

*School of Computer Science and Software Engineering
Monash University*

Bachelor of Computer Science Honours
Clayton Campus

Research Proposal
2002

Compiling OPL to HAL

(Virginia) Lee Mei Leng

Supervisors: Maria Garcia de la Banda, Kim Marriott

Contents

1	Introduction	2
2	Research Context	3
2.1	Background	3
2.1.1	Mathematical Programming	3
2.1.2	Constraint Programming	3
2.2	OPL Optimization Programming Language	4
2.3	HAL (Constraint Logic Programming)	5
2.4	Compiler	6
3	Research Plan and Methods	7
3.1	Research Methods	7
3.2	Proposed Thesis Chapter Headings	10
3.3	Timetable	11
3.4	Special Facilities Required	12
4	Relevance of the Research	12

1 Introduction

Combinatorial problems such as timetabling, scheduling, planning, rostering and many others, are at the heart of organisations. This is because they typically aim at handling company resources, such as airplanes and staff, in the most efficient and cost effective way. Unfortunately, combinatorial problems are usually NP-complete, which means no known solution exists to solve them in polynomial time. Thus, as the problem gets more complex, the time taken to solve it grows exponentially.

Manually solving these kind of problems is often costly and time consuming. Therefore, a lot of effort has been devoted to address this issue computationally. This has resulted, among others, in two programming languages paradigms, namely mathematical programming [1] and constraint programming [2]. Both have enjoyed considerable success, this can be witnessed by the increasing attention from the commercial world, and the ever-growing interest and research effort from the research community. The best examples around are the success stories of AMPL [3], GAMS [4] and CPLEX [5].

Mathematical programming approaches problems by first modeling them as equations written in high-level set and algebraic notations. Solving the problems would then require solving the equations that represent the problems. In a way, mathematical programming specifies what the solution is without saying much about how to search for it. On the other hand, constraint programming provides a powerful language which allow the user to specify not only the problem but also the search procedures, i.e. how to explore the different alternatives that might lead to a solution. Mathematical programming and constraint programming have different aims, the former aims at finding an optimal solution and the latter aims at determining whether a solution, if any, exist.

In [6] Hooker summarized the similarity between mathematical programming and constraint programming, and suggested to merge both methods, given they have complimentary strengths. By merging both methods, the end product effectively bridges the gap between modeling and problem solving, thus making the new language more expressive and high level. For example, in the context of mathematical programming, the need to optimize a certain objective function could be delayed until it can be proved that a solution actually exists. The same principle applies to constraint programming: if one or more solutions are found, the best solution can be chosen based on the objective function concept. Furthermore, since both fields have evolved independently over the years, merging the two paradigms enables them to explore the different problem solving approach. Indirectly, merging the two paradigms opens up a whole new dimension on how to solve combinatorial problems. In summary, both methods are similar enough to make their combination possible, and yet different enough to make it profitable.

Two different approaches have been recently proposed for merging mathematical programming and constraint programming. The first approach, represented by Optimization Programming Language (OPL) [7], aims at creating a hybrid modeling language of both mathematical programming and constraint programming. The second approach, represented by Modeler++ [8], aims at extending the libraries of constraint programming language to handle the functionality of modeling language.

HAL is a new constraint logic programming language currently being developed at Monash University and Melbourne University, specifically designed to support the construction, extension and use of constraint solvers. The aim of this project is to extend HAL [9] to support

mathematical programming, thus making HAL more expressive. This will be done by first extending the HAL libraries to handle some mathematical constraints followed by implementing an OPL compiler for HAL. There are no previous attempts to compile OPL model file to equivalent HAL code.

2 Research Context

2.1 Background

2.1.1 Mathematical Programming

Mathematical programming has its roots in research on planning or scheduling conducted back in the 1940s. It assumed that all the restrictions on planning or scheduling could be represented mathematically as a set of equations or inequalities involving the variables. Mathematical programming has three main components; namely an objective function, a set of constraints and a set of assumptions. The objective is a function of the variables, such as cost or profit, which is used to select among possible solutions. Constraints affect the outcome of finding a solution: too few constraints might lead to too many solutions; while too many constraints might end up being too restrictive. Regarding the set of assumptions, the basic assumption in most cases is that the problem is linear. In mathematical terminology, the objective is a linear function, and the constraints are linear equations and inequalities. Such a problem is called a linear program, and the process of modeling such a problem and solving it is called linear programming. Linear programming is particularly important because a wide variety of combinatorial problems can be modeled as a linear program and solved efficiently.

The second major field in mathematical programming is integer programming, which is based on the assumption that problem variables can only take integer values. Conversely, linear programming can only deal with real numbers. Strickly speaking, integer programming is a refinement of linear programming, and results in problems that are usually harder to solve. Note that both methods rely on the assumptions that all programs are linear. If the assumptions break down, non-linear problems are solved using other algorithms and techniques. In general, mathematical programming is highly efficient for solving combinatorial problems.

2.1.2 Constraint Programming

Constraint programming arises from the search for a programming language that enables the programmer to state the relationship between objects not only as mathematical equations but as general constraints. A constraint represents a particular relationship between some user-defined variables that needs to be hold at all times; a solution therefore exists if it satisfies all the constraints.

A typical constraint programming problem is the map colouring problem. Consider the map of Australia, the aim is to colour different states with only 3 colours available (red, yellow, blue), subject to the conditions that no two adjacent states have the same colour. We use WA, NT, SA, Q, V and NSW to represents Western Australia, Northern Territory, South Australia, Queensland, Victoria and New South Wales respectively, and \neq represents inequalities. The following constraint captures that adjacent states may not be filled with the same colour.

WA != NT, WA != SA, NT != SA, NT != Q, SA != Q,
SA != NSW, SA != V, Q != NSW, NSW != v

There are many ways in constraint programming to solve the above problem. The simplest way is to start off by assigning a colour (e.g. red) to WA, then assign a different colour to NT and so on. Each colour assignment will be checked against to ensure the constraints are satisfied. If they are not, then the colour assignment is backtracked to choose another colour. If all available colours failed to satisfy the constraints when assigned to a given variable, then the colour assigned to a previous variables are backtracked and replaced by a new colour. This method is known as backtracking. The colour assignment process will continue until a solution is found or when the process exhaust all possible options, in this case no possible solution for the problem.

A more sophisticated way to solve the above problem is based in understanding the relationship between the constraints. For example, from above, we can deduce that WA , Q and V can share the same colour. If a colour (say red) is assigned to those 3 states, only 3 states remain with 2 colours available, namely yellow and blue. Again, both NT and NSW can be found to be equal, so a different colour (say green) is assigned to both states. This decision left us with one colour choice for SA, which is blue.

Solving this sort of problem in traditional programming languages is problematic and requires a lot of effort. Unlike their counterparts, constraint programming provides a natural way of representing the problem. Essentially, it abstracts away all the details of memory management and similar issues, leaving the programmer to concentrate on representing the problem in higher level.

2.2 OPL Optimization Programming Language

Optimization Programming Language is a modeling language designed specifically for optimizing combinatorial problems. The design of OPL was motivated by the successful implementation of mathematical modeling languages like AMPL and GAMS. Modeling languages like AMPL and GAMS allowed the user to represent problems traditionally expressed in algebraic notation by using equivalent computer terms. Like AMPL and GAMS, OPL provides full support for linear programming and integer programming, including access to linear programming algorithms. The main difference between OPL and other modeling languages is that OPL also provides support for both the modeling and search aspects of constraint programming. In other words, OPL allows the user to specify not only the problem but also the search procedure tailored to the problem. Besides that, OPL also offers some new concepts for scheduling and resource allocation problems, such as activities and resources. This high-level support adds a whole new dimension to the existing modeling language. Based on the idea of exploiting the features in both mathematical programming and constraint programming's approach in solving the problem at hand, OPL enables the user to model the problem in a hybrid approach. This, together with other new concepts, like higher-order constraint and logical combination greatly enhances the readability and expressiveness of the problem, abstracting away any unnecessary details. OPL effectively bridges the gap between modeling and problem solving.

Let us illustrate the capabilities of OPL with a simple example. Consider the following OPL program:

```

enum Products {gas, chloride}; (1)
enum Components {nitrogen, hydrogen, chlorine}; (2)
float+ demand[Products, Components] = [[1, 3, 0], [1, 4, 1]]; (3)
float+ profit[Products] = [40, 50]; (4)
float+ stock[Components] = [50, 180, 40]; (5)

var float+ production[Products]; (6)
maximize (7)
    sum(p in Products) profit[p] * production[p] (8)
subject to { (9)
    forall(c in Components) (10)
        sum(p in Products) demand[p, c] * production[p] <= stock[c] (11)
}; (12)

```

The first line (1) declares an enumerated set *Products* that represents the set of products in a company. Line (2) *Components* is the chemical components used in the production. Line (3) declares *demand* a 2 dimensional array whose element $display[p, c]$ represents the demand of product p for component c . Lines (4) and (5) introduce *profit* and *stock* as two arrays representing the profit of each product and the stock available for each component. Line (6) represents the optimal production for both gas and chloride. Lines (7) and (8) represent the objective function, which is to maximize the profits of total production. The objective function is subject to constraints as depicted in lines (9) to (12), whereby total demand cannot exceed existing stock.

OPL is currently available under ILOG's oplstudio implementation. This enables OPL to have full access to CPLEX [5] linear optimizers for solving mathematical programming problem, and is linked to ILOG Solver [10] for solving constraint programming problem.

2.3 HAL (Constraint Logic Programming)

HAL is a constraint logic programming language, mainly designed to allow the user to create, extend and experiment with different constraint solvers. It requires the user to provide enough information so that the types, modes and determinism of each procedure can be determined at compile time. This allows HAL to detect many programming errors at compile time. It also enables the compiler to generate highly efficient code.

Again, let us illustrate its capabilities by means of a simple example. Below is a sample HAL code for solving the classic CLP (Constraint Logic Programming) problem *mortgage* by modeling the relationship between P the principal or amount owed, T the number of periods in the mortgage, I the interest rate of the mortgage, R the repayment due each period of the mortgage and B the balance owing at the end.

```

:- module mortgage. (1)
:- import simplex. (2)
:- export pred mortgage(cfloat, cfloat, cfloat, cfloat, cfloat). (3)
:- mode mortgage(in, in, in, in, out) is semidet. (4)
:- mode mortgage(oo, oo, oo, oo, oo) is nondet. (5)

```

```

mortgage(P, T, I, R, B) :-                               (6)
    T = 0.0,                                           (7)
    B = P.                                             (8)
mortgage(P, T, I, R, B) :-                               (9)
    T >= 1.0,                                         (10)
    NP = P + P * I - R,                               (11)
    mortgage(NP, T-1.0, I, R, B).                     (12)

```

The first line (1) states that this is the definition of the module `mortgage`. Line (2) imports a module defined earlier called `simplex`. This provides a simplex-based linear arithmetic constraint solver for constrained floats, called `cfloats`. Line (3) declares that this module exports the predicate `mortgage` which takes five `cfloats` as arguments. This is the *type* declaration for `mortgage`. Lines (4) and (5) are examples of mode of usage declarations. Since there are two declarations, `mortgage` has two possible modes of usage. In the first, the first four arguments have an **in** mode meaning their values are fixed when the predicate is called (they are input), and the last has a mode **out** which means it is uninitialized when called, and fixed on the return from the call to `mortgage` (it is output). Line (5) gives another mode for the `mortgage` where each argument has mode `oo` meaning that each argument takes a “constrained” variable and returns a “constrained” variable. This more flexible mode allows arbitrary uses of the `mortgage` predicate (those uses in which the variable do not have a given value), but will be less efficient to execute. Line (4) also states that for this mode `mortgage` is `semidet`, meaning that it either fails or succeeds once. For example `mortgage(0.0,-1.0,0.0,0.0,B)` fails since `T` is negative. The rest of the file contains the standard two rules defining `mortgage`. The first states that when the number of repayments is 0, then the balance is simply the principal. The second states that if the number of repayments is greater than one, then we make one repayment, compute the new principle `NP` and take out a mortgage with this new principle for one less time period.

2.4 Compiler

A compiler for language `L` reads a program written in language `L` and produces a new program in some other lower level language. In our case, an `OPL` file (called the model) will be translated to `HAL`. Each model file is divided into 4 sections, namely declarations, instructions, search procedures and display instructions, with the last section being optional. In the declaration section, all the variables used in the model file are declared. Instructions are mainly used in the essence of mathematical programming to solve a certain problem, or to maximize or minimize a certain objective. As for search procedures, these are typically found in constraint programming, where by a preferred search method is provided along side to assist the search of solution. All these different structures will be analysed carefully to ensure the appropriate representation in `HAL`.

There are 8 phases in a compiler, but not all phases are applicable in this context, since both source language (`OPL`) and target language (`HAL`) share some similarities. Nevertheless, a lexer and a parser are still required in our compiler. Lexer stands for lexical analyser, it reads in input from a file or standard input and use regular expression to break the input into tokens like, identifier, reserved word, arithmetic operators and comments. Regular expressions are the standard way of specifying a certain input pattern, the pattern matching process is performed based on the expression. For example, `[0-9][a-zA-Z]*` will match any string that has a digit as

the first character followed by zero or more occurrences of uppercase or lowercase alphabetic characters. A parser checks all the tokens received from the lexer to make sure it matches the grammar of the language. For instance, below is the grammar for a simple calculator:

```
expr = expr + term
expr = expr - term
expr = term

term = term * factor
term = term / factor
term = factor

factor = int
```

The above grammar ensures multiplication and division is performed before addition and subtraction, if both multiplication and division occurs on the same line, it will be performed from left to right. So $3 + 4 * 2$ will gives the answer 11, while $3 * 5 / 2$ will gives the answer 7 (integer division).

3 Research Plan and Methods

3.1 Research Methods

The aim of this research is two fold, to develop an OPL compiler for HAL and to extend HAL libraries. The first aim is based on the ideas from OPL, a compiler capable of mimicking the ability of hybridizing both mathematical programming and constraint programming. The second aim comes from the Modeler++ approach which extends the library of C++ to handle constraint programming capabilities. Both methods have their merits and faults, the former will be more user friendly since the translation process is automated. But it is also more rigid, since the user has no control over the generation of target code. The latter is harder to use for certain users, since it requires the user to posses some constraint programming basics. But the latter provides more flexibility, enabling the user to customize how the program behave. In this research, our initial attempt will be to implement the compiler. Meanwhile, HAL's library will be extended as needed. Both processes are inter-related, this can be witnessed by the need to translate reflective functions in OPL.

In the figure 1, the arrow labeled 1 represents the first approach, in which the user writes a program in OPL. The second arrow labeled 2 represents the second approach, in which the user write a program in the extended HAL. The figure suggests that both approaches move in different directions, with OPL as a transformation of a more powerful language, and Modeler++ as an extension of the existing library to reach the user. The truth is that at some point both approaches are bound to meet. This research will attempt both approaches, and the meeting point will be where the optimum solution is found.

This project will involve the development of an OPL compiler for the HAL programming language. In doing this, I will have to learn at least two different languages: a modeling language

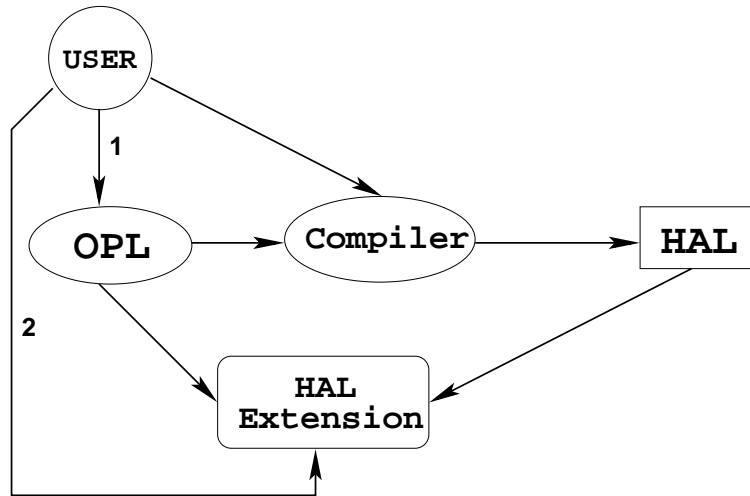


Figure 1: OPL to HAL

(OPL) and a constraint logic language (HAL). Learning these languages at the same time is no easy task, especially when they belong to different programming language paradigms. With no background in these fields, this will be a great challenge for me.

The first step in constructing a compiler is to parse OPL code correctly, this requires the compiler to understand the syntax of OPL. An incomplete version of OPL syntax can be obtained from [7], but without the complete syntax the parse tree can not be generated. There are two ways of obtaining the complete syntax. One is to study the sample OPL code and try to generate the syntax from there. Another is to contact Pascal Van Hentenryck (creator of OPL) to obtain the complete syntax. Once the complete syntax is obtained, the first phase of parser construction will be completed. To make sure the parser works correctly, a few sample OPL files will be parsed through. That includes both valid and invalid OPL model files. The next step will be to generate the parse tree.

How to produce the parse tree depends on the structure of the source code and the target code. If both source code and target code have the same code structure, the translation is straightforward. Otherwise, the translation process must analyse the source code in order to obtain enough information to generate the target code. In the case of OPL and HAL, both languages have similar structures especially if the HAL libraries are extended to support some of the mathematical constructors in OPL. Since this will be the case, we believe line by line translation will usually suffice.

The OPL compiler will be written in Mercury. Mercury is a logic programming language designed for the construction of large, reliable and efficient software. As in HAL, it requires the user to provide enough information so that the types, modes and determinism of each procedure can be determined at compile time. Again, this allows Mercury to detect many programming errors at compile time. It also enables the compiler to generate highly efficient code. The are two main reasons for choosing Mercury, first, it produces very reliable and efficient code. And second, HAL and Mercury are very similar. In fact, HAL compiles to Mercury.

For compiler construction, two separate groups of lexical analyser generator and parser generator have been evaluated. Flex and bison are well documented tools with plenty of examples,

they are stable and fast, and they can be accessed by Mercury through Mercury's C interface. Unfortunately, they do not support constraint logic programming basics like relations and constraints naturally. Thus, due to performance concerns and possible integration problems, lex and moose were chosen. Lex and moose are the lexical analyser generator and parser generator provided alongside Mercury. Lex will be used to generate the lexical analyser, which tokenises the source code and breaks it into identified tokens. Moose will be used to generate the parser, receiving tokens from the lexical analyser and parse it through the language syntax, at the same time generating the equivalent HAL code. Lex and moose have some advantages over flex and bison, due to their simplicity, ease of use and natural support for Mercury. Besides, it is easier to generate target code since Mercury and HAL have some common grounds, integration is much easier and natural too. The downside of using lex and moose is that both are relatively immature, and the error handling features in moose still needs to be improved. Another problem that might occur in the later stage of the development phase is code restructuring due to possible changes in Mercury.

Another issue that we have to consider is how to represent OPL data structures in HAL. For example, OPL has the ability to index the array using arbitrary finite sets, array itself can be of arbitrary number of dimensions. Finite sets can possibly involves complex data structures like records and sets. At this stage, it is still unclear how array indexed by arbitrary finite sets can be represented in HAL. It posed as a problem to be studied carefully. This problem is even greater when it comes to representing some novel concepts of scheduling in OPL such as origin, horizon, activities and different types of resources. All these are new concepts that needs to be modelled correctly in HAL, the challenge is to find the right method of representation.

Another issue is the generation of efficient target code, the target code is generated based on the assumption that the source code is efficient. Therefore, no optimizations will be performed; in other words, a naive compilation approach is adopted here. In the later stage of research, within our limited time frame and resources, certain procedures in OPL might be analysed to allow some optimisations, the focus is on common procedures like forall, foreach. The method presented in [11] to compile such structures into the logic language ECLiPSe [12] might be a good starting point for this.

Reflective functions are used in OPL to obtain information about the current state of the computation, (e.g. the domain of a variable at a certain stage in the search procedure). For instance, dmin is a reflective function that takes an integer variable and returns the minimum value in that variable domain. To translate the code correctly, the same function needs to exist in HAL. Only one approach can solve this problem, extend HAL's library. This idea came from the Modeler++[8] approach.

In summary, the final product, the OPL to HAL compiler will have the following features:

- Parse OPL code correctly
- Translating OPL data structures into equivalent HAL code
- Extend HAL libraries to cover two major components
 1. data e.g. sets, activities
 2. procedure e.g. reflective functions, foreach, forall iteration
- Generate efficient and optimized target code

3.2 Proposed Thesis Chapter Headings

1. Introduction
2. Background information
 - (a) OPL
 - (b) Mercury
 - (c) HAL
 - (d) Compiling OPL
3. Related work
4. Implementation
 - (a) Overview
 - (b) Lexical Analyser and Parser (lex and moose)
 - (c) Representing OPL data structures in HAL
 - (d) Type and mode consistency
 - (e) Loop and iteration translation
 - (f) Search procedures translation
 - (g) Effective code generation
 - (h) Evaluation
5. Future work
6. Conclusions
7. Bibliography

3.3 Timetable

Week	Activity
18/3	Background reading
25/3	Research OPL and Constraint Programming
1/4	Research Mercury and HAL
8/4	Research compiler construction and begin writing research proposal
15/4	Research and learn lex and moose, construct simple compiler
22/4	Draft of research proposal
29/4	Finalized research proposal, learn HAL
6/5	Research method for representing OPL data structures
13/5	Further reading
20/5	Test and verify the correctness of OPL syntax
27/5	Literature review draft and representing complex OPL data structures
3/6	Prepare for interim presentation
10/6	Finalized literature review
17/6	
24/6	Generation of parse tree
1/7	Research method for effective representation of search procedure in HAL
8/7	Develop code for representing search procedure in HAL
15/7	
22/7	Extend code to handle type checking
29/7	Extend HAL's library to include reflective functions
5/8	Generate help error messages
12/8	
19/8	Research code optimisation technique
26/8	Develop code optimisation technique for search procedure
2/9	
9/9	Finalize coding
16/9	
23/9	Work on first thesis draft
30/9	
7/10	Prepare for final presentation
14/10	Work on second thesis draft
21/10	Practice final presentation
28/10	Finish thesis
4/11	Complete project web site

3.4 Special Facilities Required

The facilities offered to Honours students at Monash University are sufficient to complete the research.

4 Relevance of the Research

Solving problems with the assistance of computer often requires modeling problem in computer terms. This requires the modeler to capture the essence of the problem while abstracts away any unnecessary details. This task is complicated by the fact that most existing modeling language requires the modeler to possess some sort of programming background. Unfortunately, people that are familiar with a certain problem domain do not tend to have the programming background. This is the case with most people involved in solving combinatorial problem. These people tend to have a mathematical background but lack of sophisticated computer skills. Even though the use of high level abstraction and resemblance to natural language has eased the use of computer language. The gap between the problem and its model is still too big in most modeling language. Our thesis will aim to extend the HAL libraries to fill the gap between modeling and problem solving, thus making the modeling task easier.

References

- [1] Irvin Lustig and Jean-Francois Puget. Program != program: Constraint programming and its relationship to mathematical programming. white paper of ilog inc., Mountain View, CA 94043, USA, 1999.
- [2] Kim Marriott and Peter J. Stuckey. *Programming with Constraint: An Introduction*. The MIT Press, Cambridge, Massachusetts, 1998.
- [3] Robert Fourer, David M. Gay, and Brian W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA, 1998.
- [4] J. Bisschop and A. Meeraus. *On the Development of a General Algebraic Modeling System in a Strategic Planning Environment*. Mathematical Programming Study, 1982.
- [5] ILOG team. Ilog cplex 7.0 reference manual. Technical report, ILOG, Gentilly, France, 2000.
- [6] John N. Hooker. Logic, optimization, and constraint programming. Technical report, Graduate School of Industrial Administration, Carnegie Mellon University, April 2000.
- [7] Pascal Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, Massachusetts, 1999.
- [8] Laurent Michel and Pascal Van Hentenryck. Modeler++ a modeling layer for constraint programming libraries. Technical report cs-00-07, Brown University, December 2000.
- [9] *An Overview of HAL*. Proceedings of Principles and Practice of Constraint Programming, October 1999.

- [10] ILOG team. Ilog solver 4.4 users manual. Technical report, ILOG, Gentilly, France, 1999.
- [11] Joachim Schimpf. Logical loops. Technical report, IC-Parc, Imperial College, London SW7 2AZ United Kingdom.
- [12] ECLiPSe team. Eclipse user manual version 4.0. Technical report, IC-Parc, Imperial College, London, July 1998.