The IDP framework reference manual

KU Leuven Knowledge Representation and Reasoning research group

October 11, 2012

Contents

TODO: everywhere in the text, replace dollar signs and { `tt ...` } by `code` { ... }

1 Installing And Running

The system has been verified to run under Windows and various Unix versions. The system also works under OSX, but for any version at or below Lion it requires developer components on the user machine to run.

1.1 Getting the system

1.1.1 Downloading the most recent version

Pre-built binaries can be retrieved from `http://dtai.cs.kuleuven.be/krr/software/idp` and installed in their default (OS-specific) way. For the reasons stated above, we do not provide pre-built OSX packages.

1.1.2 Building from source

Required software packages:

- C and C++ compiler, supporting most of the C++11 standard. Examples are GCC 4.4, Clang 3.1 and Visual Studio 11 or higher.
- Cmake build environment.
- Bison and Flex parser generator software.
- Pdflatex for building the documentation.
- (optional) Gecode for constraint programming support.

Assume idp is unpacked in `idpdir`, you want to build in `builddir` (cannot be the same as `idpdir`) and install in `installdir`. Building and installing is then achieved by executing the following commands:

```
cd <builddir>
cmake <idpdir> -DCMAKE_INSTALL_PREFIX=<installdir>
               -DCMAKE_BUILD_TYPE="Release"
make -j 4
make check
make install
```

If you have Gecode installed and want to use it for constraint programming support, you add the option `-DWITHGECODE=ON`.

Alternatively, cmake-gui can be used as a graphical way to set cmake options.

1.2 Running the software

1.2.1 Batch mode

One-shot execution of a procedure `proc` with a set of files `files` is achieved by running

```
runidp -e "proc()" files
```

Omitting the `-e` option results in execution of the `main` method if any is available.

1.2.2 Interactive mode

An interactive session, with (optionally) a set of files `files`, is started with

```
runidp files -i
```

Afterwards, help can be requested with the `help()` command. Auto-completion of available commands is available via the tab key. Additional files can be included with the `parse` command.

2 Comments

Everything between `/*` and `*/` is a comment, as well as everything between `//` and the end of the line. If a comment block starts with `/**`, but not with `/***`, then the comment is added as a description to the first thing after that comment block that can have a description. Currently, only procedures can have a description.

3 Include statements

Everywhere in an IDP file, a statement

```
#include "path/to/file"
```

is replaced by the contents of the file `path/to/file`. A statement

```
#include <filename>
```

is replaced by the contents of the standard library file `filename`. Currently the following standard library files are available:

mx Contains some useful model expansion procedures.

table_utils Contains tools for manipulating tables and converting IDP tables to lua tables.

4 Namespaces

A namespace with name `MySpace` is declared by

```
namespace MySpace {
   // content of the namespace
}
```

A Namespace can contain namespaces, vocabularies, theories, structures, terms, queries, procedures, options and using statements.

An object with name `MyName` declared in namespace `MySpace` can be referred to by absolute qualification: `MySpace::MyName`. Inside `MySpace`, `MyName` can simply be referred to by `MyName`. Additionally, *using* statements can be used to allow relative qualification. A using statement is of one of the following forms

```
using namespace MySpace
using vocabulary MyVoc
```

where `MySpace` is the name of a namespace, and `MyVoc` the name of a vocabulary. Below such a using statement, objects `MyObj` declared in `MySpace`, respectively `MyVoc`, can be referred to by `MyObj`, instead of `MySpace::Myobj`, respectively `MyVoc::MyObj`.

Every object that is declared outside a namespace, is considered to be part of the global namespace, called `idpglobal`. In other words, every IDP file implicitely starts with `namespace idpglobal`{ and ends with an additional }. Everything declared inside a library is contained within the namespace `stdspace`. It is discouraged to add or overwrite objects within stdspace.

TODO: define object

5 Vocabularies

A vocabulary with name `MyVoc` is declared by

```
vocabulary MyVoc {
   // contents of the vocabulary
}
```

A vocabulary can contain symbol declarations, symbol pointers, and other vocabularies. Symbols are types (sorts), predicate and functions symbols.

5.1 Symbol declarations

A type with name `MyType` is declared by

```
type MyType
```

When declaring a type, it can be stated that this type is a subtype or supertype of a set of other types. The following declares `MyType` to be a subtype of the previously declared types `A1` and `A2`, and a supertype of the previously declared types `B1` and B2:

```
type MyType isa A1, A2 contains B1, B2
```

In the rest of this text, we will sometimes use "parent type" as direct supertype and "ancestor type" for (in)direct supertype.

A predicate with name `MyPred`, arity 3 and types `T1`,`T2`,`T3` is declared by

```
MyPred(T1,T2,T3)
```

A predicate with arity zero can be declared by `MyPred()` or `MyPred`.

A function with name `MyFunc`, input types `T1`,`T2`,`T3` and output type `T` is declared by

```
MyFunc(T1,T2,T3):T
```

A partial function is declared by

```
partial MyFunc(T1,T2,T3):T
```

Constants of type `T` can be declared by `MyConst:T` or `MyConst():T`. Besides functions with an identifier as name, functions of arity two with names `+`,`-`,`*`,`/`,`%` and `^` can be declared, as well as unary functions with names `-` and `abs`.

Any symbol has to be declared on its own line.

5.2 Symbol pointers

To include a type, predicate, or function from a previously declared vocabulary `V` in another vocabulary `W`, write

```
/* Declaration of vocabulary V*/
vocabulary V {
   //...
   type A
   P(A)
   F(A,A):A
   //...
}

vocabulary W {
   extern type V::A
   extern V::P[A]     //also possible: extern V::P/1
   extern V::F[A,A:A]    //also possible: extern V::F/2:1
}
```

TODO: verwijder using en extern, zie meeting. Het wordt extends, right?

In the example, explicitly including type `A` of vocabulary `V` in `W` is not needed, since types of included predicates or functions are automatically included themselves. To include the whole vocabulary `V` in `W` at once, used

```
vocabulary W {
   extern vocabulary V
}
```

5.3 The standard vocabulary

The global namespace contains a fixed vocabulary `std`, which is defined as follows:

```
vocabulary std {
   type nat
   type int contains nat
   type float contains int
   type char
   type string contains char
```

```
    +(int,int) : int
    -(int,int) : int
    *(int,int) : int
    /(int,int) : float
    %(int,int) : int
    abs(int) : int
    -(int) : int

    +(float,float) : float
    -(float,float) : float
    *(float,float) : float
    /(float,float) : float
    ^(float,float) : float
    abs(float) : float
    -(float) : float
}
```

Every vocabulary implicitly contains all symbols of `std`. Also, every vocabulary contains for each of its types `A` the predicates `=(A,A)`, `<(A,A)`, and `>(A,A)` and the functions `MIN:A`, `MAX:A`, `SUCC(A):A` and `PRED(A):A`. In every structure, the symbols of `std` have the following interpretation:

`nat`	all natural numbers
`int`	all integer numbers
`float`	all floating point numbers
`char`	all characters
`string`	all strings
`+(int,int) : int`	integer addition
`-(int,int) : int`	integer subtraction
`*(int,int) : int`	integer multiplication
`/(int,int) : float`	division
`%(int,int) : int`	remainder
`abs(int) : int`	absolute value
`-(int) : int`	unary minus
`+(float,float) : float`	floating point addition
`-(float,float) : float`	floating point subtraction
`*(float,float) : float`	floating point multiplication
`/(float,float) : float`	floating point division
`^(float,float) : float`	floating point exponentiation
`abs(float) : float`	absolute value
`-(float) : float`	unary minus

The predicate `=/2` is always interpreted by equality. The order $<_{dom}$ on domain elements is defined by

- numbers are smaller than non-numbers;
- $d_1 <_{dom} d_2$ if d_1 and d_2 are numbers and $d_1 < d_2$;

- $d_1 <_{dom} d_2$ if d_1 and d_2 are strings that are not numbers and d_1 is before d_2 in the lexicographic ordering;
- $d_1 <_{dom} d_2$ is some total order on compound domain elements (which we do not specify).

Every structure contains the following fixed interpretations:

`<(A,A)`	the projection of $<_{dom}$ to the domain of A
`>(A,A)`	the projection of $>_{dom}$ to the domain of A
`MIN:A`	the $<_{dom}$-least element in the domain of A
`MAX:A`	the $<_{dom}$-greatest element in the domain of A
`SUCC(A):A`	the partial function that maps an element a of the domain of `A` to the $<_{dom}$-least element of the domain of A that is strictly larger than a
`PRED(A):A`	the partial function that maps an element a of the domain of `A` to the $<_{dom}$-greatest element of the domain of A that is strictly smaller than a

In an IDP-file, you should disambiguate which `MAX` you want to use. This is done by `MAX[:MyType]`.

6 Theories

A theory with name `MyTheory` over a vocabulary `MyVoc` is declared by

```
theory MyTheory : MyVoc {
   // contents of the theory
}
```

A theory contains sentences and inductive definitions.

6.1 Sentences

6.1.1 Terms

Before explaining the syntax for sentences, we need to introduce the concept of a term and a formula. We also give the syntax for terms and formulas in IDP.

A *term* is inductively defined as follows:

- a variable is a term;
- a constant is a term;
- if F is a function symbol with n input arguments and $t_1, \ldots, t_n$ are terms, then $F(t_1, \ldots, t_n)$ is a term.

In IDP, variables start with a letter and may contain letters, digits and underscores. When writing a term in IDP, the constant and function symbols occurring in that term should be declared before. The *type of a term* is defined as its return type (see section 5.1) in the case of constants and functions. The type of a variable is derived from its occurrences in formulas (see section 6.5). If a term occurs in an input position of a function, then the type of the term and the type of the input position must have a common ancestor type.

6.1.2 Formulas and Sentences

A *formula* is inductively defined by:

- **true** and **false** are formulas;
- if P is a predicate symbol with arity n and $t_1, \ldots, t_n$ are terms, then $P(t_1, \ldots, t_n)$ is a formula;
- if t_1 and t_2 are terms, then $t_1 = t_2$ is a formula;
- if φ and ψ are formulas and x is a variable, then the following are formulas: $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, $\varphi \Leftarrow \psi$, $\varphi \Leftrightarrow \psi$, $\forall x\ \varphi$, and $\exists x\ \varphi$.

The following order of binding is used: $\neg$ binds tightest, next $\wedge$ and $\vee$, then $\Rightarrow$ and $\Leftrightarrow$, and finally $\forall$ and $\exists$. Desambiguation can be done using brackets '(' and ')'. E.g. the formula $\forall x\ P(x) \wedge \neg Q(x) \Rightarrow R(x)$ is equivalent to the formula $\forall x\ ((P(x) \wedge (\neg Q(x))) \Rightarrow R(x))$.

As for terms, if term t occurs in predicate P, then the type of t and the type of the input position of P where it occurs must have a common ancestor type. For formulas of the form $t_1 = t_2$, t_1 and t_2 must have a common ancestor type.

The *scope* of a quantification $\forall x$ or $\exists x$, is the quantified formula. E.g., in $\forall x\ \psi$, the scope of $\forall x$ is the formula ψ. An occurrence of a variable x that is not inside the scope of a quantification $\forall x$ or $\exists x$ is called *free*. A *sentence* is a formula containing no free occurrences of variables. If an IDP problem specification contains formulas that are not sentences, the system will implicitly quantify this variable universally and return a warning message, specifying which variables occur free. Each sentence in IDP ends with a dot '.'.

The IDP syntax of the different symbols in formulas are given in the table below. Also the informal meaning of the symbols is given.

Logic	IDP	Declarative reading
$\wedge$	`&`	and
$\vee$	`|`	or
$\neg$	`~`	not
$\Rightarrow$	`=>`	implies
$\Leftarrow$	`<=`	is implied by
$\Leftrightarrow$	`<=>`	is equivalent to
$\forall$	`!`	for each
$\exists$	`?`	there exists
$=$	`=`	equals
$\neq$	`~=`	does not equal

Besides this, for every natural number n, IDP also supports the following quantifiers (with their respective meanings):

IDP	Declarative reading
`?n`	there exist exactly n different elements such that
`?<n`	there exist less than n
`?=<n`	there exist at most n
`?=n`	there exist exactly n (this is the same as `?n`)
`?>n`	there exist more than n

A universally quantified formula $\forall x\ P(x)$ becomes '`! x : P(x)`' in IDP syntax, and similarly for existentially quantified formulas. As a shorthand for the formula '`! x : ! y : ! z : Q(x,y,z).`', one can write '`! x y z : Q(x,y,z)`'.

In IDP, every variable has a type. The informal meaning of a sentence of the form $\forall x\ \psi$, respectively $\exists x\ \psi$, where x has type T is then 'for each object x of type T, ψ must be true', respectively 'there exists at least one object x of type T such that ψ is true'. The type of a variable can be declared by the user, or derived by IDP (see section 6.5).

6.1.3 Definitions

A definition defines a concept, i.e. a predicate (or multiple predicates), in terms of other predicates. Formally, a definition is a set of rules of the form

$$\forall x_1, \ldots, x_n\ P(t_1, \ldots, t_m) \leftarrow \varphi$$

where P is a predicate symbol, $t_1, \ldots, t_m$ are terms that may contain the variables $x_1, \ldots, x_n$ and φ a formula that may contain these variables. $P(t_1, \ldots, t_m)$ is called the *head* of the rule and ψ the *body*.

A definition in IDP syntax consists of a set of rules, enclosed by '{' and '}'. Each rule ends with a '`.`'. The definitional implication $\leftarrow$ is written '`<-`'. The quantifications before the head may be omitted in IDP (IDP will give a warning in this case), i.e., all free variables of a rule are implicitly universally quantified. If the body of a rule is empty (`true`), the rule symbol '`<-`' can be omitted. Recursive definitions are allowed in IDP. The semantics for a definitions are the well-founded semantics TODO: reference. As an example, the following definition defines the transitive closure of a relation `R`.

```
{
    !x y: T(x,y) <- R(x,y).
    !x y: T(x,y) <- ?z: T(x,z) & T(z,y).
}
```

6.2 Chains of (in)equalities

As in mathematics, one can write chains of (in)equalities in IDP. They can be used as shorthands for conjunctions of (in)equalities. E.g.:

```
! x y : (1 =< x < y =< 5) => ...
// is a shorthand for
! x y : ((1 =< x) & (x < y) & (y =< 5)) => ...
```

6.3 Aggregates

Aggregates are functions that take a set as argument, instead of a simple variable. IDP supports some aggregates that map a set to an integer. As such, they can be seen as integer terms.

Syntacticly, there are two kinds of sets in IDP.

- An expression of the form '`[ (phi_1,t_1) ; (phi_2,t_2) ; ... ; (phi_n,t_n)]`', where each `phi_i` is a formula and each `t_i` is a term.

- An expression of the form '{ x_1 x_2 ... x_n : phi : t}', where the x_i are variables, phi is a formula and t is a term. The variables x_i can occur in both phi and t.

The informal interpretation of sets of the different kinds is (respectively):

- The multiset of all t_i for which phi_i is true (multiple occurences of the same term are possible).
- The multiset consisting of: for every tuple of domain elements (a_1,a_2, ...,a_n), the term t[a_i/x_i] if phi[a_i/x_i] is true.

The current system has support for five aggregate functions:

Cardinality: The cardinality of a set is the number of elements in that set. The IDP syntax for the cardinality of a set S is 'card S' or '# S'. For the first kind of sets, this denotes the number of formulas phi_i that are true. For the second kind, this is interpreted as the number of tuples (a_1,a_2,..., a_n) such that phi is true.

Sum: Let S be a set of the second form, i.e., of the form '{ x_1 x_2 ... x_n : phi : t }'. Then the interpretation of 'sum S' denotes the number

$$\sum_{(\mathtt{a\_1,a\_2,\ldots,a\_n})|I\models\mathtt{phi}} \mathtt{t},$$

i.e., it is the sum of all the terms for which there exist a_1,..., a_n that make the formula phi true. For sets of the first sort, this is interpreted as

$$\sum_{i|I\models\mathtt{phi\_i}} \mathtt{t_i}.$$

Product: Products are defined similarly to sum. The syntax for the product of S is prod S.

Maximum: One can write 'max S' to denote the maximum value of the terms in S, i.e.,

$$\max_{(\mathtt{a\_1,a\_2,\ldots,a\_n})|I\models\mathtt{phi}} \mathtt{t}$$

for sets of the second sort. Sets of the first sort are handled analogously.

Minimum: To get the minimum value, write 'min S'.

When using cardinality, the terms do not matter. You can choose to write 1 for every term, but are also allowed to leave out the terms.

6.4 Partial functions

Normally, functions are total: they assign an output value to each of the input values. On the other hand, *partial* functions do not necessarily have this property. In IDP, a partial function F can arise in different situations. Either F is explicitly declared as partial, or it is a built-in integer function (for example modulo).

The semantics of a partial function F is given by transforming constraints and rules where F occurs as follows:

- in a *positive* context, $P(\ldots, F(x), \ldots)$ is transformed to $\forall y\ (F(x) = y \Rightarrow P(\ldots, y, \ldots))$;
- in a negative context, $P(\ldots, F(x), \ldots)$ is transformed to $\exists y\ (F(y) = y \wedge P(\ldots, y, \ldots))$.

Here, $P(\ldots, F(x), \ldots)$ occurs in a positive context if it occurs in sentence and in the scope of an even number of negations, or it occurs in a body of a rule and in the scope of an odd number of negations. All other occurrences are in a negative context.

6.5 The Type of a Variable

There are two ways to assign a type t to a variable v:

- Explicitly mention the type of v between '`[`' and '`]`' when v is quantified. Then v gets type t in the scope of the quantifier. E.g.,

```
theory T: V {
    ! MyVar[MyType] : ? MyVar2[MyType2] MyVar3[MyType3] : //...
}
```

- Do not mention the type of v but let the system automatically derive it. The rest of this section explains how this is done.

6.5.1 Automatic derivation of types for variables

We distinguish between *typed* and *untyped* occurrences. The following are typed occurrences of a variable x:

- an occurrence as argument of a non-overloaded predicate: $P(\ldots, x, \ldots)$;
- an occurrence as argument of a non-overloaded function: $F(\ldots, x, \ldots) = \ldots$;
- an occurrence as return value of a non-overloaded function: $F(\ldots) = x$ or $F(\ldots) \neq x$.

All others positions are untyped.

An overloaded predicate or function symbol can be disambiguated by specifying its vocabulary and/or types. E.g.,

```
! x: MyVoc::P[A,A](x,x).
! y: ?1 x : F[A:A](x) = y.
MyVoc::C[:A] > 2.
```

In this case, the occurrences of all variables are typed.

Basically, if a variable occurs in one typed position, it gets the type of that position. If a declared variable with type `T_1` occurs in a typed position of type `T_2`, then `T_1` and `T_2` should have a common ancestor type.

The more complicated cases arise when a variable does not occur in any typed position, or it occurs in two typed positions with a different type. The system is designed to give a reasonable type to such variables. However, the choices made by the system might be ad hoc or not the ones the user intended, hence, every time IDP derives a type, it will give a warning, including which type it derived for the variable.

First consider the case where a variable occurs in typed positions with different types. If all the typed positions where the variable occurs have a common ancestor type T, then the variable is assigned the least common ancestor of those types. If they do not have a common ancestor, no derivation is done.

Now consider the case where a variable does not occur in a typed position. Then, the IDP system tries to find out what the type of the variable should be using its occurrences in untyped positions in built-in overloaded functions. For example, when a variable x only occurs in $x = t$, then x will get the same type as t. It's always safer to declare a type for the variable in this case. If it is not possible to derive a type for x in this way either, the IDP system reports an error.

7 Terms

Besides from appearing in theories, terms can also be defined separately, for example for use in a minimize inference. The syntax for declaring a term `MyTerm` over a vocabulary `MyVoc` is

```
term MyStruct: MyVoc {
   //contents of the term
}
```

8 Queries

A query with name `MyQuery` over a vocabulary `MyVoc` is declared by

```
query MyQuery: MyVoc {
   //contents of the query
}
```

Here "contents of the query" is of the following form

```
{ MyVar1 MyVar2 ... MyVarn : MyFormula}
```

where all free variables of the FO($\cdot$) formula `MyFormula` appear among the `MyVar1`, `MyVar2`,...,`MyVarn`.

9 Structures

A (three-valued) structure with name `MyStruct` over a vocabulary `MyVoc` is declared by

```
structure MyStruct: MyVoc {
   //contents of the structures
}
```

or by

```
aspstructure MyStruct: MyVoc {
   //contents of the structures
}
```

9.1 Contents of a structure

A particular input to a problem can be given by giving a (three valued) interpretation to all types and some predicate and function symbols of a given vocabulary. Here, we describe the different ways to specify a structure.

9.1.1 Type Enumeration

The syntax for a type enumeration is

```
MyType = { El_1; El_2; ... ; El_n }
```

where `MyType` is the name of the enumerated type and `El_1; El_2; ... ; El_n` are the names of the objects of that type. Names of objects can be (positive and negative) integers, strings, chars, compound domain elements, or identifiers that start with an upper- or lowercase letter. Identifiers are shorthands for strings (without the quotes) and can be interchanged within IDP specifications. In lua-blocks, however, only the string variant should be used.

If one type is a subtype of another, all elements of the subtype are added to the supertype also. In the case all subtypes of a given type are specified, the supertype is derived to be the union of all elements of the subtypes. If a type is not specified, all domain elements of that type that occur in a predicate or function interpretation (see below) are automatically added to that type.

9.1.2 Predicate Enumeration

The syntax for enumerating all tuples for which a predicate `MyPred` with n arguments is true is as follows.

```
MyPred = { El_1_1,..., El_1_n;
           ... ;
           El_m_1,..., El_m_n
         }
```

It is also possible to write parentheses around tuples.

```
MyPred = { (El_1_1,..., El_1_n);
           ... ;
           (El_m_1,..., El_m_n)
         }
```

This notation makes it possible to state that a proposition (a predicate with no arguments) is true, by using an empty tuple.

```
true = { () }
false = { }
```

However, we recommend using `true` and `false` instead of { `()` } and {}.

9.1.3 Function Enumeration

The syntax for enumerating a function `MyFunc` with n arguments is

```
MyFunc = { El_1_1,...,El_1_n -> El_1;
    ...;
    El_m_1,...,El_m_n -> El_m
}
```

To give the interpretation of a constant, one can simply write 'MyConst = El' instead of 'MyConst = { -> El }'.

9.1.4 Three-Valued Predicate/Function interpretations

Three-valued interpretations are given by either

- enumerating the certainly true and certainly false tuples;
- enumerating the certainly true and the unknown tuples;
- enumerating the unknown and the certainly false tuples.

The third set of tuples can than be derived from the two that were given. To specify which tuples are enumerated, use `<ct>`, `<cf>` and `<u>`. For example

```
P<ct> = { /* enumeration of the certainly true tuples of P */ }
P<u> = { /* enumeration of the unknown tuples of P */ }
```

9.1.5 Interpretation by Procedures

The syntax

```
P = procedure MyProc
```

is used to interpret a predicate or function symbol P by a procedure MyProc (see below). If P is an n-ary predicate, then MyProc should be an n-ary procedure that returns a boolean. If P is an n-ary function, then MyProc should be and n-ary function that returns a number, string, or compound domain element (depending on the return type of P).

9.1.6 Shorthands

Shorthands like 'MyType = {1..10; 15..20}' or 'MyType = { a..e; A..E }' may be used for enumerating types or predicates with only one argument.

9.2 ASP structures

An ASP structure consists of a list of facts in the usual ASP syntax. In particular, everything from a % till the end of the line is considered a comment, and - before an atom denotes classical negation (negation as failure is not available). A fact about functions is written like F(a)=b or -F(c)=d.

10 Procedures

10.1 Declaring a procedure

A procedure with name `MyProc` and arguments `A1`, ..., `An` is declared by

```
procedure MyProc(A1,...,An) {
   // contents of the procedure
}
```

Inside a procedure, any chunk of Lua code can be written. For Lua's reference manual, see `http://www.lua.org/manual/5.1/`. In the following, we assume that the reader is familiar with the basic concepts of Lua. Like in most programming languages, a procedure should be declared before it can be used in other procedures (either in the same file or in earlier included files). There is one exception to this: procedures in the global namespace (for example, all built-in procedures) can always be used, no matter what.

10.2 IDP types

Besides the standard types of variables available in Lua, the following extra types are available in IDP procedures.

sort A set of sorts with the same name. Can be used as a single sort if the set is a singleton.

predicate_symbol A set of predicates with the same name, but possibly with different arities. Can be used as a single predicate if the set is a singleton. If `P` is a predicate_symbol and `n` an integer, then `P/n` returns a predicate_symbol containing all predicates in `P` with arity `n`. If `s1`, ..., `sn` are sorts, then `P[s1,...,sn]` returns a predicate_symbol containing all predicates Q/n in `P`, such that the i'th sort of Q belongs to the set `si`, for $1 \leq i \leq n$.

function_symbol A set of first-order functions with the same name, but possibly with different arities. Can be used as a single first-order function if the set is a singleton. If `F` is a function_symbol and `n` an integer, then `F/n:1` returns a function_symbol containing all function in `F` with arity `n`. If `s1`, ..., `sn`, `t` are sorts, then `F[s1,...,sn:t]` returns a function_symbol containing all functions G/n in `F`, such that the i'th sort of F belongs to the set `si`, for $1 \leq i \leq n$, and the output sort of G belongs to `t`.

symbol A set of symbols of a vocabulary with the same name. Can be used as if it were a sort, predicate_symbol, or function_symbol.

vocabulary A vocabulary. If `V` is a vocabulary and `s` a string, `V[s]` returns the symbols in `V` with name `s`.

compound A domainelement of the form $F(d_1, \ldots, d_n)$, where F is a first-order function and d_1, ..., d_n are domain elements.

tuple A tuple of domain elements. `T[n]` returns the `n`'th element in tuple `T` (This is 1-based, thus the first element is referred to as `T[1]`).

predicate_table A table of tuples of domain elements.

predicate_interpretation An interpretation for a predicate. If `T` is a predicate_interpretation, then `T.ct`, `T.pt`, `T.cf`, `T.pf` return a predicate_table containing, respectively, the certainly true, possibly true, certainly false, and possibly false tuples in `T`.

function_interpretation An interpretation for a function. `F.graph` returns the predicate_interpretation of the graph associated to the function_interpretation `F`.

structure A first-order structure. To obtain the interpretation of a sort, singleton predicate_symbol, or singleton function_symbol `symb` in structure `S`, write `S[symb]`.

theory A logic theory.

options A set of options.

namespace A namespace.

overloaded An overloaded object.

10.3 Built-in procedures

A lot of procedures are already built-in. The command `help()` gives you an overview of all available sub-namespaces, procedures,.... The `stdspace` namespace contains all built-in procedures.

10.3.1 stdspace

The `stdspace` contains the following procedures:

elements(d) Returns a procedure, stopargument, and a beginindex (hence, a Lua-iterator) such that "for e in elements(d) do ... end" iterates over all elements in the given domain d.

help(namespace) List the procedures in the given namespace.

idptype(something) Returns custom typeids for first-class idp citizens.

tuples(table) Returns a Lua-iterator such that "for t in tuples(table) do ... end" iterates over all tuples in the given predicate table.

All procedures in `stdspace` are included in the global namespace and hence can be called by `procedure()` instead of by `stdspace.procedure()`.

Furthermore: `stdspace` contains the following subnamespaces:

idpintern

inferences

options

structure

theory

10.3.2 idpintern

This namespace contains internal procedures of the IDP system. Using them is unsafe, since most procedures there are not tested well, or not documented. We recommend this only for advanced users. `idpinter` is not included in the global namespace; help for idpintern can be obtained by `help(stdspace.idpintern)`.

10.3.3 inferences

The `inferences` namespace and all its procedures are included in the global namespace. Hence `inferences.xxx` should never be used. This namespace contains several inference methods:

calculatedefinitions(theory,structure) Make the structure more precise than the given one by evaluating all definitions with known open symbols. This procedure works recursively: as long as some definition of which all open symbols are known exists, it calculates the definition (possibly deriving open symbols of other definitions).

ground(theory,structure) Create the reduced grounding of the given theory and structure.

groundeq(theory,structure,modeleq) Create the reduced grounding of the given theory and structure. modeleq is a boolean parameter: whether or not the grounding should preserve the number of models (it always preserves satisfiability but might not preserve the number of models if modeleq is false).

printgrounding(theory,structure) Print the reduced grounding of the given theory and structure. MEMORY EFFICIENT: does not store the grounding internally.

groundpropagate(theory,structure) Return a structure, made more precise than the input by grounding and unit propagation on the theory. Returns nil when propagation makes the givens structure inconsistent.

optimalpropagate(theory,structure) Return a structure, made more precise than the input by generating all models and checking which literals always have the same truth value This propagation is complete: everything that can be derived from the theory will be derived. Returns nil when propagation results in an inconsistent structure.

propagate(theory,structure) Returns a structure, made more precise than the input by doing symbolic propagation on the theory. Returns nil when propagation results in an inconsistent structure.

minimize(theory,structure,term) Returns all models of the theory that extend the given structure and such that the term is minimal.

modelexpandpartial(theory,structure) Apply model expansion to theory T, structure S. The result is a table of (possibly three-valued) structures that are more precise then S and that satisfy T and, if getoptions().trace == true, a trace of the solver. (this procedure is equivalent to first calling modelexpandpartial and subsequently calling alltwovaluedextensions.

modelexpand(theory,structure) Apply model expansion to theory T, structure S. The result is a table of two-valued structures that are more precise then S and that satisfy T and, if getoptions().trace == true, a trace of the solver. (this procedure is equivalent to first calling modelexpandpartial and subsequently calling alltwovaluedextensions.

query(query,structure) Generate all solutions to the given query in the given structure. The result is the set of element-tuples that certainly satisfy the query in the structure.

sat(theory,structure) Checks satisfiability of the given theory-structure combination. Returns true if and only if there exists a model extending the structure and satisfying the theory.

10.3.4 options

Like the `inferences` namespace, the `options` namespace and all its procedures are included in the global namespace. This namespace consists of the following procedures:

getoptions() Get the current options.

newoptions() Create new options, equal to the standard options.

setascurrentoptions(options) Sets the given options as the current options, used by all other commands.

10.3.5 structure

Also this namespace and all its procedures are included in the global namespace. Here, you can find several procedures for manipulating logical structures.

alltwovaluedextensions(structure) This procedures takes one (three-valued) structure and returns all structures over the same vocabulary that extend the given structure and are two-valued.

alltwovaluedextensions(table) This procedures takes a table of structures and returns all two-valued extensions of any of the given structures.

clean(structure) Modifies the given structure (the structure is the same, but its representation changes): transforms fully specified three-valued relations into two-valued ones. For example if `P` has domain `[1..2]` and in the given structure, `P<ct> = {1}, P<cf> = {2}`, in the end, `P` is `{1}`.

clone(structure) Returns a structure identical to the given one.

isconsistent(structure) Check whether the structure is consistent.

makefalse(predicate_interpretation,table) Sets all tuples of the given table to false. Modifies the table-interpretation.

makefalse(predicate_interpretation,tuple) Sets the interpretation of the given tuple to false. Modifies the table-interpretation.

maketrue(predicate_interpretation,table) Sets all tuples of the given table to true. Modifies the table-interpretation.

maketrue(predicate_interpretation,tuple) Sets the interpretation of the given tuple to true. Modifies the table-interpretation.

makeunknown(predicate_interpretation,table) Sets all tuples of the given table to unknown. Modifies the table-interpretation.

makeunknown(predicate_interpretation,tuple) Sets the interpretation of the given tuple to unknown. Modifies the table-interpretation.

newstructure(vocabulary,string) Create an empty structure with the given name over the given vocabulary.

createdummytuple() Create an empty tuple.

iterator(domain) Create an iterator for the given sorttable.

iterator(predicate_table) Create an iterator for the given predtable.

size(predicate_table) Get the size of the given table.

range(number,number) Create a domain containing all integers between First and Last.

10.3.6 theory

The `theory` namespace and all its procedures are included in the global namespace. It contains methods for manipulating theories, most of which modify the given theory.

clone(theory) Returns a theory identical to the given one.

completion(theory) Add definitional completion of all the definitions in the theory to the given theory. Modifies its argument.

flatten(theory) Rewrites formulas with the same operations in their child formulas by reducing the nesting. For example $a \wedge (b \wedge c)$ will be rewritten to $a \wedge b \wedge c$. Modifies the given theory.

merge(theory,theory) Create a new theory which is the result of combining (the conjunction of) both input theories.

pushnegations(theory) Push negations inwards until they are right in front of literals. Modifies the given theory.

removenesting(theory) Move nested terms out of predicates (except for the equality-predicate) and functions. Modifies the given theory.

10.3.7 Miscellaneous

TODO: WHY is the parse method directly in global_namespace?

parse(string) Parses the given file and adds its information into the datastructures.

10.3.8 The mx library

The mx standard library file contains some useful commands for model expansion. It can be used by

```
#include <mx>
```

All commands in the mx standard library file are defined in a namespace with the same name (and hence should be called by `mx::command(...)`. The following commands are supported by mx:

one(theory,structure) Does model expansion but only searches for one model (no matter what the `nbmodels` option is set to). Returns this structure (in contrast to the standard modelexpansion which returns a list of structures).

all(theory,structure) Returns all models of the theory that extend the given structure.

printmodels(list) Prints a given list of models or prints unsatisfiable if the list is empty.

10.3.9 The table_utils library

The table_utils standard library file can be include by

```
#include <table_utils>
```

It contains several useful commands for manipulating tables, converting predicate tables to lua-tables,. . . . TODO: document

11 Options

TODO: How does the verbosity work? The IDP system has various options. To print the current values of all options, use `print(getoptions())`. To set an option, you can use the following lua-code

```
stdoptions.MyOption = MyValue
```

where `MyOption` is the name of the option and `MyValue` is the value you want to give it. If you want to have multiple option sets, you can make them with them with

```
FirstOptionSet = newOptions()
SecondOptionSet = newOptions()
FirstOptionSet.MyOption = MyValue
SecondOptionSet.MyOption = MyValue
```

To activate an option set, use the procedure `setascurrentoptions(MyOptionSet)`. From that moment, `MyOptionSet` will be used in all comands.

11.1 Verbosity options

groundverbosity = [0..max(int)] Verbosity of the grounder. The higher the verbosity, the more debug information is printed.

satverbosity = [0..max(int)] Like groundverbosity, but controls the verbosity of MiniSAT(ID)

propagateverbosity = [0..max(int)] Like groundverbosity, but controls the verbosity of the propagation.

11.2 Modelexpansion options

nbmodels = [0..max(int)] Set the number of models wanted from the modelexpansion inference. If set to 0, all models are returned.

trace = [false, true] If true, the procedure modelexpand produces also an execution trace of MiniSAT(ID)

11.3 Propagation options

groundwithbounds = [false, true] Enable/disable bounded grounding (if enabled, first do symbolic propagation to provide ct and cf bounds for formulas to reduce the size of the grounding in every inferences that grounds (groundpropagate/ground/modelexpand/...)).

longestbranch = [0..2147483647] The longest branch allowed in BDDs during propagation. The higher, the more precise the propagation will be (but also, the more time it will take).

nrpropsteps = [0..2147483647] The number of propagation steps used in the propagate-inference. The higher, the more precise the propagation will be (but also, the more time it will take).

relativepropsteps = [false, true] If true, the total number of propagation steps is nrpropsteps multiplied by the number of formulas.

11.4 Printing options

language = [ecnf, idp, tptp] The language used when printing objects.

longnames = [false, true] If true, everything is printed with reference to their vocabulary. For example, a predicate `P` from vocabulary `V` will be printed as `V::P` instead of `P`.

11.5 General options

timeout = [0..max(int)] Set the timeout for inferences (in seconds)

seed = [0..max(int)] Set the seed for the random generator (used in the estimators for BDDs and in the SAT-solver)

randomvaluechoice = [false, true] Controls the solver: if set to true, the assignment to choice literals is random, if set to false, the solver default assigns false to choice literals.

12 IDP^2 vs IDP^3

Users of the old IDP^2-system will still have a bunch of files with different syntaxis. Here are the basic rules for transforming them to IDP^3.

- Blocks in the IDP^2 system now are structures, vocabularies and theories.

- Everything that used to be in a Given: Declare: or Find: block, now belongs to a vocabulary.
  - The sentences and definitions from the Satisfying: block should be moved to a theory.
  - The Data: block is essentialy what is now a structure.

- UNA-DCA declarations in the vocabulary are no longer allowed:
  - In the IDP^2 system you could write `type Direction=Up;Down`. This had the effect of creating new domain elements Up and Down and constants Up and Down that could be used in the theory (Satisfying block).
  - For the moment, this is not yet possible in the IDP^3 system. If you want the same effect: you add `type Direction`, `Up: Direction` and `Down: Direction` to the vocabulary (this creates the type and makes the constants. In the structure, you interprete Direction by `Direction = u;d` and you interprete the constants by `Up = u; Down = d`.

- Three-valued interpretations have a slightly different syntax.
  - In a Data: block of the IDP^2 system, one could write `P = A;BC`, meaning that `P` should be certainly true for `A` and `B` and that `P` should be certainly false for `C`. In IDP^3, we write (as explained above `P<ct> = A;B` and `P<cf> = C`.

- Aggregates have a slightly different syntaxis (see above).

13 Common errors and warnings

In this section the most common error and warning messages you might encounter are listed and a short explanation is given. Some general recommendations:

- *Earliest exception first*: As one error might propagate to more errors in code which is in fact correct, always start resolving errors from the first one encountered.

- *Warnings are important*: Usually, warnings notify that some part of the specification can be *interpreted in multiple ways* (and the system tells you which choice it made) or that the system *suspects you made an error*, although the specification does not violate any rule. So always at least verify whether the correct resolution was made and preferably change the specification to remove the warning.

13.1 Syntax error

“” Some part of the specification is invalid $FO(\cdot)^{\texttt{IDP}}$ syntax. Some possible fixes might be presented by the system. General recommendations to prevent this kind of errors:

- Declare all symbols in the vocabulary and check the number of arguments.

- Check the number of brackets.

- Variables are separated by whitespace, arguments by commas and tuples by “;”.

13.2 Unquantified variables

“” The system encountered a 0-ary symbol which was not quantified and not declared in the vocabulary. It assumes it is then a variable which is taken to have the default quantification in the context in which it occurs, but warns you as you might have intended another quantification or had intended it to be some other or undeclared predicate or constant symbol.

13.3 Derived type

“” This warning is produced if the type of a variable or term is ambiguous. It makes a usually safe guess to the intended type, but it should at least be checked and preferably stated explicitly.

13.4 Underivable type

“” In some cases, there are multiple types possible and no safe guess is available, such as two possible types which have no related parents. In that case, the intended type has to be stated explicitly.

13.5 Infinite grounding

“” When constructing the grounding, the system might detect that it has to make an infinite grounding (it can also go undetected in some cases). It is guaranteed that this will not happen using default options if all variable types are explicitly stated and no infinite types are used for any variable or argument type (parent types can be infinite).