DiMo– A Tool for Discrete Modelling Using Propositional Logic Version 0.3.0

Martin Lange, Maurice Herwig
University of Kassel, Germany

November 25, 2022

Contents

1 Discrete Modelling Problems

The DiMo language can be used to model various discrete decision or computation problems as logical problems in quantifier-free logic. The four logical problems that are supported are

- the *satisfiability problem* for finding *some* model of a given formula;
- the *validity problem* for checking whether *all* interpretations satisfy a given formula, respectively for finding a countermodel;
- the *model class* problem for finding *all* models of a given formula;
- the *equivalence* problem for checking whether two given formulas are semantically equivalent on a given set of propositions, respectively finding a model (in those propositions) that separates the two.

The DiMo language can be used to specify formulas *parametrised* by natural numbers. The logical problems above are then applied to *all* instances of such formula schemes.

2 Basic Design Principles of the Language

Specifying problems. A DiMo input starts with the specification of the problem, defined by the keywords `SATISFIABLE`, `VALID`, `EQUIVALENT` or `MODELS`, followed by the formula scheme to be tested and the output control.

In case of the equivalence problem this is followed by the keyword `TO` and a second formula scheme. Note that this is a generalisation of the standard equivalence problem which checks equality between the model classes of two formulas. Here the input is given by two formulas φ and ψ and a set of propositional variables L, and the question is to decide whether the model classes of φ and ψ coincide when restricted to L. In other

words: for every propositional variable assignment ϑ s.t. $\vartheta \models \varphi$, is there a variable assignment ϑ' s.t. $\vartheta' \models \psi$ and $\vartheta(X) = \vartheta'(X)$ for all $X \in L$. Moreover, does the same hold vice-versa between ψ and φ? Note that this generalises semantic equivalence in the sense that each pair of semantically equivalent formulas is also equivalent when restricted to any L but not vice-versa.

Specifying interesting predicates. One can then optionally specify the (names) of the predicate symbols which should be considered in the output. This can be used to interact with these predicates in the control of the output program flow. This is indicated by the keyword `PROPOSITIONS`, followed by a list of those predicate names. Note that all predicate symbols not listed here can't use in the output controll. Hence, without the `PROPOSITIONS` section in the input, no models / countermodels are generated.

Furthermore, when the chosen problem is the equivalence problem then it is taken to be restricted to those propositions that are given in this section. Consequently, when no propositions are given here, the equivalence problem boils down to a simple equi-satisfiability check.

Specifying parameters. This section is followed by a section specifying the parameters to the formula scheme, started with the keyword `PARAMETERS`. Parameters are variables for natural numbers and can be denoted with an arbitrary alphanumeric string starting with a lowercase letter. Parameters to the scheme are specified as a comma-separated list of name-domain pairs, for instance

```
n : NAT
```

defines a parameter `n` that stands for any number in $\{0, 1, \ldots\}$. Other means to specify a domain are, for instance

`{3,..}`	for the set $\{3, 4, 5, \ldots\}$,
`{3,7,..}`	for the set $\{3, 7, 11, 15, \ldots\}$,
`{3,7,..,20}`	for the set $\{3, 7, 11, 15, 19\}$,
`{3,7,20,4,16}`	for the set $\{3, 4, 7, 16, 20\}$.

It is possible to impose certain simple conditions on parameter combinations by appending the keyword `WITH`, followed by a comma-separated non-empty list of basic comparisons like `n < m`. Allowed comparisons are *equals*, *greater-than*, *less-than*, *greater-or-equals*, *less-or-equals*, and *unequals*.

Specifying formula abbreviations. There is no distinction between a predicate, like `P(n,m)` and an auxiliary formula. It is thus possible to abbreviate certain formula parts and give the definition of these predicates in a final section started with the keyword `WHERE`, followed by a list of definitions of the form

pattern `=` *formula*

Note that there is no comma separation between these definitions. A pattern in this context is a predicate symbol, possibly followed by a comma-separated list of variables or numbers, enclosed in parentheses, i.e. `P` or `P(n)` or `P(0,n)`. Note that DiMo does not type-check predicates w.r.t. their arity. A symbol should only be used with the same arity throughout the entire specification.

Specifying terms and finite sets. Terms are made up from integers and variables which are either specified in the `PARAMETERS` section or introduced by the junctors `FORALL` and `FORSOME`, using the arithmetical operations of addition, subtraction and multiplication. Terms can be used as arguments to predicate symbols, for instance as in `X(n,n+2)`.

A finite set is an enumeration of terms like `{1,n,n+3,m-2,n*m+i,10}`. It is possible to define linear sets by specifying the least, possibly the second-to-least, and the last element. This follows the same convention as used in the specification of domains above. For instance, `{1,..,n}`, `{0,2,..,n-1}` and `{n,2*n,..,m*n}` are valid specifications of such sets provided that the variables have been defined either as parameters or via generalised junctors, see below.

There are also the operators `MIN` and `MAX` which turn sets into terms. For instance, `MIN {n+1,9,k,m-n}` is a term.

Specifying formulas. The DiMo language supports propositional logic with the standard Boolean unary or binary operators $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$, as well as conjunctions and disjunctions over finite sets of indices (e.g. $\bigwedge_{i=1}^{n} \ldots$). For instance, formula (scheme) $\bigwedge_{i=0}^{n-1} X_i \leftrightarrow X_{i+1}$, parametrised by a natural number n would be specified as

```
FORALL i: {0,..,n-1} . X(i) <-> X(i+1)
```

In general, such a junctor defines a new variable `i` which can be used in the formula following the dot. A disjunction $\bigvee$ is denoted using the keyword `FORSOME`.

3 Examples

3.1 The n-Queens Problem

A standard discrete modelling problem is the following: given a chess board of size $n \times n$ with $n \geq 1$, place n queens on this board such that none of them threatens another. A queen in chess can move an arbitrary distance along a row, column or diagonale. Clearly, the problem amounts to placing one queen in each column for example. Moreover, to exclude possible threats between queens, by symmetry it suffices to consider one direction per row/column/diagonale only, for example no queen should "see" another queen in the directions east, north, north-east and south-east.

The question of finding a solution to the n-queens problem boils down to finding a model of the following propositional formula (scheme) φ_n for given n. We use propositional variables $D_{i,j}$ with $1 \leq i, j, \leq n$ to intuitively denote that the square in column i and row j is occupied by a queen.

$$\varphi_n := \bigwedge_{i=1}^{n} \bigvee_{j=1}^{n} D_{i,j} \qquad \text{"}\textit{in each column place a queen in some row}\text{"}$$

$$\wedge \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n-1} D_{i,j} \rightarrow \bigwedge_{k=j+1}^{n} \neg D_{i,k} \qquad \text{"}\textit{no two queens per row}\text{"}$$

$$\wedge \bigwedge_{i=1}^{n-1} \bigwedge_{j=1}^{n} D_{i,j} \rightarrow \bigwedge_{k=i+1}^{n} \neg D_{k,j} \qquad \text{"}\textit{no two queens per column}\text{"}$$

$$\wedge \bigwedge_{i=1}^{n-1} \bigwedge_{j=1}^{n-1} D_{i,j} \rightarrow \bigwedge_{\substack{k=1 \\ \text{s.t. } i+k \leq n \text{ and } j+k \leq n}}^{\infty} \neg D_{i+k,j+k} \qquad \text{"}\textit{no two queens per diagonale up}\text{"}$$

$$\wedge \bigwedge_{i=1}^{n-1} \bigwedge_{j=2}^{n} D_{i,j} \rightarrow \bigwedge_{\substack{k=1 \\ \text{s.t. } i+k \leq n \text{ and } j-k \geq 1}}^{\infty} \neg D_{i+k,j-k} \qquad \text{"}\textit{no two queens per diagonale down}\text{"}$$

The problem of finding a model for each instance of φ_n can be specified as a DiMo program as follows.

```
SATISFIABLE NQueens(n)
PROPOSITIONS D
PARAMETERS n: {1,..}
FORMULAS
  NQueens(n) = AtLeastOneInEachRow(n)
             & AtMostOneInEachRow(n)
             & AtMostOneInEachColumn(n)
             & AtMostOneInEachDiagUp(n)
             & AtMostOneInEachDiagDown(n)

  AtLeastOneInEachRow(n)     =
    FORALL i: {1,..,n}. FORSOME j: {1,..,n}. D(i,j)

  AtMostOneInEachRow(n)      =
    FORALL i: {1,..,n}. FORALL j: {1,..,n-1}.
      D(i,j) -> FORALL k: {j+1,..,n}. -D(i,k)

  AtMostOneInEachColumn(n)   =
    FORALL i: {1,..,n-1}. FORALL j: {1,..,n}.
      D(i,j) -> FORALL k: {i+1,..,n}. -D(k,j)

  AtMostOneInEachDiagUp(n)   =
    FORALL i: {1,..,n-1}. FORALL j: {1,..,n-1}.
      D(i,j) -> FORALL k: {1,..,MIN {n-i,n-j}}. -D(i+k,j+k)

  AtMostOneInEachDiagDown(n) =
    FORALL i: {1,..,n-1}. FORALL j: {2,..,n}.
      D(i,j) -> FORALL k: {1,..,MIN {n-i,j-1}}. -D(i+k,j-k)
```

Testing this problem with the DiMo tool reveals solutions in terms of propositional variable assignments from which the placement of queens on the chess boards of respective sizes can immediately be derived.

```
[~] dimo.native examples/nQueens.dm
Instance n=1 ..................................... satisfiable.
  Satisfying assignment: D(1,1)
Instance n=2 ..................................... unsatisfiable.
Instance n=3 ..................................... unsatisfiable.
Instance n=4 ..................................... satisfiable.
  Satisfying assignment: -D(1,1), -D(1,2), D(1,3), -D(1,4), D(2,1), -D(2,2), -D(2,3),
  -D(2,4), -D(3,1), -D(3,2), -D(3,3), D(3,4), -D(4,1), D(4,2), -D(4,3), -D(4,4)
...
```

The output shows that the DiMo tool has chosen the standard enumeration $n = 1, 2, \ldots$ for the parameter `n`, instantiated the formulas scheme for each instance of this enumeration and checked each of them for satisfiability subsequently. For $n = 2$ and $n = 3$ there is no solution to the n-queens problem as one can easily check. Consequently, these instances are reported as being unsatisfiable. On the other hand, the instances $n = 1, 4, 5, \ldots$ are satisfiable, and a propositional model is found. It is easy to read off a solution to the n-queens problem from the assignment satisfying the n-th formula, given the interpretation of the propositions $D_{i,j}$ mentioned above.

3.2 A Simple Combinatorial Problem

As a second example we consider the simple combinatorial problem of choosing exactly m out of n items. This can equally be modelled as the problem of finding some or all models of a propositional formula $\varphi_{n,m}$ with $n \geq 1$ and $0 \leq m \leq n$. We use propositions A_i to denote that the i-th element is chosen. The straight-forward scheme

$$\bigvee_{\substack{I \subseteq \{1,\ldots,n\} \\ |C|=m}} \Big(\bigwedge_{i \in I} A_i\Big) \wedge \bigwedge_{i \in \{1,\ldots,n\} \setminus I} \neg A_i\Big)$$

cannot be written down as such in the DiMo language because variables for sets of natural numbers are not supported (yet). However, note that this is a formula of size $\mathcal{O}(n \cdot \binom{n}{m}) = \mathcal{O}(n^{m+1})$, and there is a much shorter way of specifying this which uses formulas of size $\mathcal{O}(n^2)$ only. The trick is to use auxiliary propositions $B_{i,j}$ to abbreviate the fact that at least j out of the first i elements have chosen, or equivalently, at least j of the $A_1, \ldots, A_i$ need to be set to true. Consider

$$\begin{aligned}
\varphi_{n,m} :=& B_{n,m} && \text{“at least } m \text{ out of (the first) } n \text{ are chosen”} \\
& \wedge \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} B_{i,j} \rightarrow && \text{“if } j \text{ of } i \text{ are chosen then} \\
& \qquad A_i \wedge B_{i-1,j-1} && \text{the } i\text{-th is chosen and } j\text{-1 of } i\text{-1 are chosen} \\
& \qquad \vee B_{i-1,j} && \text{or } j \text{ of } i\text{-1 are chosen”} \\
& \wedge \bigwedge_{j=1}^{m} \neg B_{0,j} && \text{“it is impossible to choose } j \text{ from 0 elements”}
\end{aligned}$$

Using the same principle one can also demand that at the same time *at most* m out of n are chosen. Note that this is the case if and only if *at least* $n-m$ are *not chosen*. Based on this, the following DiMo program specifies the problem of choosing exactly m out n elements represented by propositions A_i. The auxiliary propositions $B_{i,j}$ are hidden away from output by not listing them in the `PROPOSITIONS` section.

```
MODELS       ExactlyMofN(n,m)
PROPOSITIONS A
PARAMETERS   n : {1,..}, m : NAT WITH m <= n
FORMULAS

  ExactlyMofN(n,m) = AtLeastMChosen(n,m) & AtMostMChosen(n,m)

  AtLeastMChosen(n,m) = B(n,m) & BsWellBehaved(n,m) & BsEndOk(m)
  BsWellBehaved(n,m)  = FORALL i : {1,..,n}. FORALL j: {1,..,m}.
                          B(i,j) -> A(i) & B(i-1,j-1) | B(i-1,j)
  BsEndOk(m)          = FORALL j : {1,..,m}. -B(0,j)

  AtMostMChosen(n,m) = AtLeastNminusMNotChosen(n,n-m)
```

```
  AtLeastNminusMNotChosen(n,m) = N(n,m) & NsWellBehaved(n,m) & NsEndOk(m)
  NsWellBehaved(n,m) = FORALL i : {1,..,n}. FORALL j: {1,..,m}.
                          N(i,j) -> -A(i) & N(i-1,j-1) | N(i-1,j)
  NsEndOk(m)         = FORALL j : {1,..,m}. -N(0,j)
```

Note that formulas constraining the propositions `N(i,j)` are copied verbatim from those for the `B(i,j)`. It is not (yet) possible to re-use such code because only integer variables can be used as parameters, not propositions themselves.

Running the DiMo tool on this program results in finding all possibilities to choose m out of n elements, represented by propositional variable assignments, for some enumeration of the parameters n and m.

```
[~] dimo.native examples/exactlyMofN.dm
Instance m=0, n=1 ..............................
  Found model -A(1)
1 model found.
Instance m=0, n=2 ..............................
  Found model -A(1), -A(2)
1 model found.
Instance m=1, n=1 ..............................
  Found model A(1)
1 model found.
Instance m=0, n=3 ..............................
  Found model -A(1), -A(2), -A(3)
1 model found.
Instance m=1, n=2 ..............................
  Found model A(1), -A(2)
  Found model -A(1), A(2)
2 models found.
Instance m=0, n=4 ..............................
  Found model -A(1), -A(2), -A(3), -A(4)
1 model found.
Instance m=1, n=3 ..............................
  Found model A(1), -A(2), -A(3)
  Found model -A(1), A(2), -A(3)
  Found model -A(1), -A(2), A(3)
3 models found.
Instance m=2, n=2 ..............................
  Found model A(1), A(2)
1 model found.
Instance m=0, n=5 ..............................
  Found model -A(1), -A(2), -A(3), -A(4), -A(5)
1 model found.
...
```

The chosen enumeration of all pairs of parameters n, m such that $n \geq 1$, and $0 \leq m \leq n$ is the standard "diagonale" $(1,0), (2,0), (1,1), (3,0), (2,1), (4,0), \ldots$. Here, for each such instance DiMo finds all possibilities to choose m out of n elements, represented by a propositional variable assignment in the variables A_i.

4 Output control

Since DiMo version 0.3.0 is it possible to generate another output for computed models as the default output you can see in the examples before. For this, we can add at the end of the DiMo-language an output program that run after the calculation of a model and generate an output string. For example, we can generate a string in an HTML format, so that a web browser can render the output and show a for human very easy readable model.

The output program followed very simple program flow and use normal control statements like if-else, for and print. For the propositions that are defined in the DiMo-language by the propositions, we can evaluate the value of this proposition at every single position. For example, $D(1,2)$ give the boolean value of the Proposition D at position $1,2$. If the Proposition at the asked position is undefined, so we get undefined as result, so that we need for the output program a three-valued logic, that works more like a try-catch in normal program languages. Fig. 1 shows the evaluation of our logic for standard boolean operations.

φ \ ψ	t	f	u
t	t	f	u
f	f	f	u
u	u	u	u

φ \ ψ	t	f	u
t	t	t	u
f	t	f	u
u	u	u	u

φ	$\neg\,\varphi$
t	f
f	t
u	u

Figure 1: Evaluation of the three-valued-logic for the boolean standard operations and (left), or (center) and negation (right) with the used the abbreviations: t := true, f := false, u := undefined.

4.1 The n-Queens Problem

For the n-Queens Problem example (Sec. 3.1) we can generate with the following output program an HTML table, to show the model in a human-readable format.

```
IF HASMODEL THEN

    PRINT "<table>"

    FOR  x FROM 1 TO n STEP 1 DO

        PRINT "<tr>"

        FOR y FROM 1 TO n STEP 1 DO

            IF (x + y) MOD 2 = 0 THEN
                PRINT "<th class='even' >"
            ELSE
                PRINT "<th class='odd' >"
            ENDIF

            IF D(x,y) THEN
                PRINT "&#128081;"
            ENDIF

            PRINT "</th>"

        DONE

        PRINT "</tr>"

    DONE

    PRINT "</table>"

ELSE
    PRINT "<p>unsatisfiable</p>"
ENDIF
```

By running the DiMo-tool for the instance, $n = 4$ with the formulas for the n-Queen's problem, we get the following output string. By rendering this string with a little bit additional CSS in a browser (Fig. 2), we can see a for humans very easy readable output. Especially when the models become very large, this is particularly useful.

```
<table>
    <tr>
        <th class='even'></th>
        <th class='odd'></th>
        <th class='even'>&#128081;</th>
        <th class='odd'></th>
    </tr>
```

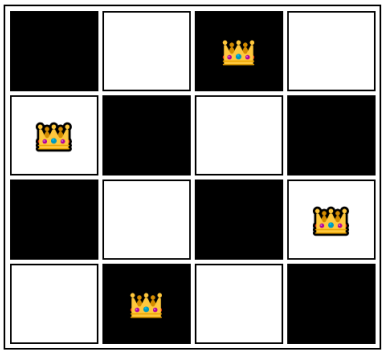

Figure 2: The rendered output for the 4-Queens Problem.

```
    <tr>
        <th class='odd'>&#128081;</th>
        <th class='even'></th>
        <th class='odd'></th>
        <th class='even'></th>
    </tr>
    <tr>
        <th class='even'></th>
        <th class='odd'></th>
        <th class='even'></th>
        <th class='odd'>&#128081;</th>
    </tr>
    <tr>
        <th class='odd'></th>
        <th class='even'>&#128081;</th>
        <th class='odd'></th>
        <th class='even'></th>
    </tr>
</table>
```

A The DiMo Tool

A.1 License

As of version 0.2.1, DiMo is available via GitHub under the BSD-3-Clauses licence.

A.2 Installation

The DiMo testing tool is written in OCaml using version 4.08.0. The simplest way to use the tool is to install the OCaml package manager `Opam`[1]. Make sure you use version 2 of `Opam`. For correct functionality, version 0.7.1 of the Opam package `ocaml-sat-solvers`[2] is needed which is not installed with previous versions of Opam.

Opam then gives a simple way to install the OCaml compiler and runtime environment. To use the precompiled byte code version this should suffice. To compile a native version of the DiMo tool one needs additional components that can be installed via `Opam`. See the file `README` for details.

B Syntax of the DiMo Language

The syntax of the DiMo input language is given by the following grammar.

$$\langle Main \rangle \ ::= \langle Problem \rangle \ \left[\ \texttt{PROPOSITIONS} \ \langle Props \rangle \ \right] \ \left[\ \texttt{PARAMETERS} \ \langle Params \rangle \ \right] \ \left[\ \texttt{FORMULAS} \ \langle Defs \rangle \ \right] \langle OutLang \rangle$$

$$\langle Problem \rangle \ ::= \texttt{SATISFIABLE} \ \langle Formula \rangle$$

[1] `https://opam.ocaml.org/`

[2] `https://opam.ocaml.org/packages/ocaml-sat-solvers/ocaml-sat-solvers.0.7.1`

$$
\begin{array}{rcl}
 & | & \texttt{VALID}\ \langle \mathit{Formula} \rangle \\
 & | & \texttt{EQUIVALENT}\ \langle \mathit{Formula} \rangle\ \texttt{TO}\ \langle \mathit{Formula} \rangle \\
 & | & \texttt{MODELS}\ \langle \mathit{Formula} \rangle \\
\langle \mathit{Props} \rangle & ::= & \epsilon \mid \langle \mathit{Ident} \rangle \left(\texttt{,}\ \langle \mathit{Ident} \rangle \right)^* \\
\langle \mathit{Ident} \rangle & ::= & \left\{ \texttt{A}, \ldots, \texttt{Z} \right\} \left\{ \texttt{A}, \ldots, \texttt{Z}, \texttt{a} \ldots, \texttt{z}, \texttt{0}, \ldots, \texttt{9}, \texttt{\_} \right\}^* \left\{ \texttt{'} \right\}^* \\
\langle \mathit{Params} \rangle & ::= & \langle \mathit{Var} \rangle\ \texttt{:}\ \langle \mathit{Domain} \rangle \left(\texttt{,}\ \langle \mathit{Var} \rangle\ \texttt{:}\ \langle \mathit{Domain} \rangle \right)^* \left[\texttt{WITH}\ \langle \mathit{Constr} \rangle \right] \\
\langle \mathit{Var} \rangle & ::= & \left\{ \texttt{a}, \ldots, \texttt{z} \right\} \left\{ \texttt{A}, \ldots, \texttt{Z}, \texttt{a} \ldots, \texttt{z}, \texttt{0}, \ldots, \texttt{9}, \texttt{\_} \right\}^* \left\{ \texttt{'} \right\}^* \\
\langle \mathit{Domain} \rangle & ::= & \texttt{Nat} \\
 & | & \texttt{\{}\ \langle \mathit{Number} \rangle\ \texttt{,} \left[\langle \mathit{Number} \rangle\ \texttt{,} \right] \texttt{..} \left[\texttt{,}\ \langle \mathit{Number} \rangle \right] \texttt{\}} \\
 & | & \texttt{\{}\ \langle \mathit{Number} \rangle \left(\texttt{,}\ \langle \mathit{Number} \rangle \right)^* \texttt{\}} \\
\langle \mathit{Number} \rangle & ::= & \left\{ \texttt{1}, \ldots, \texttt{9} \right\} \left\{ \texttt{0}, \ldots, \texttt{9} \right\}^* \\
\langle \mathit{Constr} \rangle & ::= & \langle \mathit{Var} \rangle\ \langle \mathit{Cmp} \rangle\ \langle \mathit{Var} \rangle \left(\texttt{,}\ \langle \mathit{Var} \rangle\ \langle \mathit{Cmp} \rangle\ \langle \mathit{Var} \rangle \right)^* \\
\langle \mathit{Cmp} \rangle & ::= & \texttt{=} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<>} \\
\langle \mathit{Defs} \rangle & ::= & \epsilon \mid \langle \mathit{Pattern} \rangle\ \texttt{=}\ \langle \mathit{Formula} \rangle \left(\texttt{,}\ \langle \mathit{Pattern} \rangle\ \texttt{=}\ \langle \mathit{Formula} \rangle \right)^* \\
\langle \mathit{Pattern} \rangle & ::= & \langle \mathit{Ident} \rangle \left[\texttt{(}\ \langle \mathit{Atom} \rangle \left(\texttt{,}\ \langle \mathit{Atom} \rangle \right)^* \texttt{)} \right] \\
\langle \mathit{Atom} \rangle & ::= & \langle \mathit{Var} \rangle \mid \langle \mathit{Number} \rangle \\
\langle \mathit{Formula} \rangle & ::= & \texttt{True} \mid \texttt{False} \mid \langle \mathit{Pred} \rangle \\
 & | & \texttt{(}\ \langle \mathit{Formula} \rangle\ \texttt{)} \\
 & | & \texttt{-}\ \langle \mathit{Formula} \rangle \mid \langle \mathit{Formula} \rangle\ \langle \mathit{Junct} \rangle\ \langle \mathit{Formula} \rangle \\
 & | & \langle \mathit{GenJunct} \rangle\ \langle \mathit{Var} \rangle\ \texttt{:}\ \langle \mathit{FinSet} \rangle\ \texttt{.}\ \langle \mathit{Formula} \rangle \\
\langle \mathit{Junct} \rangle & ::= & \texttt{\&} \mid \texttt{|} \mid \texttt{->} \mid \texttt{<->} \\
\langle \mathit{GenJunct} \rangle & ::= & \texttt{FORALL} \mid \texttt{FORSOME} \\
\langle \mathit{Pred} \rangle & ::= & \langle \mathit{Ident} \rangle \left[\texttt{(}\ \langle \mathit{Term} \rangle \left(\texttt{,}\ \langle \mathit{Term} \rangle \right)^* \texttt{)} \right] \\
\langle \mathit{Term} \rangle & ::= & \langle \mathit{Atom} \rangle \mid \texttt{(}\ \langle \mathit{Term} \rangle\ \texttt{)} \mid \langle \mathit{UnOp} \rangle\ \langle \mathit{Term} \rangle \mid \langle \mathit{Term} \rangle\ \langle \mathit{BinOp} \rangle\ \langle \mathit{Term} \rangle \mid \langle \mathit{AccOp} \rangle\ \langle \mathit{FinSet} \rangle \\
\langle \mathit{UnOp} \rangle & ::= & \texttt{LOG} \mid \texttt{FLOG} \\
\langle \mathit{BinOp} \rangle & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{\^{}} \mid \texttt{MOD} \\
\langle \mathit{AccOp} \rangle & ::= & \texttt{MIN} \mid \texttt{MAX} \\
\langle \mathit{FinSet} \rangle & ::= & \texttt{\{}\ \langle \mathit{Term} \rangle\ \texttt{,} \left[\langle \mathit{Term} \rangle\ \texttt{,} \right] \texttt{..}\ \texttt{,}\ \langle \mathit{Term} \rangle\ \texttt{\}} \\
 & | & \texttt{\{}\ \langle \mathit{Term} \rangle \left(\texttt{,}\ \langle \mathit{Term} \rangle \right)^* \texttt{\}} \\
 & | & \langle \mathit{FinSet} \rangle\ \langle \mathit{SetOp} \rangle\ \langle \mathit{FinSet} \rangle \\
\langle \mathit{SetOp} \rangle & ::= & \texttt{\&} \mid \texttt{|} \mid - \\
\langle \mathit{OutLang} \rangle & ::= & \texttt{OUTPUT}\ \langle \mathit{OutProg} \rangle \\
\langle \mathit{OutProg} \rangle & ::= & \texttt{SKIP} \mid \texttt{EXIT} \mid \texttt{PRINT}\ \langle \mathit{String} \rangle \mid \texttt{PRINTF}\ \langle \mathit{String} \rangle\ \texttt{(}\ \langle \mathit{Terms} \rangle\ \texttt{)} \\
 & | & \texttt{IF}\ \langle \mathit{BExpr} \rangle\ \texttt{THEN}\ \langle \mathit{OutProg} \rangle\ \texttt{ELSE}\ \langle \mathit{OutProg} \rangle\ \texttt{UNDEF}\ \langle \mathit{OutProg} \rangle\ \texttt{ENDIF}
\end{array}
$$

$$\left| \texttt{FOR}\ \langle \mathit{Ident} \rangle\ \texttt{OF OF PROPOSITIONS DO}\ \langle \mathit{OutProg} \rangle\ \texttt{DONE} \right.$$

$$\left| \texttt{FOR}\ \langle \mathit{Var} \rangle\ \texttt{FROM}\ \langle \mathit{Term} \rangle\ \texttt{TO}\ \langle \mathit{Term} \rangle\ \texttt{STEP}\ \langle \mathit{Term} \rangle\ \texttt{DO}\ \langle \mathit{OutProg} \rangle\ \texttt{DONE} \right.$$

$$\left| \langle \mathit{OutProg} \rangle\ \langle \mathit{OutProg} \rangle \right.$$

$$\langle \mathit{BExpr} \rangle\ ::= \texttt{HASMODEL}\ \left| \langle \mathit{Props} \rangle\ \texttt{(}\ \langle \mathit{Term} \rangle\ \left[\left(\texttt{,}\ \langle \mathit{Term} \rangle \right)^* \right] \texttt{)}\ \right| \langle \mathit{Term} \rangle\ \langle \mathit{cmp} \rangle\ \langle \mathit{Term} \rangle$$

$$\left| \langle \mathit{BExpr} \rangle\ \&\langle \mathit{BExpr} \rangle\ \right| \langle \mathit{BExpr} \rangle\ |\langle \mathit{BExpr} \rangle\ \left| \neg\langle \mathit{BExpr} \rangle \right.$$

$$\langle \mathit{String} \rangle\ ::= \texttt{"}\ \left(\Sigma \setminus \texttt{"} \right)^* \texttt{"}$$

A note to the `PRINTF` command: here use the DiMo tool the normal printf syntax from conventional programming languages, where each substring %i is replaced by the corresponding $\langle term \rangle$ value.

A note regarding precedences: the DiMo tool implements the standard precedence rules for Boolean operators, i.e. `A & B | C` is parsed in the same way as `(A & B) | C`.

The precedence and associativity rules for the arithmetic operators are as follows: Addition, subtraction, multiplication and division are left-associative; exponentiation is right-associative. This enumeration also reflects the increasing order of precedence amongst these operations.

C Version History

Version 0.3.0 (25/11/2022)

Feature changes:

- Add output control
- Change to ocam-sat-solvers version 0.7.1

Version 0.2.3 (21/06/2021)

Bug fixes:

- Fixed throwing of `exception Invalid`$_{a}rgument(\text{"}Bytes.create\text{"})oninstanceswithlong(ormany)parameternames.$

Version 0.2.2 (01/07/2020)

Feature changes:

- Output of Boolean constants has been changed to match input, now `True` and `False`
- The format of variables and formulas has been extended to allow quote symbols in the end as in `x'` etc.

Version 0.2.1 (29/10/2019)

Feature changes:

- DiMo has moved to GitHub under the BSD-3-Clauses license.

Version 0.2.0 (10/09/2019)

Bug fixes:

- The tokenizer did not recognise line breaks properly so that failure because of lexing errors resulted in error messages reporting the line and column number wrongly. Fixed.
- Applying the LOG function to argument 0 does not result in a stack overflow anymore. Instead the program terminates with failure.

Feature changes:

- The DiMo language has been extended with a function FLOG on terms that computes the rounded-off logarithm to base 2.
- The standard precedence and associativity rules for the arithmetic operators have been implemented. The DiMo language has been extended with a function FLOG on terms that computes the rounded-off logarithm to base 2.

Version 0.1.5 (03/07/2019)

Bug fixes:

- Operator MAX was interpreted as MIN: fixed.

Version 0.1.4 (01/07/2019)

Bug fixes:

- Command line parameters `--onlypos` and `--onlyneg` no longer ignored in checking validity problems.
- Problem with non-atomic terms like `i+1` in set definitions, leading to program crashes with error "`compare: invalid argument`", fixed.
- Problem with instantiation of formula schemes into negation normal form fixed.

Feature changes:

- Model enumeration problems now report the number of models that were found.

Version 0.1.2 (27/11/2018)

Bug fixes:

- Faulty instantiation of formulas containing negation, generalised junctions (`FORALL`, `FORSOME`) and propositional constants (`True`, `False`) fixed.
- Crash through array-out-of-bounds exception when specifying finite domains by enumeration of alle elements (e.g. `n: {1,2}`) fixed.

Feature changes:

- Output in debug mode streamlined.
- Added command-line parameters to suppress the printing of negative, resp. positive literals in models.

Version 0.1.1 (26/11/2018)

Bug fixes:

- Problem with producing output for validity problems trying to show refuting assignments for valid rather than invalid formulas fixed.
- Program crashes due to an error in function "toCNF.negate" caused by non-literal arguments fixed.