}$," where i_p is an input of p, and the subgoal **"when** (Φ) $\{\psi\{i_n/i_p\} \wedge \Phi''\}$" is supported.
- Each precondition not associated with any input is true in the initial state.

[4] Entailment can be determined using unification.

Since we are restricting our consideration to pure data-processing domains, we can ignore "sibling" subgoal clobbering. Actions only create new objects; they don't change the world or existing objects, so there is no opportunity for parallel branches to interact with each other. Note also that there is nothing preventing us from having multiple I/O links coming in to a single input, providing redundant ways of producing that input. At execution time, the agent will need to choose which to use, but deferring this choice to execution time can provide flexibility and robustness, since some data source may be unavailable, late, or of poor quality.

9 Conclusions and Related Work

We have described DPADL, an action language for data processing domains, which is used in the IMAGEbot system. The parser for the language, and a planner that supports the language, are fully implemented, and the whole system is fully integrated with the TOPS ecological forecasting system, which is under ongoing development; IMAGEbot can sense, plan and act in the TOPS domain.

We have compared DPADL to PDDL and discussed some of the reasons PDDL is not suitable for data processing domains. Of these, the most important are that it relies on the CWA, provides no support for inputs, outputs and object creation, and is very limited in the kinds of constraints that can be expressed. These problems could be addressed by less radical changes to the PDDL language. Some features, such as the use of **when** expressions in goals and the initial state and quantification over potentially infinite sets, are necessary for describing data processing on a causal level. We have found this low-level causal representation quite conducive to planning, since standard planning techniques can be used to correctly reason about the result of chaining multiple data-processing actions together. With a more abstract representation, paradoxically, more effort would be required of the domain designer .

DDL, the language used in the Europa planner (Jönsson *et al.* 2000), the descendent of the Remote Agent planner that flew on-board Deep Space One, supports constraints and rich temporal action models. In fact, the constraint reasoning system we use was taken from Europa. DDL supports a limited ability to create new objects, but not as a consequence of action execution. DDL domain descriptions are quite different from those of either PDDL or DPADL. Rather than describing actions in terms of preconditions and effects, DDL uses explanatory frame axioms. That is, for every condition that could be achieved, the domain designer must specify how to achieve it, listing all actions that could support it and other conditions that must be satisfied. DDL is also timeline-based and makes no distinction between states and actions. While these may be good design decisions for spacecraft domains, they are not appropriate for data processing domains.

DAML-S (Ankolenkar *et al.* 2002) and WSDL (Christensen *et al.* 2002) are languages for describing web services, both based on XML. DAML-S is the more expressive, allowing the specification of types using a description logic and allowing one to specify preconditions and post-conditions, which might be used by a planning agent. However, we don't believe that description logics are expressive

enough to describe the data-processing operations that we need to support.

The Earth Science Markup Language (ESML; http://esml.itsc.uah.edu) is another language based on XML, under development at the University of Alabama in Huntsville to provide metadata descriptions for Earth Science data. Unlike DAML-S and WSDL, ESML is well suited to describing the complex data structures that appear in scientific data. Unlike DPADL, it is only intended to describe data files, not data processing operations, but it does provide explicit support for describing the syntax and semantics of data files and allows the specification of constraints in the form of equations. Although it is less expressive and more specialized than DPADL, it is a promising metadata standard for Earth Science. In the near future, we hope to support conversion between ESML and DPADL metadata specifications.

Near the far end of the expressiveness spectrum, the situation calculus (McCarthy & Hayes 1969) provides plenty of expressive power, but at a price: planning requires first-order theorem proving. We opted instead to make our language as simple as possible, but no more so. DPADL does not support domain axioms, nondeterministic effects or uncertainty expressed in terms of possible worlds, and much of the apparent complexity of the language is handled by a compiler, which reduces complex expressions into primitives that a simple planner can cope with. Despite the superficial similarity to program synthesis (Stickel *et al.* 1994), DPADL action descriptions are not expressive enough to describe arbitrary program elements, and the plans themselves do not contain loops or conditionals.

Of the many planning domain description languages that have been devised, the closest to DPADL is ADLIM (Golden 2000), on which it is based. Advances over ADLIM include tight integration with the run-time environment (Java) and constraint system and a Java-like object-oriented syntax that makes it natural to describe objects and their properties. As discussed in Sections 2 and 6.3, this encodes valuable information used by the planner.

Collage (Lansky & Philpot 1993) and MVP (Chien *et al.* 1997) were planners that automated image manipulation tasks. However, they didn't focus as much on accurate causal models of data processing, so their representation requirements were simpler.

Acknowledgments

I am indebted to Wanlin Pang, Jeremy Frank, Ellen Spertus and Petr Votava for helpful comments and discussions. This work was funded by the NASA Computing, Information and Communication Technologies (CICT) Intelligent Systems program.

References

Ankolenkar, A.; Burnstein, M.; Hobbs, J. R.; Lassila, O.; Martin, D. L.; McDermott, D.; McIlraith, S. A.; Narayana, S.; Paolucci, M.; Payne, T. R.; and Sycara, K. 2002. DAML-S: Web service description for the semantic web. In *Proceedings of the 1st Int'l Semantic Web Conference (ISWC)*.

Chien, S.; Fisher, F.; Lo, E.; Mortensen, H.; and Greeley, R. 1997. Using artificial intelligence planning to automate science data analysis for large image database. In *Proc. 1997 Conference on Knowledge Discovery and Data Mining*.

Christensen, E.; Curbera, F.; Meredith, G.; and Weerawarana, S. 2002. Web Services Description Language (WSDL) 1.1. Technical report, World Wide Web Consortium. Available at http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

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

Golden, K., and Frank, J. 2002. Universal quantification in a constraint-based planner. In *Proc. 6th Intl. Conf. AI Planning Systems*.

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. 2000. Acting on information: a plan language for manipulating data. In *Proceedings of the 2nd NASA Intl. Planning and Scheduling workshop*, 28–33. Published as NASA Conference Proceedings NASA/CP-2000-209590.

Jönsson, A.; Morris, P.; Muscettola, N.; and Rajan, K. 2000. Planning in interplanetary space: Theory and practice. In *Proc. 5th Intl. Conf. AI Planning Systems*.

Lansky, A. L., and Philpot, A. G. 1993. AI-based planning for data analysis tasks. In *Proceedings of the Ninth IEEE Conference on Artificial Intelligence for Applications (CAIA-93)*.

McCarthy, J., and Hayes, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*. Edinburgh University Press. 463–502.

McDermott, D. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 2(2):35–55.

Nemani, R.; Votava, P.; Roads, J.; White, M.; Thornton, P.; and Coughlan, J. 2002. Terrestrial observation and prediction system: Integration of satellite and surface weather observations with ecosystem models. In *Proceedings of the 2002 International Geoscience and Remote Sensing Symposium (IGARSS)*.

Reiter, R. 1991. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V., ed., *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*. Academic Press. 359–380.

Stickel, M.; Waldinger, R.; Lowry, M.; Pressburger, T.; and Underwood, I. 1994. Deductive composition of astronomical software from subroutine libraries. In *Proceedings of the 12th Conference on Automated Deduction*.