The LPL Modelling Language : Highlights

Hürlimann Tony
Department of Informatics
University of Fribourg, Switzerland
`www.virtual-optima.com`

September 12, 2003

Abstract

LPL is a structured mathematical and logical modelling and programming language with a powerful index mechanism, which allows one to build, maintain, modify, and document large linear, non-linear, and other mathematical models. It is also a modelling system with a point-and-click user interface to browse, edit and run models. The LPL includes some outstanding features: Integration of algorithmic and declarative knowledge, logical modelling, automatic generation of model documentation, and others.[1]

---

[1]This paper appears in: Kallrath J. (ed.), (2003), Modelling Languages in Mathematical Optimization, Kluwer Academic Publishers Boston/Dordrecht/London.

Contents

1 History

Since more than twenty years, several larger linear models (LPs and MIPs) are maintained and used at the Department of Informatics (formerly Institute of Automation and Operations Research) at the University of Fribourg. At the beginning of the eighties these models were stored and maintained in a MPSX-standard format. Ad-hoc tools were written to make specific modifications to the files. The first general and successful model manipulation tool on the IBM-PC was a generator written in dBASEII (from Ashton-Tate) that generated – from some constraint specifications – several database tables that represented the ROWS, COLUMNS, BOUNDS, and RHS section of a MPSX-file [Pasquier J., Hättenschwiler P. , Hürlimann T., Sudan B., 1986]. In 1986, we started a research project to store the models in an algebraic form, where the tables would be in a database and the mathematical structure in a text file. The idea was that in such a way we only needed to modify the data in the tables to generate a new model instance.

The result of my dissertation [Hürlimann T. 1987] was a small specialized algebraic language – called LPL (Linear Programming Language) – which was able to formulate all our models in a very readable and short way. The language had even some interesting features still not found in present languages: (1) It was possible to define hierarchical indexed sets, (2) One could automatically generate databases in dBASE format, the algebraic structure guided the generation, and (3) it had some limited way to generate the bipartite activity-constraint graph, an extremely valuable tool for certain models to browse and understand the mathematical structure of a model instance. Hierarchical sets was especially appealing since many constraints were organized in a way which was best viewed as trees: Constraints were generated recursively top down the tree. LPL was also the first language to integrate unit of measurements (see [Hürlimann T. 1991], for example).

With the termination of my dissertation the research project ended. But the program was used on a regular base at the Department and on need I added more and more features to the language, were the design principle always was (and still is): "Make it as simple as possible, but not simpler" (Einstein A.). (That means, I had always be very reluctant to all modeler requirements, because users tend to ask you to add thousands of features they would not use anymore.)

LPL always was meant to be a language to maintain our models at the Department. Although a free copy was put on the Web already in 1992, LPL was regarded as a productive internal product, and – more and more – as a playground for developing new ideas in modelling generation and manipulation: report generation, logical modelling, other graphical views of

a large model etc. The aspect of playground became more important when a new research project started in 1996, financed by the Swiss Federal Research Foundation. This project ended in 2001 and the result was basically, what LPL is today: A full-fetched declarative and algorithmic modelling language.[2]

Hence, LPL is an old language created in the eighties with the knowledge of the last twenty years in modelling language design, but it is a modern language and had changed considerably. The radical changes were possible basically due to the absence of an "LPL-community", although quite a lot downloaded it and apparently used it over the years. But nothing was undertaken to make it widely known. At the present time, LPL is firmly supported at the Department of Informatics as a permanent ongoing research project and LPL has also its own home page on the Web [Virtual-Optima].

2 Some Basic Ideas

The concept of *Modelling Language* has a standard meaning in the community of Operations Research were it comes from: an algebraic language. Although recent new languages add the paradigm of constraint programming, it still is meant basically as a *declarative* language, in contrast to an *algorithmic* language, under which paradigm most present *programming* languages could be subsumed.

Most programming languages could be classified into three paradigms: imperative, functional, and logic programming. Imperative languages (Pascal, C, Java) are closed related to the way a computer works: given a set of memory locations a program is a sequence of well defined sequence on retrieving, storing and transforming the content of these locations. A functional language (Lisp) implements Church's λ-calculus, where everything is considered as a function. In a logical language (Prolog), the computation can be viewed as a kind of (constructive) proof. Common to all three paradigms is that they implement a Turing machine, which allows them to implement any possible algorithm. In this sense they are all *algorithmic* languages, although the way they implement and execute a computation is very different from each other. Implementing a problem in such a language, means **to model** the way the problem must be proceeded. Hence, a (computer) program is a special *model*, a model which implements an algorithm.

---

[2]LPL started as a **L**inear **P**rogramming **L**anguage, but soon it became a **L**ogical **P**rogramming **L**anguage, because it is possible to use logical operators in constraints. A recent extension of automatic model documentation generation suggested still another meaning: **L**iterate **P**rogramming **L**anguage. But the true and hidden meaning of LPL is still: "**L**'art **P**our **L**'art".

However, modelling is a much broader idea than programming. We use the concept of *model* in a large number of contexts. We use modelling clay to form small replica of physical objects; children – and sometimes also adults – play with a model railway or model aeroplane; architects build model houses in order to show them to new clients; some people work as photo models, others take someone for a model, many would like to have a model friend. Models can be much smaller than the original (an orrery, a mechanical model of the solar system) or much bigger (Bohr's model of the atom).

But besides of theses less formal meanings, we also *model* a feasible (or sometimes infeasible!) space as a conjunction (or disjunction) of mathematical constraints. This is very different from a computer program: A mathematical formula models a space in a purely *declarative* way. There is no hint or instruction on how the model should be "processed". It is there just waiting that somebody extracts its hidden properties. While the run-time system of a compiled algorithmic program is eager to execute its instructions step by step, a declarative model waits to be transformed into another form. The model $\{x + y = 10, x - 2y = 4\}$, for example, could be transformed into the model $\{x = 8, y = 2\}$, which by the way is called its "solution" – but it is still the same model. No notion of "computation" or "processing" is implied when we have a mathematical model. We merely want to *represent* this model viewed as a two-dimensional x-y-diagram, for example, or to generate an explanation on its behavior (what-if-queries). These specific "transformations", of course, imply that an particular algorithm must be used.

Hence, the concept of modelling implies that the concept of modelling language must be viewed in a much broader context too. Most actual so called "modelling languages" are algebraic (or declarative) languages with scripting capability. MOSEL, AIMMS, and also LPL take a different approach: They try to integrate the two important paradigm into a single language: the algorithmic and declarative paradigm to create a new kind of language, which at the end is a generalized programming language, a true *modelling language*. The discussion here – seemingly on terms – is not merely theoretical. The reason why we would need languages containing both paradigms is real: Some problems are "best" formulated in an algorithmic way, others in purely declarative way and still others – and this is very important – as a blend of both. Below I give an example of such a problem.

All this means: Modelling must become a discipline which must follow certain criteria of quality like software engineering, the discipline of good programming. These criteria are *reliability* to ensure correctness and robustness of models, and *transparency* to enhance model extendibility, reusabil-

ity and compatibility. Reliability can be achieved by a unique notation, in which models are coded, by a formal specification of its semantics, and by introducing types, units and other checking mechanisms, in order to enforce domain consistencies. Transparency can be obtained by flexible decomposition techniques which fractionates the whole model into easily maintainable software modules and components. These are the requirements not only for the present programming languages, but also for the future modelling languages: (1) A unique notation for all kind of models; (2) Error checking mechanisms; (3) Decomposition techniques to break down a complex problem into manageable components. (For more arguments and information on these theoretical aspects see [Hürlimann T. 1999].)

3 LPL Highlights

The most difficult thing in designing a computer language is to decide what feature to add and what to leave out. LPL is no exception. Nothing is easier than add all kind of nice features needed once or a while, with the result of having a inconsistent and unmanageable giant language. The history of programming language is full of instructive anecdotes with fights and angriness, hopes and deceptions. The success of Wirth's Pascal in contrast to the goliath Algol68 was a striking event, and was a lesson I have learnt (hopefully). Creating a language in which all problems can be coded has a strange fascination. And what else do I propose when recommending a combined language of algorithmic and declarative knowledge? Yes, we are in quest of the language that allows to formulate all kind of models, but not in ten different ways!

   LPL has been designed in order to fulfill very different tasks in modelling which we judge to be important:

1. Orthogonality: Each element introduced into the language should have a general meaning if possible. For example: introducing the power operator in expression implies to use it in constraints too – which leads us to non-linear models; other example: introducing the logical OR operator in expression implies to use it again in constraints – which leads use to logical based models.

2. Unique notation: Every task should be written using a unique notation, for example: declarative and algorithmic knowledge formulation, data modelling and structure modelling.

3. Decomposition: It should be possible to break down a complex model into different pieces of code.

Using these design principles, LPL was implemented with the following features:

1. As a language which combines declarative and algorithmic knowledge likewise. One can formulate complete mathematical linear, non-linear and logical models in a purely declarative way. The data can be stored within the model or outside in databases. The interface to various solvers is configurable. It also allows the user to write algorithms. Define data structures and it has all control structures of another programming language. The language permits to break down the whole in logical modules – which are itself models or model-parts.

2. As a language that allows the user to define and manipulate the data. It can be used to generate a database (DDL: data definition language). Like a data manipulation language (DML) in database, LPL also can manipulate (join, select, project, etc.) multi-dimensional data (data cubes). The data are read from or written to databases or text files in various ways. Tools exist to generate various pivot tables from a multi-dimensional data cube.

3. As a modelling environment with an easy to use graphical interface. It allows the user to view and traverse the model in different ways and to find and modify all elements on a single mouse click.

4. As a model documentation tool: From an LPL model specification enriched with inline documentation, one can generate automatically a LaTeX-file or a PDF-file to generate a printable or publishable model documentation.

5. As a library ready to be use by other applications: LPL comes as a dynamic link library and its whole functionality can be used (and hidden from the user) in another application. One can create other modelling systems based on the library. LPL own model browser is only one that was put on top of the basic design.

6. As a client/server architecture: The same basic architecture of LPL can be used as a client/server application. Installing an LPL-Client on a machine and an LPL-Server on another, the two can communicate: The client can send the model to the server where the model can be manipulated, calculated or solved by a LPL-server in the same way as locally. It is completely transparent to the user (true ASP: Application Service Providing). On the server machine, one only needs to install a

WWW-Server, an FTP-Server, Tomcat (a free Java-based distributed software) and an LPL-Server.

Three of the mentioned features will be shown in more details in the next three sections: (1) combine algorithmic and declarative knowledge, (2) logical models, (3) documentation generation using a model. The complete specification of the LPL syntax and semantic can be found in the Reference Manual of LPL [Hürlimann T. 2003].

4   The Cutting Stock Problem

A paper machine produces a standard width of 100 inches. This width has to been cut into four widths: 97 rolls of width 45 inches, 610 rolls of width 36 inches, 395 rolls of width 31 inches, and 211 rolls of width 14 inches. The cut should be such that the total waste would be minimal, (for more information on the problem see [Chvatal V., 1983]).

From the model implementation point of view we can say that this problem can be solved efficiently using the following three modules: (1) We need a LP that formulates a cutting stock problem with a limited number of patterns. (2) We need an integer Knapsack problem, which decides which pattern must be entered the basic matrix of (1), in order to improve the objective. (3) We need an algorithm that solves these two declarative models an an iterated way, until some criteria are fulfilled. Hence, we can formulate this problem by composing it from three "pieces of knowledge": two of which are declarative representable and one that is a purely algorithmic piece.

In LPL the complete problem would be formulated as following:

```
MODEL CuttingStock "The Cutting Stock Problem";
 SET
   widths ALIAS w     "the different ordered smaller rolls";
   patterns ALIAS p   "possible cutting patterns";
 PARAMETER
   B                  "the width of the initial roll";
   width{w}           "the width of smaller roll w";
   demand{w}          "the demand for smaller roll w";
   a{w,p}  ::= IF(w=p,FLOOR(B/width)) "number of w rolls in p";
   du{w}              "cuts duals";

 MODEL cutting_stock ALIAS cs;
   VARIABLE X{p}      "number of initial rolls cut according to p";
     yy[1,1]; zz[1,1];  --asserts that the duals of 'cuts' are available
```

```
    CONSTRAINT cuts{w}: SUM{p} a*X + yy-zz >= demand;
    MINIMIZE obj: SUM{p} X;
  END

  MODEL find_pattern ALIAS fp;
    INTEGER VARIABLE y{w} [0,7]  "the number of rolls of size w in new patt";
    CONSTRAINT patt: SUM{w} width*y <= B;
    MAXIMIZE  cont: SUM{w} du*y;
  END

  MODEL DATA myData3 "Chvatal p.196";
  BEGIN
    w    := /1:4/;
    p    := /1:4/;
    B    := 100;
    width{w}  := [45  36  31  14];
    demand{w} := [97 610 395 211];
  END

BEGIN
  cutting_stock;
  du{w}:=cs.cuts.dual;
  find_pattern;
  WHILE (fp.cont > 1.0001) DO
    p:=ADDM('1');
    a{w,p}:= IF(p=#p,fp.y,a);
    cutting_stock;
    du{w}:=cs.cuts.dual;
    find_pattern;
  END;
  cutting_stock;
  WRITE "obj: ####.#\n####.# of pattern [ ## ]\n"
        : cs.obj, ROW{p|cs.X} (cs.X, COL{w} a);
END
```

The basic syntax of LPL is:

```
  MODEL ModelName;
    <declarative part of the model>
  [ BEGIN
    <algorithmic part of the model> ]
  END
```

Each model begins with the keyword `MODEL` and ends with the keyword `END`. In the declarative part the user declares the sets (`SET`), the parameters (`PARAMETER`), the variables (`VARIABLE`), the constraints (`CONSTRAINT`), and recursively other sub-models beginning with another `MODEL` keyword. The optional algorithmic part begins with `BEGIN` and contains a sequence of instructions.

Each declaration begins with a keyword `SET, PARAMETER` etc. Then comes a names and several attributes. For example the declaration of the set *widths* contains only one attribute: an alias, which is simply another (in general a shorter) name. LPL does not distinguish between index names and set names, so these aliases can be used as indexes in expressions. Parameters can be declared over sets. `a{w,p}`, for example, declares a matrix over the sets w and p. It also contains two other attributes: an expression and a comment. The expression is: `IF(w=p,FLOOR(B/width))`, which implement the assignment:

$$a_{w,p} = \begin{cases} \lceil B/width_w \rceil, & w = p \\ 0, & w \neq p \end{cases}$$

The indexes can be omitted if the context permits it. This makes the whole expression shorter and sometimes more readable. The second attribute is a comment in `"..."`. It allows the user to explain the object. This text is part of an automatically generated model documentation.

The next model part is the submodel `cutting_stock` which contains the declarations of three variables, a constraint, and a minimizing function. The submodel implements the following linear program:

$$\{\min \sum_p X_p \quad | \quad \sum_p a_{w,p} \cdot y_w \geq demand_w, x_p \geq 0, \quad \forall w\}$$

that is, minimize the waste (the number of pattern), subjected to the constraints that all demands must be fulfilled. The two singleton variables `y, z` are technical variables to assert that the reduced prices of the constraints exist. There values are fixed at 1. The attribute `[1,1]` says, that the lower and upper bounds of these variables (`y,z`) are 1. (The lower and upper bound could be arbitrary expressions.)

Next comes the submodel `find_pattern` which consists of the declarations of an integer variable, a constraint, and a maximization function. This submodel implements the following integer linear Knapsack problem:

$$\{\max \sum_w du_w \cdot y_w \quad | \quad \sum_w width_w \cdot y_w \leq B, \quad y_w \in \{0 \ldots 7\}\}$$

The last declaration is another submodel, in LPL a very special model: it is a `MODEL DATA`, where all data are collected. This model only has an algorithmic part and assigns the data defined in Chvatal's book. It defines four different widths and four different pattern, and assigns the values to the numerical vectors `width` and `demand`, as well as to the singleton parameter `B`.

This ends the declaration of the main model and an algorithmic part is started with the keyword `BEGIN`. This contains a sequence of instructions. The first instruction (`cutting_stock;`) says to proceed (actually solve, since it contains a minimizing function) the first submodel `cutting_stock`. The values of the dual cuts are now assigned to the vector `du`. All objects are accessible anywhere within the model using the dot-notation in LPL. The expression `cs.cuts.dual`, for example, refers to an object `cuts` within model `cs` (note that `cs` is an alias for `cutting_stock`). Hence `cs.cuts` refers to the constraints within the submodel `cutting_stock`. Each constraint contains several values, one of them are the reduced prices calculated by a simplex solver, and LPL refers them with the word `dual`. The third instruction is `find_pattern;`, hence the solving of the Knapsack problem.

As long as the objective value of the Knapsack model is larger than one, the instructions within the `WHILE` sequence are executed. That is: first a new element is added to the set `pattern` using LPL's function `ADDM()`. Then the values of the variables `y` of the Knapsack problem are assigned to the newly created column within the matrix `a`. Finally the two submodels are solved again.

At the end, the result is printed out with a `WRITE` instruction. The output of this `WRITE`-statement (which will not been explained here) is:

```
obj:  452.3
  48.5 of pattern [  2 0 0 0 ]
 100.7 of pattern [  0 2 0 0 ]
 105.5 of pattern [  0 2 0 2 ]
 197.5 of pattern [  0 1 2 0 ]
```

The model was decomposed into four model components, which follows the logic of the problem: two declarative submodels and an algorithmic part that guides their solution process. Furthermore, the data are clearly separated from the model specification and collected in another specialize submodel. To treat another model instance (with different data), one only need to exchange it with another `MODEL DATA` or to load another one.

5 Liquid Container

A company produces two liquids at unknown quantities x and y. The liquids are stored in two containers of capacity a and b. The two liquids can be stored in both containers, but cannot be mixed in the same container. Formulate the capacity constraint.

Three cases can be distinguishes: Either (1) we produce only the first liquid, or (2) only the second or (3) both. In the first case we have: $y = 0$ and $x \leq a + b$. In the second case we have: $x = 0$ and $y \leq a + b$. In the third case we have: $x \leq a$ and $y \leq b$. At least one of the three cases must be true. Hence, this constraint could be expressed as follows:

$$((y = 0) \wedge (x \leq a + b)) \vee ((x = 0) \wedge (y \leq a + b)) \vee ((x \leq a) \wedge (y \leq b))$$

.

It is easy to verify that this formula corresponds to the following shorter one:

$$((x > a) \rightarrow (y \leq 0)) \wedge ((y > b) \rightarrow (x \leq 0))$$

(We suppose that $x, y \geq 0$).

In LPL this could be coded as the following model:

```
MODEL Logic4 "Two Liquid Container";
  PARAMETER a:=10; b:=12;
  VARIABLE x [0,a+b]; y [0,a+b];
  CONSTRAINT capa: (x>a -> y<=0) AND (y>b -> x<=0);
  MAXIMIZE obj:x+y;
  WRITE x, y;
END
```

Hence, LPL allows one to mix logical and mathematical operators within a constraint. LPL automatically translates this specification into a linear MIP model – adding new 0-1 variables – which then can be solved by a standard MIP solver. Considerable efforts have been invested in this translation. It is difficult in the worst case to find the sharpest MIP for a given logical formulation. The number of variables or constraint may grow exponentially with the number of existing propositions or real variables in the logical formulation. To avoid this, LPL uses some heuristic to limit the generation, with the effect that the resulting MIP may be far from the sharpest formulation. Nevertheless, even in such a case, the translation of LPL can be useful, because a user can then manually manipulate and improve LPL's proposition and reformulate the problem on this base.

6 Model Documentation

Documenting a model is extremely important. However, this feature is still badly neglected by the model language designers as well as by the users. Documenting a model takes time and is the least rewarding job, because "the boss" is only interested in the results! The same is true in programming. The only language that supports now documentation extensively is *Java* with its tool, the *javadoc.* There exist tools created a long time ago by [Knuth D. E., 1992] that allow a programmer to generate a complex documentation of the source code, but it seems not to be used widely.

However, a well documented model is an invaluable investment (supposing that the model is not used just once). Many models take months to be built – and to document this process make it much more transparent and accessible to the decision makers. Model documentation can also be an excellent mean to prepare case-studies in learning modelling building. Building models is a creative process and must be learnt. One of the best way to learn it, is to study existing models. However, having only the simple mathematical formulation is sometimes not very helpful in understanding the formulas. Hence, a model enriched with explanation is much more useful. In the Internet, virtually thousands of mathematical models – in various language formulations – can be downloaded. How much more useful would they be, if an explanation text would accompany them!

Well then, let's code the model here and write a documentation apart from it. Why bother? Why integrate the documentation into the already heavily overloaded code? Because this is the only way to keep the code and the documentation consistent. How often the user is willing to write the documentation – at the end of the project, and then the model changes again... . Both should be done at the same time and in the same environment. Integrating the documentation into the model itself has another benefit: It can be regenerated (semi-)automatically at any time.

LPL includes a LaTeX generation tool. The mathematical and logical formulas are typeset with the free software TeX. Qualified comment and additional text as well as figures and graphics can be included into the formal model specification. From TeX then one can generate a PDF-file, HTML-files, or even XML-files using free software.

As an example, let's take a queen position problem on a chessboard. Suppose the problem is as following: How many queens are needed, and in what position, so that every unoccupied square of the chessboard is under direct attack from some queen? We introduce a binary variable X for each (i, j) position on the chessboard, which is 1 if the position is occupied by

some queen, otherwise it is 0. The objective function now is to minimize the number of queens:

$$\min \sum_{i,j} X_{i,j}$$

The constraint can be formulated as following: "For each (i,j) position on the chessboard one must require that it is under attack of at least one queen." "at least" in this context means OR. Hence, one can formulate this in LPL as following (where S is also an index over row/column like i or j):

```
CONSTRAINT
  ATT{i,j}: OR{S|S<>i and S-i+j>=1 and S-i+j<=#S} X[S,S-i+j]
    OR OR{S|S<>i and i+j-S>=1 and i+j-S<=#S} X[S,i+j-S]
    OR OR{S|S<>i} X[S,j] OR OR{S|S<>j} X[i,S];
```

The LaTeX generation tool of LPL translates this constraint automatically into the following LaTeX typeset formula:

$$ATT_{i,j} : \exists_{S|S\neq i \wedge S-i+j\geq 1 \wedge S-i+j\leq \#S} X_{S,S-i+j}$$
$$\vee \exists_{S|S\neq i \wedge i+j-S\geq 1 \wedge i+j-S\leq \#S} X_{S,i+j-S}$$
$$\vee \exists_{S|S\neq i} X_{S,j} \vee \exists_{S|S\neq j} X_{i,S} \qquad (?)$$

7   Conclusion

A short and necessary limited introduction to LPL was given in this paper. LPL is still an ongoing research project at the Department of Fribourg at the University of Fribourg and keeps it status as a playground for new and exciting ideas in modelling representation and manipulation. At the same time, there exists a stable version ready to be used in productive applications.

References

[Chvatal V., 1983] Chvatal V., (1983), Linear Programming, page 195ff.

[Pasquier J., Hättenschwiler P. , Hürlimann T., Sudan B., 1986] Pasquier J., Hättenschwiler P. , Hürlimann T., Sudan B., (1986), A Convenient Technique for Constructing your ouwn MPSX Generator Using dBASE II. Angewandte Informatik 28(7): 295–300.

[Hürlimann T. 1987] Hürlimann T., (1987), LPL: A structured Language for Modeling Linear Programs, Dissertation at the University of Fribourg, editor Peter Lang Verlag, Bern.

[Hürlimann T. 1991] Hürlimann T., (1991), Units in LPL, Institute for Automation and Operations Research, Working Paper No. 182, April, Fribourg.

[Hürlimann T. 1999] Hürlimann T., (1999), Mathematical Modeling and Optimization, An Essay for the Design of Computer-Based Modeling Tools, Kluwer Academic Publ., (Applied Optimization 31).

[Hürlimann T. 2003] Hürlimann T., (2003), Reference Manual for the LPL Modelling Language, (see: www.virtual-optima.com/download/manual.pdf).

[Knuth D. E., 1992] Knuth D. E., (1992), Literate Programming, CSLI (Center for the Study of Language and Information), Lecture Notes Number 27.

[Virtual-Optima] VO, `www.virtual-optima.com` , The home page of Virtual-Optima.