603867

DPADL: An action language for data processing domains

Keith Golden
NASA Ames Research Center
kgolden@email.arc.nasa.gov

Abstract

This paper presents 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 will provide automation for an ecosystem forecasting system called TOPS.

1 Introduction

NASA is a data-collection agency. Terabytes of data are gathered each day by NASA telescopes, satellites and other spacecraft, and processing these data is truly challenging. Current approaches to data processing were devised when data volumes were small and it was possible for scientists to "look at" all the data. With the large data volumes now available, increasing levels of automation are needed if scientists are to avoid drowning in data. This is especially true in the Earth Science Enterprise, where high bandwidth communication makes it practical to return terabytes per day, and where the wide variety of instruments and the even wider variety of uses of the data make scripted approaches to automation too labor intensive.

We are working with Earth scientists to help address this challenge in the context of an ecosystem forecasting system called TOPS (http://www.forestry.umt.edu/ntsg/Projects/TOPS/). Our approach is to cast the problem of automating data-processing operations as a planning problem. We have developed a planner-based softbot (**soft**ware ro**bot**), called IMAGEbot, to generate and execute dataflow 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. Aspects of the planner are described in [2]. In this paper, we describe the IMAGEbot action language, which we call DPADL, for Data Processing Action Description Language. DPADL is a successor to the ADLIM language, described in [3]. A key requirement of the language is to provide precise causal models of data-processing actions, which can be used either for plan generation, or to derive metadata descriptions of the outputs of a given plan.

Given the richness and complexity of software systems and data-processing programs, the language used to describe them must be expressive. In particular, it must support

- First-class objects: Most things in the world and in software environments can be viewed as objects with certain attributes and relations to other objects. For example, a person may be described in terms of name, address, workplace., etc., while a file has a name, host location, owner, etc.

- Object creation and copying: Many programs create new objects, such as files, sometimes by copying or modifying other objects. The language should have a simple way of describing such operations.

- Operations on sets of objects: Many programs act on all members of some set. For example, `lpr *` prints all files in the current directory, and an image processing operation may affect all pixels in an image or a specified region of an image.

- Integration with a run-time environment: It is not sufficient to generate plans; it is necessary to execute them, so there must be a way to describe how to execute the operations provided by the environment and obtain information from the environment.

- Constraints: Determining the appropriate parameters for an action can be challenging. Parameter values can depend on other actions or objects in the plan, via complex constraints.

We have developed a language for describing objects and actions in a data-processing domain. The language provides tight integration with the Java run-time environment. In particular, Java code can be embedded in the action descriptions, planner variables can be bound to Java objects as well as primitive types, and constraints can be enforced by invoking methods on those objects.

In the remainder of the paper, we describe the language in detail. The elements of a domain description include specifications of types (Section 2), functions and relations (Section 3), goals (Section 5) and actions (Section 6). Definitions of types and functions can include constraints (Section 4).

At the end of each section, we present a BNF grammar covering the language elements described in that section. For example, we have stated so far that domain descriptions include types, function, goals and actions, and can include embedded Java code. The top-level production rule is:

```
DOMAIN  ::=  (TYPEDEF | FUNCTIONDEF | ACTIONDEF | GOALDEF
             | <INLINE_CODE>)+ <EOF>
```

where symbols in SMALL CAPS are non-terminals, symbols in <ANGLE_BRACKETS> are terminals, and keywords are shown in `typewriter` font.

2 Objects and primitive types

The language supports the definition of new types. The keyword for introducing a new type definition is **`type`**. For example

```
static type Filename isa string
```

introduces a new type, `Filename`, which is a subtype of `string`. The keyword **`static`** means that no instance of `Filename`, once created, can ever be changed. A type that is not static is **`fluent`**. Subtypes of `object` may represent Java objects. For example,

```
static type Tile isa object mapsto tops.modis.Tile
```

means that the type `Tile` corresponds to the Java class `tops.modis.Tile`.

Alternatively, we can define a type by listing of all possible instances of the type. This is similar to enumerated types in C/C++, but without the restriction to integral values. As in C/C++, enumerated values can have symbolic names attached to them.

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

```
type File isa object {
  key Path pathname;
  Directory parentDirectory;
  Filename filename;
  ...
}
```

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. When referring to the 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,

```
pic2.pixelValue(10,10) := pic1.pixelValue(0,0);
```

describes an effect that sets the value of the pixel at coordinates 10,10 in the image `pic2` to be equal to the value of the pixel 0,0 in the image `pic1`.

Object notation is not just syntactic sugar. The planner can reduce search by exploiting the fact that attributes of static objects don't change once the object is created, and Section 6.3.2 discusses the role attributes play when objects are copied.

```
      TypeDef ::= (static | fluent)? type ((<IDENTIFIER> =
                  { TypeMembers } )|(TypeSpec))(TypeBody | ; )
  TypeMembers ::= ((<IDENTIFIER> =)? Literal) (, TypeMembers)?
     TypeSpec ::= PrimitiveType | ( <IDENTIFIER> isa TypeName
                  ( implements TypeName )? ) ( mapsto <CLASSNAME> )?
     TypeBody ::= { ( MemberDef | ConstraintSpec )* }
    MemberDef ::= ( static | fluent )? key? TypeName <IDENTIFIER>
                  ( ParamList )? ( MemberBody | ; )
   MemberBody ::= { ( ConstraintSpec )* }
PrimitiveType ::= int | unsigned | real | string | object | boolean
     TypeName ::= <IDENTIFIER> | PrimitiveType
```

3 Functions and relations

Object attributes may be viewed as a functions or relations, where one of the arguments is singled out as special, namely, the object itself. We can also have functions and relations that are not object attributes. For example,

```
fluent real elevation(real lon, real lat);
```

defines a function that takes two real values, representing longitude and latitude, and returns a real value representing elevation. There is no special syntax for defining relations; they are just functions whose value is type boolean. 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. A fluent with no arguments may be considered a variable, and a static function with no arguments is essentially a constant.

```
FUNCTIONDEF ::= (static | fluent) TYPENAME ( <IDENTIFIER>
                | <OPERATOR> ) ( ( PARAMS? ) )? ( ; | { ( CONSTRSPEC )* } )
     PARAMS ::= ( PARAMDEF ( , PARAMS )? ) | :rest PARAMDEF
   PARAMDEF ::= TYPENAME <IDENTIFIER>
```

4 Constraints

IMAGEbot uses a constraint-based planner to reason about the often complex dependencies and interactions among actions and objects in the plan. Constraints associated with types, attributes and functions can be selected from a library of constraints or defined in terms of arbitrary Java code embedded in the type and function definitions. 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 [2].

4.1 Constraints on types

We can define unary constraints on static types, effectively restricting membership of the type to a proper subset of the parent type. For example,

```
static type Filename isa 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 limitation on filenames. `Matches` is a constraint from the constraint library requiring a string to a match a regular expression. Constraints can also be defined in terms of arbitrary Java code, as discussed below.

4.2 Constraints on attributes

We can define constraints on attributes as well as types. For example,

```
static type Tile isa object mapsto tops.modis.Tile {
  key string uniqueId {
    constraint {
      value(this) := $ this.getUID() $;
      this(value) := $ Tile.findTile(value) $;
    }
  ...
```

means that the `uniqueId` attribute of a (mosaic) `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 "`:=`" specifies 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`, and the keyword **`this`** refers to an object of the type being defined, in this case `Tile`, so **`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. 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.

4.3 Constraints on functions

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 referenced using a syntax similar to that used for C++ operator overloading. For example to specify that the "+" operator applied to strings concatenates them, we can write

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

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

```
CONSTRSPEC   ::=  constraint ( <IDENTIFIER> ARGLIST ; )
                  | { ( CONSTRAINT )+ } )
 CONSTRAINT  ::=  ( CONSTRARG CONSTRARGS := <INLINE_CODE> ; )
                  | ( [ CONSTRARG ] CONSTRARGS :=  [ <INLINE_CODE> ,
                  <INLINE_CODE> ]  ; )
                  | ( { CONSTARG } CONSTRARGS :=  { <INLINE_CODE> }  ; )
 CONSTRARG   ::=  <IDENTIFIER> | value | this
CONSTRARG2   ::=  ( CONSTRARG | [ CONSTRARG ] | { CONSTRARG } )
                  ( = ADDITIVEEXPRESSION )?
CONSTRARGS   ::=  ( CONSTRARG2 [ , CONSTRARGS ] )
```

5 Goals

Goals are used to describe data products that the system should produce. Data product descriptions should specify the following four attributes:

- Semantics: the information represented by the data. That is, what facts about the world that can be inferred from the data contents.
- 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.
- Location: where the data file should be put or delivered.

Time is an optional argument of all fluents, and specifying location requires no special syntax. The mapping between semantics and syntax is specified using the keyword **when**. For example, to specify that a file represents the temperature over a particular region, using the LAZEA projection and the function `tempEncoding` to map from temperatures to pixel values, we could write:

```
when(tempEncoding(temperature(long,lat)) = t && proj(LAZEA,x,y,long,lat)
     && 0 <= x < MAXX && 0 <= y < MAXY) {
  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). There are significant differences between the LHS and the RHS of a goal. The LHS implicitly refers to the initial state (unless another time is specified) and properties of the world, and the RHS refers to the final state (whenever the goal achieved) and properties of the data.

```
       GoalDef ::= goal <IDENTIFIER> ( Params? )    { ( (output | forall
                   | exists) Params )* OrExp }
         OrExp ::= CondExp ( || CondExp )*
       CondExp ::= (when ( AndExp ) { CondExp } ( else {
                   CondExp } )? ) | AndExp
        AndExp ::= EqualExp ( && ( EqualExp ) )*
      EqualExp ::= RelationExp ( ( = | != ) RelationExp )*
   RelationExp ::= AdditiveExp ( ( < | > | <= | >= ) AdditiveExp )*
   AdditiveExp ::= MultiplExp ( ( + | - ) MultiplExp )*
    MultiplExp ::= UnaryExp ( ( * | / | % ) UnaryExp )*
      UnaryExp ::= ( + | - | ! )? PrimaryExp
    PrimaryExp ::= ( AndExp ) | ( Function | this ) ( . Function )* | Literal
      Function ::= <IDENTIFIER> ( ( Args ) )?
       Literal ::= <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL>
                   | <CHARACTER_LITERAL> | <STRING_LITERAL> | <NULL>
                   | true | false
          Args ::= AdditiveExpression (, Args )?
```

6 Actions

Actions can be sensors (which produce outputs based on the state of the world) or filters (which produce outputs 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 is part of the action description.

```
ActionDef ::= action <IDENTIFIER> ( Params ) { (( input | output
              | forall) Params) | Precond | Effect | Exec )* }
```

6.1 Inputs, outputs and parameters

As in PDDL[6], actions are parameterized, and parameters are typed. In addition to ordinary parameters, two kinds of parameters 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.

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 6.3.2.

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 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 ecosystem 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, and to reference states other than the prior state. In other words, preconditions, like goals, can include metadata descriptions, which are described using the keyword **when**.

```
PRECOND  ::=  precond ( OREXP ; )+
```

6.3 Effects

Effects are used primarily 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 )+
```

6.3.1 Conditional effects

Conditional effects are the same as in PDDL. As with goals, they 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).

Stripping away the syntactic sugar, every 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. We depart from the C++/Java syntax and use the notation ":=" to denote assignment and "=" to denote equality.

```
      WHENEXP  ::=  (when ( ANDEXP )  { ( WHENEXP )* } ( else {
                    ( WHENEXP )* } )? ) | CONSEQUENT
   CONSEQUENT  ::=  ( ASSIGNMENTEXP | NEWDECL )
ASSIGNMENTEXP  ::=  CFUNCTION ( . CFUNCTION )* ( := ( EQUALITYEXP | NEWEXP ) )? ;
    CFUNCTION  ::=  <IDENTIFIER> ( ( CARGS ) )?
        CARGS  ::=  ( ADDITIVEEXP | NEWEXP ) (, CARGS )?
```

6.3.2 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(input.contains(orig)) {
  output.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. In contrast, in ADLIM, all relations involving the original assumed to hold for the copy unless explicitly overridden, so it would be necessary to declare that `in.contains(copy)` is false after copying `orig` to `copy`.

The copy and the original need not be the same type, as long as one is a subtype of the other or one **implements** the other. All attributes common to both types are copied.

NEWDECL	::=	**new** TYPENAME <IDENTIFIER> (**copyof** <IDENTIFIER>)? ((**{** (ATTRIBUTES)* **}**) | **;**)
NEWEXP	::=	**new** TYPENAME (**copyof** <IDENTIFIER>)? ((**{** ATTRIBUTES * **}**) | **;**)
ATTRIBUTES	::=	FUNCTION **:=** (EQUALITYEXP **;** | NEWEXP)

6.4 Execution

The procedure for executing an action is specified using inlined Java code.

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

7 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 the planner that supports the language, are fully implemented, and we are developing DPADL descriptions for the TOPS ecosystem forecasting system. Currently, a subset of TOPS, dealing with MODIS satellite data, is supported.

It would be great to use the now-standard PDDL[6], the language devised for the AIPS programming competitions, rather than write a new language. Unfortunately, PDDL is not adequate for describing data processing domains. It provides no support for object creation and copying, no explicit inputs or outputs, cannot describe information content of data, and does not support integration with a run-time environment or constraint reasoning system. There are also syntactic disadvantages, such as lack of functions and object-oriented notation, which could be worked around but are obstacles to clean domain descriptions. We opted instead to base our language on the well-known syntax of C++ and Java, since many programmers are familiar with that syntax, so the learning curve for the language should be reduced. An additional motivation was that action descriptions and program code both describe the same things – state change, conditional on the current state, so using similar syntax for both is appealing.

Alternatively, we could have used the situation calculus [5], which provides plenty of expressive power, but at a price. 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 the planner can cope with. Despite the superficial similarity to program synthesis [7], 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 languages that have been devised, the closest to DPADL is ADLIM[3], 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 6.3.2, this is not just syntactic sugar, but encodes valuable information used by the planner.

Collage [4] and MVP [1] 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.

References

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

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

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

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

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

[6] D. McDermott. The 1998 AI Planning Systems Competition.

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