FDDL — Finite Domain Definition Language

Adapted from the PDDL manual produced by the AIPS-98 Planning Competition Committee

Michael P Fourman

Institute for Communicating and Collaborative Systems
Institute for Intelligent Systems and their Applications
Laboratory for Foundations of Computer Science
Informatics
The University of Edinburgh

Abstract

This note describes the syntax and semantics of the Domain Definition Language, FDDL, a specification language for finite domains. The language provides rudimentary facilities for specifying structures with a fixed finite domain, equipped with a collection of subsets (known as types), a collection of relations, and some uninterpreted predicates.

In addition, FDDL provides a syntax for typed first-order logic (without function symbols).

A domain specification fixes the underlying structure of elements, types, and relations. Our aim is to find and count the ways in which the predicates may then be interpreted over this structure to satisfy given axioms.

PropSat is a BDD-based package for counting and finding models for FDDLdomains.

1 Introduction

This note describes the syntax and semantics of the Finite Domain Definition Language, (FDDL). The language provides rudimentary facilities for specifying finite structures, and a syntax for typed first-order logic (without function symbols).

2 Domains

A domain, is a finite set, $\mathcal{D}$, of *constants*, equipped with a set, $\mathcal{T} \subseteq \mathcal{P}(\mathcal{D})$, of *types* (fixed subsets of the domain), a number of fixed *relations* (of various typed arities), and a *signature*, $\mathcal{S}$, specifying a number of *predicates*, and their typed arities. (Although types correspond semantically to unary relations, in the current implementation, types are not syntactically available as unary relations-they can only be used to restrict the range of a quantified variable.)

The idea is first to specify a particular finite structure, then to consider how we may add further predicates satisfying particular axioms. To pin down the interpretations of the fixed relations, we may specify a number of *facts*—which are interpreted with a closed world semantics (and are therefore restricted to be universal Horn sentences).

Once we have specified a domain, we can ask whether the predicates have interpretations making a given formula in first-order predicate calculus, true. We can count the number of interpretations making a given set of *axiom*s, true, and exhibit one such, if any there be.

So the components of a problem specification are:

types	Specified by enumerating elements and by stipulating type inclusions. Elements may have several types.
constants	Specified by enumeration.
relations	Whose fixed interpretation is determined, using a closed world assumption, by the facts.
facts	Universal Horn sentences with closed-world semantics.
predicates	Whose interpretation is restricted by the facts and axioms.
axioms	Arbitrary first-order formulae, interpreted over the finite domain.

3 A Simple Example

To give a flavour of the language, consider the following problem. There are eight teams in a tournament. Each team must play exactly three others. How many ways are there to arrange such a tournament? Find one.

```
(define (domain tournament)
  (:types team)
  (:constants t0 t1 t2 t3 t4 t5 t6 t7 - team)
  (:predicates (plays ?x ?y - team))
  (:axioms
    (forall (?x - team) (not (plays ?x ?x)))                ;irreflexive
```

```
    (forall (?x ?y - team) (iff (plays ?x ?y) (plays ?y ?x))) ;symmetric
    (forall (?x - team) (= 3 (?y - team) (plays ?x ?y)))
   )
  )
```

This domain has one type, with eight elements. We search for graphs with this set of nodes, such that each node has degree three — binary relations which are irreflexive, symmetric, and have the required property.

Of the 268,435,456 irreflexive symmetric relations on eight elements, 19,355 represent valid ways to order the tournament. Here is one of them:

```
["(plays t7 t6)", "(plays t6 t7)", "(plays t7 t5)", "(plays t5 t7)",
 "(plays t6 t5)", "(plays t5 t6)", "(plays t7 t4)", "(plays t4 t7)",
 "(plays t6 t4)", "(plays t4 t6)", "(plays t5 t4)", "(plays t4 t5)",
 "(plays t3 t2)", "(plays t2 t3)", "(plays t3 t1)", "(plays t1 t3)",
 "(plays t2 t1)", "(plays t1 t2)", "(plays t3 t0)", "(plays t0 t3)",
 "(plays t2 t0)", "(plays t0 t2)", "(plays t1 t0)", "(plays t0 t1)"]
```

Suppose we now elaborate the example. There are three junior and five senior teams. We want to ensure that each junior team plays at least one junior team, and each senior team at least one senior team.

```
(define (domain tournament)
  (:types junior senior - team)
  (:constants t0 t1 t2 - junior t3 t4 t5 t6 t7 - senior)
  (:predicates (plays ?x ?y - team))
  (:axioms
    (forall (?x - team) (not (plays ?x ?x)))
    (forall (?x ?y - team) (iff (plays ?x ?y) (plays ?y ?x)))
    (forall (?x - junior)(exists (?y - junior) (plays ?x ?y)))
    (forall (?x - senior)(exists (?y - senior) (plays ?x ?y)))
    (forall (?x - team)(= 3 (?y - team) (plays ?x ?y)))
   )
  )
```

The `types` declaration introduces junior and senior as subtypes of team. The `constants` declaration introduces the eight teams, within their respective subtypes.

All domains include the built-in type, `object`, interpreted as the underlying set of the domain, and the equality relation, `=`. Most domains, like our examples, define further types, interpreted as subsets, and further predicates. FDDL provides syntax for specifying type inclusions.

Elements of the domain are specified by constants (whose interpretations are assumed to be distinct).

The two features of the language not exhibited by the examples so far are facts and relations. These will be demonstrated in due course.

4 Syntactic Notation

Our notation is an Extended BNF (EBNF) with the following conventions:

- Each rule is of the form <*syntactic element*> ::= *expansion*.
- Angle brackets delimit names of syntactic elements.
- Square brackets ([and]) surround optional material.
- An asterisk (*) means "zero or more of"; a plus (+) means "one or more of."
- Some syntactic elements are parameterized. E.g., <list (symbol)> might denote a list of symbols, where there is an EBNF definition for <list x> and a definition for <symbol>. The former might look like

  <list x> ::= (x*)

  so that a list of symbols is just (<symbol>*).
- Ordinary parentheses are an essential part of the syntax we are defining and have no semantics in the EBNF meta language.

Comments in FDDL begin with a semicolon (";") and end with the next newline. Any such string behaves like a single space.

5 Syntax

We now describe the language more formally. The EBNF for defining a domain structure is:

```
<domain>            ::= (define (domain <name>)
                                [<types-def>]
                                [<constants-def>]
                                [<relations-def>]
                                [<predicates-def>]
                                [<facts-def>]
                                [<axioms-def>])
<types-def>         ::= (:types <typed list (name)>)
<constants-def>     ::= (:constants <typed list (name)>)
<relations-def>     ::= (:relations <atomic formula skeleton>+)
<predicates-def>    ::= (:predicates <atomic formula skeleton>+)
<facts-def>         ::= (:facts <positive formula>+)
<axioms-def>        ::= (:axioms <formula>+)
<atomic formula skeleton>
                    ::= (<predicate> <typed list (variable)>)
<predicate>         ::= <name>
<variable>          ::= ?<name>
```

The optional components of a domain may come in any order. Just one FDDL definition of a domain may appear per file.

Names of domains, like other occurrences of syntactic category `<name>`, are strings of characters beginning with a letter and containing letters, digits, hyphens ("-"),and underscores ("_"). Case is not significant.

The `:types` argument uses a syntax borrowed from McDermott's NISP:

```
<typed(x)> ::= x+- <type>
<typed(x)> ::= x*
<typed list (x)> ::= <typed (x)>*
<type>           ::= <name>
<type>           ::= (either <type>+)
```

A typed list is used to declare the types of a list of entities; the types are preceded by a minus sign ("-"), and every other element of the list is declared to be of the first type that follows it, or `object` if there are no types that follow it. An example of a `<typed list(name)>` is

```
integer float - number physob
```

If this occurs as a `:types` argument to a domain, it declares four new types, `integer`, `number`, `float`, and `physob`. The first two are subclasses of `number`, the last a subclass of `object` (by default). That is, every integer is a number, every float is a number, and every physical object is an object.

In addition to atomic type names, there are union types `(either` t_1 ... t_k`)` is the union of types t_1 to t_k.

The `:constants` field has the same syntax as the `:types` field, but the semantics is different. Now the names are taken as new constants in this domain, whose types are given as described above. E.g., the declaration

```
(:constants sahara - theater
            division1 division2 - division)
```

indicates that in this domain there are three distinguished constants, `sahara` denoting a `theater` and two symbols denoting `division`s. An object may be declared repeatedly to belong to any number of types; the types are just subsets of the finite set of all declared objects.

The `:predicates` field consists of a list of declarations of predicates, once again using the typed-list syntax to declare the arguments of each one.

The syntax `<literal(name)>` will be defined in Section 6.

6 Formulae

FDDL axioms are quite expressive. All of function-free first-order predicate logic (including nested quantifiers) is allowed. Numeric quantifiers are provided.

FDDL facts are (much) more restrictive: only universal horn formulae are allowed. Since we are dealing with only finite domains these can be syntactically slightly more relaxed than usual. Positive formulae are built from relation atoms using `and` and `forall`. Horn formulae are built from positive formulae using universal quantification and, at most, a single implication.

```
<formula>            ::= <atomic formula(term)>
<formula>            ::= <literal(term)>
<formula>            ::= (and <formula>*)
<formula>            ::= (or <formula>*)
<formula>            ::= (not <formula>)
<formula>            ::= (imply <formula> <formula>)
<formula>            ::= (iff <formula> <formula>)
<formula>            ::= (exists (<typed list(variable)>) <formula> )
<formula>            ::= (forall (<typed list(variable)>) <formula> )
<formula>            ::= (< <num> (<typed list(variable)>) <formula> )
<formula>            ::= (= <num> (<typed list(variable)>) <formula> )
<formula>            ::= (> <num> (<typed list(variable)>) <formula> )
<formula>            ::= (=< <num> (<typed list(variable)>) <formula> )
<formula>            ::= (>= <num> (<typed list(variable)>) <formula> )
<horn formula>       ::= <positive formula>
<horn formula>       ::= (imply <positive formula> <positive formula>)
<horn formula>       ::=
                         (forall (<typed list(variable)>) <horn formula> )
<positive formula>   ::= <atomic formula(term)>
<positive formula>   ::= (and <positive formula>*)
<positive formula>   ::=
                         (forall (<typed list(variable)>) <positive formula> )
<literal(t)>         ::= <atomic formula(t)>
<literal(t)>         ::= (not <atomic formula(t)>)
<atomic formula(t)>  ::= (<predicate> t*)
<term>               ::= <name>
<term>               ::= <variable>
```

where the category `<num>` represents natural numbers, represented in decimal notation.

Warning: An occurrence of a `<predicate>` should agree with its declaration in terms of number and, when applicable, types of arguments. In the current implementation, declarations of predicates are permitted, but neither required nor used to check that predicates are applied consistently. Declarations of relations are used — they are used in facts and a closed-world assumption is applied to them. However, the types and number of their arguments are not checked.

7 Semantics

The semantics of most of these expresssions is standard. The numeric quantifiers are interpreted as expressing that there are particular numbers of instantiations of the (tuple of)

quantified variables making the body true. Thus, for example, `(exists (?x) <phi>)` is equivalent to `(>= 1 (?x) <phi>)`.

Facts have a closed-world semantics. This means that relations are given a minimally true interpretation. A particular instantiation of a relation is true iff it is implied by the facts. Since the facts are universal Horn sentences, this gives a structure consistent with the facts — ie the facts are true in this interpretation.

8 Checking Propositional Satisfiability

PropSat is a tool written in ML to check the satisfiability of FDDL specifications. It translates specifications to propositional form and uses David Long's LIBBDD library to check satisfiability.

PropSat onLine

You can find further example domains and problems, and run PropSat online at

`http://homepages.inf.ec.ac.uk/mfourman/propsat/`

PropSat on DICE

`propsat` can be invoked (on DICE machines within informatics@edinburgh) by running the script at`/home/mfourman/bin/propsat`.

This puts you into an interactive ML session where propsat is available.

Usage: at the ml prompt `<` type

`propsat "`⟨*domain-file*⟩`";`

To exit the ML session use control-C to interrupt the ML interpreter, and then type `q` (or `?` for other options).