



Mosel: An Overview

Last update: 5 July, 2007

Mosel: An Overview

Y. Colombani and S. Heipcke

Dash Optimization, Blisworth House, Blisworth, Northants NN7 3BX, U.K.

Yves.Colombani@dashoptimization.com,

Susanne.Heipcke@dashoptimization.com

May 2002, last rev. January 2007

**[これは、皆様のご便宜のための参考翻訳です。
誤訳、不適切な訳があるかもしれませんので、
原文と併読なさるようになさってください。]**

アブストラクト

このペーパーは、標準のマトリクススペースのソルバーとの、モデル作成と解の報告のインタフェースとして、このソフトウェアを使用するのに必要な Mosel 言語の基礎を紹介します。これをさらに、一歩前進させて、より複雑なソリューション・アルゴリズムを実行するのに、どのような Mosel が使用できるかも紹介します。

Mosel ライブラリを使用して、Mosel 言語で書かれているモデルは、C、C++、Java、C#、Visual Basic などのプログラミング言語で書かれているアプリケーション・プログラムと統合でき、また、これらのアプリケーションからアクセスできます。

モデル作成とそれを解くための Mosel 環境はオープンで、モジュール構造になっていますが、この環境は、容易に拡張することができるように設計されており、特定のタイプの問題やソルバーに制限されていません。このペーパーでは、ユーザが、既存の Mosel 言語を拡張し、例えば、他のソルバーにアクセスするための、新しい機能性を提供させるようにするには、どのようにしたらよいかについても説明します。

キーワード: モデル作成言語、プログラミング言語、複数のソルバー

目次

1	イントロダクション	4
1.1	ソルバー・モジュール	4
1.2	その他のモジュール	5
1.3	ユーザ・モジュール	5
1.4	I/Oドライバー	6
1.5	このペーパーの内容	6
2	オーバービュー	7
2.1	Mosel言語 - 一つの小さな例	7
2.2	上の例の拡張	8
3	Moselの言語	9
3.1	タイプとデータ構造	9
3.2	データ、データファイルへのアクセスのイニシアライゼーション	11
3.3	言語構成	14
3.3.1	Selections	14
3.4	ループ	15
3.5	例：集合 (set) を使う	16
3.6	サブルーチン	17
4	Moselライブラリ	18
5	モジュール	19
5.1	利用可能なモジュール	20
5.2	<i>mmxp</i> : MIPのためのvariable fixing heuristic	23
5.3	<i>mmquad</i> : QPを定義し、解く	26
6	パッケージ	27
6.1	新しいサブルーチンを定義する	28
7	ユーザのモジュールを書く	29
7.1	新しいサブルーチンを定義する	30
7.2	新しいタイプを創る	32
7.2.1	モジュールのコンテキスト	34
7.2.2	タイプの生成と削除	35
7.2.3	ストリングへの、および、ストリングからのタイプの変換	36
7.2.4	演算子のOverloading	37
8	むすび	40
	Bibliography	41

1 イントロダクション

Mosel は、問題をモデル化して解くための環境で、ライブラリ形式、もしくは、スタンドアロンプログラムとして提供されます。Mosel は、一つの言語ですが、この言語は、モデル作成言語とプログラミング言語の機能を兼ね備えており、これにより、これらの2つの概念の長所が結合されている言語です。問題を「モデル作成言語」を使って表現し、アルゴリズム操作が「スクリプト言語」で書かれている AMPL[Fourer et al., 1993]のような「伝統的な」モデル作成環境(同様に、OPL[Van Hentenryck, 1998] の場合は、OPL-スクリプト)とは対照的に、Mosel では、(例えば、決定変数を宣言するとか、制約式を表現するような)モデル作成ステートメントと、実際に問題を解く手順(例えば、最適化コマンドのコール)との間には、まったく、境目がありません。

この相乗作用のおかげで、ユーザは、モデル作成のためのステートメントとモデルを解くためのステートメントを織り交ぜ、組み合わせることで、複雑なソリューション・アルゴリズムをプログラムできます。

1.1 ソルバー・モジュール

さまざまな問題は、それぞれ、それ自身の特定のタイプの変数と制約式をもち、したがって、たった1種類のソルバーが、すべての問題で効率的であるはずがありません。これを考慮に入れ、Mosel では、デフォルトとして、どんなソルバーとも結合せず、モジュールとして提供されている外部のソルバーとの、ダイナミック・インタフェースを提供しています。それぞれのソルバーモジュールは、それぞれ、それ自身の、一組の手順とファンクションを持ち、それらは、直接的に、Mosel 言語のボキャブラリーを広げ、したがって、言語としてのケイバビリティを拡張できます。このアーキテクチャーにより、Mosel と使用されるソルバーとの効率的なリンクが保証されます。例えば、MPL [Maximal, 2001]のような、他のシステムでも、似たようなコネクションが見られますが、モジュールの概念により、Mosel は、特定のタイプのソルバーに制限されることなく、したがって、ユーザは、ユーザレベルで、個々のソルバーの使い方の細目を決めることができます。例えば、ある LP ソルバーでは、手順 'setcoeff(ctr, var, coeff)' を使い、制約式 'ctr' の変数 'var' のマトリクス係数をセットする必要があるとしても、そのような手順は、他のソルバーでは、何の意味もなさないかもしれません。同様に、Xpress-Optimizer のような、それが適切なソルバーである場合、モジュール *mmquad* は意味があり、二次式で言語のシンタックスを拡張します。

モジュール・アーキテクチャーの主要な利点は、新しいソリューション・テクノロジーが使えるようになったとき、その技術へのアクセスを行うために、Mosel を変更する必要がまったくないということです。

Mosel の、現在利用可能なソルバー・モジュールはいくつかあり、*mmxprs* は、Xpress-Optimizer により、線形問題、混合整数問題を解くためのアクセスを、*mmxslp* は、SLP (Sequential Linear Programming) により、非線形問題を定義し、解くためのアクセスを、*mmstsp* は、Stochastic

Programming問題を解くためのアクセスを、そして、*kalis*は、Constraint Programming問題を解くためのアクセスを、それぞれ、提供しています。

1.2 その他のモジュール

また、Mosel のモジュールのアーキテクチャーは、ソルバー以外のソフトウェアへの環境を開く手段として使うこともできるのも注目してください。例えば、*mmodbc*という Mosel のモジュールで、ユーザは、標準の SQL コマンドを使って、ODBC インタフェースを定義するデータベースとスプレッドシートにアクセスできます。他のライブラリを書き、特定のデータベースとコミュニケーションするのに必要な機能性を得ることもできます(ODBC に加えて、*mmodbc* は、既に、Microsoft Excel への、ソフトウェア特有のインタフェースも提供しています)。

別の Mosel モジュール(*mmjobs*)により、複数のモデルを扱うことができ、それには、メモリの中で、同期を取り、データを交換するためのメカニズムも含んでおり、これにより、例えば、Mosel を使い、広範囲な(分解)アルゴリズムによるインプリメンテーションが行えます。Dash Whitepaper "Multiple models and parallel solving with Mosel"には、このようなアルゴリズムのいくつかの例が説明されています。

モジュール *mmive* を使うと、ユーザは、Xpress-IVE のグラフィカル環境で、自分自身のグラフィックスを作成できます。さらに、モジュール *mmxad* (Xpress Application Developer XAD)により、ユーザは、Mosel モデルの中で、完全なグラフィカル・アプリケーションを定義することも出来ます。

モジュールを書き、ジェネリック・インタフェースを持たない他のアプリケーションとコミュニケーションするのに必要な機能を得ることもできます。

1.3 ユーザ・モジュール

提供されているモジュールに加え、Mosel は、ユーザによるどんな種類の追加にもオープンです。Mosel と、そのモジュールとのコミュニケーションは、特定のプロトコルとライブラリを使用します。このネイティブなインターフェースはパブリックであり、したがって、ユーザに、自分自身のモジュールを実行するのを妨げません。C プログラム言語の形式で書かれたものなら、どのようなプログラミング・タスクも、Mosel 言語から使用できるモジュールに変換できます。

下記は、ユーザによって書かれるモジュールの用途の例ですが、もちろん、これらの用途に限られてはいません。

- アプリケーション固有のデータハンドリング(例えば、複合データ・タイプの定義、メモリ内でのデータ入力、出力)
- 外部プログラムへのアクセス

- モデルから、プログラミング言語で実行されたソリューションアルゴリズム、ヒューリスティクスにアクセスする(多分、既存コードの再使用する)
- (例えば、ヒューリスティクスから頻繁にコールする必要があるソートアルゴリズムのような)特定の、そして、時間が重要なプログラミング・タスクを効率的に実行するためのアクセス

1.4 I/O ドライバー

最近、Mosel の「ファイル」の概念は、一般化されました。そして、現在では、例えば、下記のようなものも、「ファイル」です。

- 物理的なファイル(テキストやバイナリー)
- メモリの 1 ブロック
- オペレーティングシステムで提供されるファイル・ディスクリプタ
- ファンクション(コールバック)
- データベース

現在、配布されている Mosel は、一組の I/O ドライバーを持っており、それらにより、特定のデータソースへのインターフェースが提供されたり、Mosel ライブラリを動かしているアプリケーションと Mosel モデルとの間の情報交換を、非常に直接的な方法で行うことを可能にしたりすること出来ます。また、Mosel のネイティブの Interface(NI)を使い、ユーザは、自分自身のドライバーをインプリメントできます。I/O ドライバーの使い方は、Mosel のかなり高度な使い方なので、この機能は、"Generalized file handling in Mosel"と題する、別の Dash Whitepaper で説明されています。

1.5 このペーパーの内容

このペーパーの最初の部分は、システムの一般的なアーキテクチャーの概要を与えて、小さい例を使って、この言語を紹介します。Mosel 言語の持つ特徴の特定の部分については、第 2 セクションで、より詳細に議論しますが多くのプログラムからの抜粋がそれに続きます。次のセクションでは、Mosel のモジュールの概念について説明して、現在、商業的に利用可能なモジュールを紹介します。それに次いで、もっと高度なソリューション・アルゴリズムで、どのように Mosel を使ったらよいかを示します。ソフトウェアを統合することを目的とすると、プログラミング言語から、Mosel 言語で定義されたオブジェクトにアクセスする方法を知ることは重要です。これは次のセクションの話題です。最後のセクションでは、ユーザが、どのようにして、Mosel のネイティブ・インターフェースを通して、自分自身のモジュールをインプリメントできるかについて説明します。

2 オーバービュー

2.1 Mosel 言語 - 一つの小さな例

下記は、2つの資源制約式の下で、2つ製品を、導入、展開のように生産したら利益が最大になるか、という問題です。ここで、xs は小さい製品の数、xl は大きい製品の数を示す変数です。ここで、この生産計画を、Mosel でどのように、簡単な混合整数計画(MIP)問題として定式化して、解くかを示します。

一般構造: Mosel プログラムは、すべて、model というキーワードで始まり、次に、名前が続き、end-model で終わります。オブジェクトは、それらが、アサインメントにより、明白に定義されている場合を除き (例えば、 $i := 1$ は、 i を整数として定義し、それに 1 という値を割り当てる)、すべて、declarations section でデクレアされなければなりません。モデルには、このような、いくつかの declarations section が、複数の場所にあります。この例の場合は、2 の変数 xs と変数 xl とを、タイプ mpvar であると定義します。次いで、モデルは、2 つの変数からなる、2 つの一次不等式の制約式を定義し、変数 xs と変数 xl が整数値だけを取るよう制約します。

```

-----
model Chess
  uses "mmxprs"                                ! Use Xpress-Optimizer for solving
  declarations
    xs,xl: mpvar                                ! Decision variables
  end-declarations
  3*xs + 2*xl <= 160                            ! Constraint: limit on working hours
  xs + 3*xl <= 200                             ! Constraint: raw material availability
  xs is_integer; xl is_integer                 ! Integrality constraints
  maximize(5*xs + 20*xl)                       ! Objective: maximize the total profit
  writeln("Solution: ", getobjval)             ! Print objective function value
  writeln("small: ", getsol(xs))              ! Print solution values for variables xs
  writeln("large: ", getsol(xl))              ! and xl
end-model
-----

```

問題を解く: 手順 maximize により、一次式 $5*xs+20*xl$ を最大にするために、Xpress-Optimizer をコールします。Mosel には「デフォルト」ソルバーがないので、プログラムの最初で、ステートメント"mmxprs"により、Xpress-Optimizer を使うことを指定します。

解をプリントする: 最後の3行のステートメントにより、最適解での目的関数の値と、2つの変数の最適解での値が印刷されます。 getsol(xs) の代わりに、「dot notation style」である xs.sol

を使用することもできます。

ラインの分割: (`xs is_integer; xl is_integer` のように)、セミコロンを挿入することで、いくつかのステートメントを、1 行に書くことができることに注意してください。これに反して、"line end" や continuation を示す特別なキャラクタ がないので、いくつかの行にまたがるステートメントは、すべての行をオペレータ (+, >= など) で終わらせ、ステートメントが完了していないことを明確にしなければなりません。

コメント: 例で示されているように、Moselでは、コメントは、その前に、`!` を置いてから書きます。複数の行をまたがるコメントは、`!` で始まり、`!` で終わります。

2.2 上の例の拡張

Mosel の表現力を示すために、前のモデルをエンハンスしたバージョンを以下に示します。

制約式に名前を付ける: この例でも、制約式に名前をつけますが、ここでは、もはや、一次式を使わず、ソルバーをコールするとき、そのリファレンスを使います。二つの不等式のリファレンスは、例えば、これらの制約式の解の情報 (dual, slack, activity) へのアクセスに使います。

データ構造: この2番目の例では、データ構造セットと配列も導入されています。ここでは、配列 `DescrV` と、そのインデックスセット `Allvars` の両方がダイナミックです。なぜなら、これらが生成される時、そのサイズとコンテンツが決められていないからです。これらのコンテンツは、`declarations section` に続く `assignments` で定義されます。

```
-----
model Chess2
  uses "mmxprs"
  declarations
    Allvars: set of mpvar           ! Set of variables
    DescrV: array(Allvars) of string ! Descriptions of variables
    xs, xl: mpvar
  end-declarations
  DescrV(xs):= "Small"           ! Define descriptions for variables
  DescrV(xl):= "Large"
  Profit:= 5*xs + 20*xl          ! Name the objective function
  Time:= 3*xs + 2*xl <= 160
  Wood:= xs + 3*xl <= 200
  xs is_integer; xl is_integer
  maximize(Profit)
```

```
writeln("Solution: ", getobjval)

forall(x in Allvars)                ! Print the solutions of all variables
    writeln(DescrV(x), ": ", getsol(x))    ! for which a description is defined
end-model
```

Figure 1 は、Xpress-IVE の開発環境で、このモデルを実行した結果を示しています。

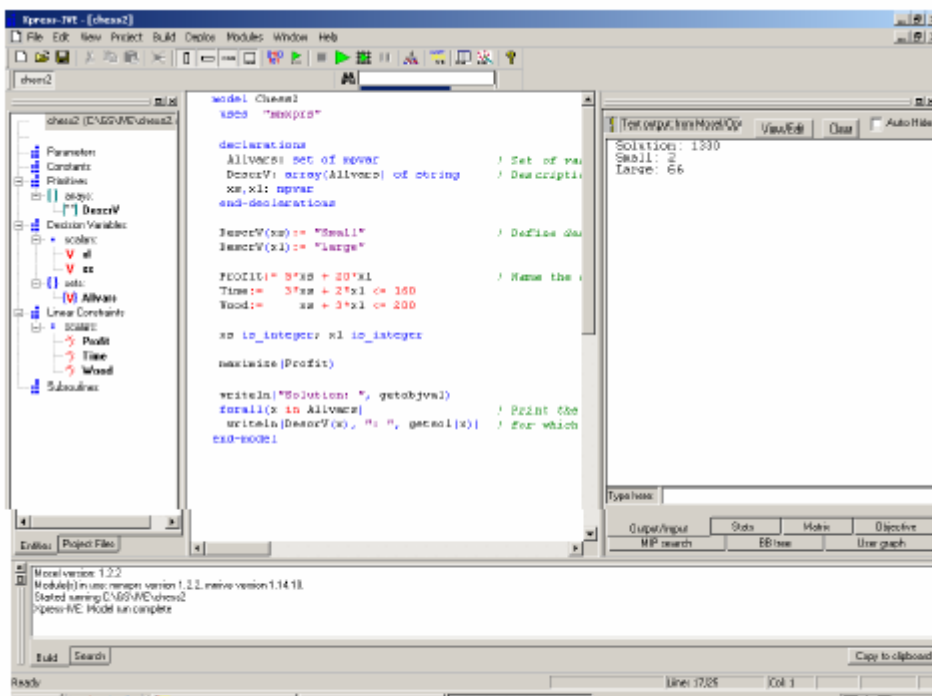


Figure 1: IVE display after model execution

3 Mosel の言語

3.1 タイプとデータ構造

Mosel は、他のプログラミング言語でもそうであるように、下記のような基本タイプを提供します。

- integer
- real (double precision floating point numbers)
- boolean (symbols true and false)
- string (single character and any text).

数値計画法を対象とした、MP タイプの mpvar (decision variables)、および、linctr (線形制約式) と共に、これらにより、Mosel の基本タイプが形成されています。

基本のタイプに加え、Mosel は、下記の定義を行います。

structured types set	(collection of elements of a given type)
array	(collection of labeled objects of a given type)
list	(ordered collection of elements of a given type)
record	(collection of objects of any type)

前のセクションで触れたように、Mosel のデータ構造(set と array)は、それらが生成されたときに、サイズ、コンテンツが未知の場合、ダイナミックです。下記の定義は、constant sets (R1, S1)、list s(L1)、static arrays (A1, A2)をもたらします。

```
-----
declarations
  R1 = 3..5
  S1 = {"red", "green", "blue"}
  A1: array (S1) of real
  A2: array(R1, -2..1) of mpvar
  L1 = [1, 2, 3, 4, 5]
end-declarations
```

Dynamic sets (R2, S2)、lists (L2, L3)、arrays (A3, A4) は、以下のように定義とともに生成されま

```
-----
declarations
  S2: set of string
  A3: array(S2,S2) of linctr
  A4: array(R2:range) of boolean
  L2: list of real
  L3: array(set of integer) of list of string
end-declarations
```

上の二つの例では、special type of set:が導入されます。ここで、R1 と R2 はレンジ(すなわち、ordered sets of integers) です。3..5 は、R1 が、3~5までのすべての整数のセットであることを示しています。

レコードの定義には、declarations ブロックといくつかの類似性があります。まず、キーワード

record から始まり、それに、フィールド名とタイプのリストが続き、キーワード end-record で定義の終了を示します。record の定義は、declarations ブロックになければなりません。下記の code extract により、3 つのフィールド ('Source', 'Sink', and 'Cost') を持ったレコードが定義され、ネットワークの arc を表すのに使用されます。

```

-----
declarations
  arc = record
    Source, Sink: string           ! Source and sink of arc
    Cost: real                     ! Cost coefficient
  end-record
  OneArc: arc
  ARCS: array(ARCSET: range) of arc
end-declarations
OneArc.Source:= "A"; OneArc.Sink:= "B"
ARCS(1).Cost:= 5.1

```

ユーザは、Mosel 言語の事前に定義されたタイプと同様に扱われる新しいタイプを定義することができます。上の record 'arc' の命名は、「ユーザが定義するタイプ」の record の例です。

3.2 データ、データファイルへのアクセスのイニシアライゼーション

データ構造には、Mosel モデルの中で、直接、値を割り当てるか、外部のファイルから読まれる値でイニシアライズします。下記は、Mosel モデルでの、値の割り当ての例です。

```

-----
declarations
  V: real
  A1: array(3..5) of real
  S: set of string
  L: list of integer
  A2: array(range,range) of boolean
  AL: array(set of string) of list of real
end-declarations
V := 0.5                               ! Assign a value to a scalar
A1 :: [1.5, 2.3, 4.5]                   ! Initialize an array with known index range
A1(3) := 1.8                            ! (Re)Assign a single array entry

```

```

S := {"A", "BC", "DEF"}           ! Assign a set
L := [1, 2, 3, 3, 4, 5, 4]       ! Assign a list
A2 :: (1..2 0..2)[true, false, false, true, true, false] ! Initialize a 2-dim. array
AL :: ({"A", "B"})[[3, -6, 1.5], [2, 8.4]] ! Initialize an array of lists
AL("C") := [1, 2.5, 4, -0.5]     ! Assign an array entry

```

データがファイルから読まれるとき、この例のように、たいいていの場合、ダイナミックなデータ構造が使用されます。この例は、Mosel での initializations section の使い方を例示したものです。

```

declarations
  A: array(1..6) of real           ! Static array definition
  S: set of string                 ! Dynamic set
  C: array(S) of real              ! Dynamic array
  L: array(set of integer) of list of string ! Dynamic array of lists
  R: array(range) of arc           ! Dynamic array of records
end-declarations
initializations from "initdata.dat"
  A C S L R
end-initializations
writeln("I = ", I, "¥nA = ", A, "¥nC = ", C)
writeln("S = ", S, "¥nL = ", L, "¥nR = ", R)

```

このプログラムで読まれるデータファイル `initdata.dat` は、以下のコンテンツを持っています。

```

I: 10
A: [2 4 6 8]
C: [{"red"} 3 {"green"} 4.5 {"blue"} 6 {"yellow"} 2.1]
S: ["white" "black"]
L: [(1) [{"a" "b"}] (3) [{"c" "d" "e" "f"}] (6) [{"i" "i" "i"}]]
R: [(1) [{"A" "B" 1.2}] (2) [{"A" "C" 4.5}] (3) [{"B" "C" 3}]]

```

static array A にたいしては、インデックスは既知なので、値は、単純なリストとして与えられま

す。dynamic array には、データファイルには、インデックスも含まれている必要があります。このデータファイルにより、このプログラムは、以下のアウトプットを生み出します。

```
-----
I = 10
A = [2,4,6,8,0,0]
C = [('red',3),('green',4.5),('blue',6),('yellow',2.1)]
S = {'red','green','blue','yellow','white','black'}
L = [(1,['a','b']), (3,['c','d','e','f']), (6,['i','i','i'])]
R = [(1,[Source='A' Sink='B' Cost=1.2]), (2,[Source='A' Sink='C' Cost=4.5]),
(3,[Source='B' Sink='C' Cost=3])]
-----
```

static array A では、すべてのエントリーが定義されています。したがって、インデックスを指定する必要はまったくありません。dynamic array には、n-tuple のリストが印刷されますが、ここで、最初の n-1 要素がインデックスであり、最後の要素が array entry の値です。

柔軟性を増すために、手順 read/readIn や手順 write/ writeIn を使い、テキストファイルから読み込んだり、書きだしたりすることも可能です。下記のコンテンツを持つデータファイル readdata.dat を読むためには、Mosel プログラムで、下のように入ります。

```
-----
# Data:
A(1) = 5.2
A(2) = 3.4
-----
declarations
  j: integer
  B: array(range) of real
end-declarations
fopen("readdata.dat", F_INPUT)
fskipline("#") ! Skip lines starting with a '#'
repeat
  readln('A('j,',') =',B(j))           ! Read the indices and the value on a line
until (getparam("nbread")<4)         ! until a line is not properly constructed
fclose(F_INPUT)
-----
```

3.3 言語構成

また、データタイプに加え、他のプログラミング言語と同様、Mosel も、典型的なフローコントロール (selections and loops) を提供しています。

3.3.1 Selections

selection のもっとも簡単なフォームは if-then ステートメントです。これは拡張すれば、if-thenelse とか、if-then-elif-then-else というステートメントも可能で、二つの条件を連続してテストし、下記の例のように、二つの条件を両方とも満たさない場合は、別の alternative を実行します。

```
-----  
declarations  
  A : integer  
  x: mpvar  
end-declarations  
if A >= 20 then  
  x <= 7  
elif A <= 10 then  
  x >= 35  
else  
  x = 0  
end-if  
-----
```

もし、Aの値が 20 に等しいか、もしくは、大きい場合は、変数 x には、upper bound 7 が適用されます。そして、もし、A の値が 10 に等しいか、もしくは、小さい場合には、変数 x には、lower bound 35 が適用されます。他の場合(すなわち、A は、10 より大きくて、20 より小さい場合は)、すべて、変数 x の値は、0 にされます。

いくつかの相互排他的な条件(ここでは、A の値)がテストされるときは、下のように、case ステートメントを使用すべきです。

```
-----  
declarations  
-----
```

```

A : integer
x: mpvar
end-declarations
case A of
  -MAX_INT..10 : x >= 35
  20..MAX_INT : x <= 7
  12, 15 : x = 1
  else x = 0
end-if

```

3.4 ループ

ループは、繰り返すことが必要なアクションを一つにまとめてグループ化し、それを、インデックスのすべての値や、カウンタ(forall)に従って、繰り返したり、ある条件が満たされているか、否か (while, repeat-until)によって繰り返したりするのに使います。Moselのforallループ、whileループには2つのバージョンが存在します。そのうちのinlineバージョンは、下のように、一つのステートメント上でループを行うためのものです。

```

declarations
  x: array(1..10) of mpvar
end-declarations
forall(i in 1..10) x(i) is_binary

```

そして、2番目のバージョンはforall-do (while-do)で、これにより、1つのブロック・ステートメントを囲い、最後は、end-doで閉じます。

```

declarations
  x: array(1..10) of mpvar
end-declarations
forall(i in 1..10) do
  x(i) is_integer
  x(i) <= 100
end-do

```

```
end-do
```

3.5 例：集合 (set) を使う

以下の例は、set の操作を紹介し、いろいろなネスト(入れ子に) されたループの使い方を示します。このプログラムは、"Sieve of Eratosthenes(エラステネスのふるい)"を使い、2 から、ある所与の上限までの間の素数の集合を計算します。ここでは、見つけられるあらゆる素数の倍数のすべての数が、残っている数の集合から削除されます。

```
model Prime
  parameters
    LIMIT=100                !Search for prime numbers in 2..LIMIT
  end-parameters
  declarations
    SNumbers: set of integer  ! Set of numbers to be checked
    SPrime: set of integer    ! Set of prime numbers
  end-declarations
  SNumbers:={2..LIMIT}
  writeln("Prime numbers between 2 and ", LIMIT, ":")
  n:=2
  repeat
    while (not(n in SNumbers)) n+=1
    SPrime += {n}
    i:=n
    while (i<=LIMIT) do
      SNumbers-= {i}
      i+=n
    end-do
  until SNumbers={}
  writeln(SPrime)
end-model
```

集合 (set) オペレータ: オペレータ `+=` と `-=` を使って、集合に部分集合を加えたり、取り除いたりすることができます。(注意: インデックスセットとして、一度、使用されると、集合が縮小しないかもしれないことに注意してください。) また、Mosel は、集合の標準オペレーションを定義しています: union、intersection、difference (operators `+`、`*`、`-`)。

ラン・タイム・パラメータ: この例は、parameters section を紹介しています。このセクションで定義された定数の値は、モデルの実行のときにリセットすることが出来ます。そうでなければ、与えられたデフォルト値が使用されます。ここで、プログラムを動かす人が、set of number の上限を選べるようにしています。もう一つの典型的な使い方は、パラメタの形で、データファイルの名前を指定することです。

3.6 サブルーチン

Mosel は、一組の、あらかじめ定義されたサブルーチン (例えば、`write / writeln`、また、`cos`、`exp`、`ln` のような関数、また、`getsol`、`reverse`、`getsize` のようなモデルのオブジェクトにアクセスするためのサブルーチン) を提供しています。しかし、個々の特定のプログラムのニーズに従って、新しい手順やファンクションを定義することも出来ます。Mosel での、ユーザ定義のサブルーチンは、それぞれ、`procedure / end-procedure`、`function / end-function` でマークされなければなりません。関数から戻される値は、下の例に示されているように、`returned` に割り当てられなければなりません。パラメタをサブルーチンにパスすることも可能です。サブルーチンの名前に続いて、パラメタ、パラメタのリストが括弧の中に加えられます。

```

model "Simple subroutines"
  function timestwo(b:integer):integer
    returned := 2*b
  end-function
  procedure printstart
    writeln("The program starts here.")
  end-procedure
  printstart
  a:=3
  writeln("a = ", a)
  a:=timestwo(a)
  writeln("a = ", a)
end-model

```

サブルーチンの構造は、モデルのそれと似ています。サブルーチンは、対応するサブルーチンだけで有効なローカルのパラメタを宣言するため、declarations section を持てます。サブルーチンのコールもネストできますし、繰り返し、コールできます。

Forward declaration: Mosel は、キーワード forward を使うことで、ユーザは、サブルーチンを、定義とは分離して declare できます。

Mosel では、個々のバージョンが、異なる数、異なるパラメタタイプを持っていれば、サブルーチンの名前を再使用できます。この機能は、一般に、overloading と呼ばれます。ユーザは、Mosel、および、ユーザ自身が定義するファンクション、手順にたいして、サブルーチンの overloaded version を、追加して定義できます。

1

4 Mosel ライブラリ

言語で書かれているモデルには、Mosel の C インタフェースを通して、C からアクセスできます (他に利用可能な他のインタフェース: Java, C#, Visual Basic)。このインタフェースは、2 つのライブラリの形で提供されています。これは、Mosel で書かれたモデルやソリューション・アルゴリズムを、C 言語で書かれた既存のアルゴリズムの一部を (再) 利用して、何らかの大きいシステムに統合したり、Mosel を他のソフトウェアに統合したりする場合に興味深いことでしょう。

BIM ファイル (portable Binary Model file) にモデルファイルをコンパイルする場合、Mosel Model Compiler ライブラリを使用される必要があります。この BIM ファイルは、モデルを実行するため、Mosel Run Time ライブラリで、入力されます。

Mosel ライブラリを使用すると、モデルをコンパイルしてランするだけでなく、別のモデリング・オブジェクトの情報へのアクセスもできます。以下の例は、セクション 3.5 示された、モデル Prime を、どのようにコンパイルして、パラメタ LIMIT に異なる値を与えて実行して、結果として得られる素数の集合を印刷するかを示しています。 (この例は、モデル Prime2 で実際に動きましたが、出力の印刷は、C で行ったため、ありません。)

得たい結果 (2 と 500 の間の素数) を内容とする集合 SPrime の中味を印刷するには、ファンクション XPRMfindident を使い、Mosel reference をこのオブジェクトに取り出すことが必要です。こうして、この集合の要素を一覧表にして、それらのそれぞれの値を得ることができます。

```
-----
#include <stdio.h>
#include "xprm_mc.h"
#include "xprm_rt.h"
```

```

int main()
{
    XPRMmodel mod;
    XPRMalltypes rvalue, setitem;
    XPRMset set;
    int result, i, size, first, last;
    XPRMinit();
    XPRMcompmod(NULL, "prime2.mos", NULL, NULL); /* Compile model Prime2*/
    mod=XPRMloadmod("prime2.bim", NULL);          /* Load the BIM file */
    XPRMrunmod(mod, &result, "LIMIT=500");       /* Run the model */
    XPRMfindident(mod, "SPrime", &rvalue);       /* Get the object 'SPrime' */
    set = rvalue.set;
    size = XPRMgetsetsize(set);                  /* Get the size of the set */
    if(size>0)
    {
        first = XPRMgetfirstsetndx(set);         /* Get number of the first index */
        last = XPRMgetlastsetndx(set);          /* Get number of the last index */
        printf("Prime numbers from 2 to 500:¥n");
        for(i=first;i<=last;i++)                /* Print all set elements */
            printf(" %d, ", XPRMgetsetval(set,i,&setitem)->integer);
        printf("¥n");
    }
    XPRMfinish();
    return 0;
}

```

5 モジュール

Moselのオリジナルな特徴は、モジュール(Cプログラミング言語で書かれているダイナミックなライブラリで、Mosel Native Interface が設定するコンベンションを遵守する)を使って言語を拡張できることです。モジュールにより、下記のことからに関して、Mosel 言語を拡張します。

-
- 新しい constant symbols
 - 新しい subroutines
 - 新しい types

- 新しい I/O drivers
- 新しい control parameters

このリストで最も重要な項目は、サブルーチンとタイプはです。モジュールで定義されたサブルーチンは、完全に新しい function か手順、もしくは、Mosel の overload existing subroutine です。例えば、モジュールは、C か C++のライブラリ function の形で、容易に利用可能なアルゴリズムや solution heuristic を呼ぶサブルーチンかもしれません。

モジュールで定義された新しいタイプは、(integeror、mpvar のような) Mosel 自身のタイプのよう扱われます。これらは、(arrays、 sets などのような)複雑なデータ構造で使用されたり、initializations にあるファイルから読み込まれたり、または、サブルーチンのパラメタとして登場することができます。Mosel のオペレータは、新しいタイプと共に機能するために overload できます。finite domain constraint solvers などのソルバーをサポートするために、新しいタイプの定義が必要になるかもしれません。

皆様にお届けする Mosel は、一組の I/O ドライバーを持っています。これらのドライバーにより、(ODBC などの)特定のデータソースへのインタフェースを行ったり、Mosel ライブラリ、Mosel モデルをランしているアプリケーションの間の情報交換を行ったりすることが出来ますが、その方法は非常に直接的な方法で、これにより、圧縮されたり、暗号化されたりされているファイルの read/write を渡す際に、ユーザに様々な可能性を提供することができます。

モジュールで発行される定数、コントロール・パラメータは、それ自体では、ほとんど意味がありません。それらは、それらのタイプ、サブルーチンに関連して使用されるのが普通です。

ソリューション・アルゴリズムのプロトタイプは、いろいろなことを研究して、ソリューション戦略やソルバーを組み合わせて行いますが、モジュールは、ソリューションアルゴリズムの急速なプロトタイプのための適切な手段と考えられます。例えば、mmxprs と、Kalis のような finite domain constraint solver へのアクセスを提供するモジュールを使用すると、branching ツリーのあらゆるノードで、constraint propagation アルゴリズムを使い、subproblem を解き、得られた解によって、MIP 問題のカットを生成するような MIP search を定義することが出来ます(インプリメンテーションの例については、Dash Whitepaper "Hybrid MIP/CP solving with Xpress-Optimizer and Xpress-Kalis"を見てください)。

5.1 利用可能なモジュール

現時点では、下記のモジュールが利用可能です。

Solvers: mmxprs、mmquad、mmxslp、mmsp、Kalis

Data handling: mmodbc, mmetc
System: mmsystem
Graphics: mmive, mmxad

これまでの例で、モジュール *mmxprs* は、既に、Xpress-Optimizer の問題を解くのに使用されました。Mosel 言語からアクセスできる形で、問題を解くための基本的なタスクとアルゴリズムの設定を行うことに加え、このモジュールのおもしろい特徴は、下に示すプログラムの一部に示されているような、Mosel から、裏に潜んだ solver C ライブラリのコールバック・ファンクションを定義できることです。以下の例は、整数ソリューションが見つけられたときは、いつも、コールされ、その時点でのソリューションを印刷するためのファンクションを定義します。

```

uses "mmxprs"
declarations
  x: array(1..10) of mpvar
end-declarations
public procedure printsol
  writeln("Solution:", getsol(Objective))
  forall(i in 1..10) write("x(",i,")=", getsol(x(i)), "¥t")
  writeln
end-procedure
setcallback(XPRS_CB_INTSOL, "printsol")

```

モジュール *mmquad* により、二次計画問題を定式化して、解くことができます(セクション 5.3 の例を見てください)。

モジュール *mmxslp* では、非線形問題を定式化して、解くことができます。

最近、追加されたのは、モジュール *mmsp* で、これにより、Mosel の中で Stochastic Programming がサポートされるようになりました。

モジュール *kalis* により、Artelys の Constraint Programming solver Kalis にアクセスできます。

モジュールで、データファイルへの追加的なインタフェースを定義できます。mmetc は、手順 *diskdata* を定義し、これにより、mp-model の「data in-and-output がエミュレート(特定のハードウェア向けに開発されたソフトウェアを別の設計のハードウェア上で実行させること)できます [Dash, 1999]。モジュール *mmodbc* は、Mosel の *initializations* ブロック、もしくは、より大きい柔軟

性を得るためには、標準の SQL コマンドを使い、ODBC インタフェースを持つどんなデータソースへのアクセスも可能にします。また、このモジュールで、Excel への software-specific なインタフェースを定義することもできます。以下のプログラムの抜粋の例では、ODBC コネクションを通して、MS-Excel のスプレッドシートから 2 次元の array とそのサイズを読み込みます。データベースなどの別のデータソースにスイッチするには、単にファイル名を変えるだけで済むことに注意してください。

```

model sizes
  uses "mmodbc"
  declarations
    Nprod, Nrm: integer
  end-declarations
  initializations from 'mmodbc.odbc:ssxmpl.xls'
    Nprod Nrm
  end-initializations
  declarations
    PneedsR: array(1..Nprod,1..Nrm) of real
  end-declarations
  initializations from 'mmodbc.odbc:ssxmpl.xls'
    PneedsR as 'USAGE'
  end-initializations
end-model

```

モジュール *mmsystem* は、`gettime` やファイル・ハンドリングのようなファンクションを提供します。これは、Mosel 言語の中から、オペレーティングシステムのコマンドを使用するようなことまで可能にします。もっとも、これは、明らかに、モデルファイルのポータビリティの犠牲のもとです。

モジュール *mmive* により、ユーザは、グラフィカルユーザーインターフェース Xpress-IVE を使って、Figure 2 の図などのような、ユーザ自身のグラフを描きながら、Mosel モデルで作業を行うことが出来ます。

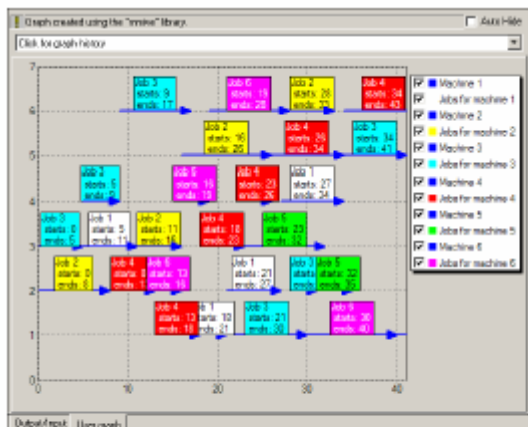


Figure 2: Gantt chart drawn by mmive

Xpress Application Designer (モジュール *mmxad*) により、ユーザは、Mosel を使って、例えば、Figure 3 に示されている人員計画作成アプリケーションなどのような、完全にグラフィカルなアプリケーションを作成することができます。

5.2 *mmxp* : MIP のための variable fixing heuristic

このセクションでは、Xpress-Optimizer (モジュール *mmxprs*) を使って、混合整数計画 (MIP) 問題を解くためのソリューション・ヒューリスティクスの例を示します。このソルバー・モジュールで Mosel 言語に提供される、すべてのサブルーチン、タイプ、定数、および、コントロール・パラメータは、太字でハイライトされます。

この問題のインプリメンテーションを構造化を助けるため、問題の定式化、および、ソリューション・アルゴリズムは、いくつかのサブルーチンに分割されるだけでなく、main model file の異なるファイルに収容されます。

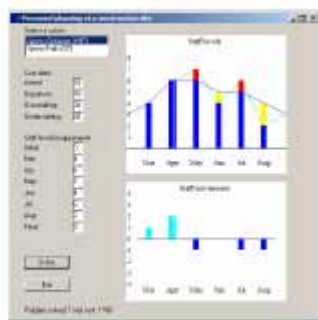


Figure 3: XAD application: planning the personnel at a construction site

```

model "Fixing binary variables"
  uses "mmxprs"
  include "fixbv_pb.mos"
  include "fixbv_solve.mos"
  solution:=solve
  writeln("The objective value is: ", solution)
end-model

```

下記は、ファイル fixbv_pb.mos に含まれているモデル定義の抜粋です。

```

declarations
  RF=1..2 ! Range of factories (f)
  RT=1..4 ! Range of time periods (t)
  (...)
  open: array(RF,RT) of mpar
end-declarations
(...)
forall(f in RF,t in RT) open(f,t) is_binary

```

このモデルは、オープン binary 変数を持っており、これらの binary 変数にたいして、以下のステップからなる variable fixing heuristic が実行されます。

Solve the LP problem.

Fix the binary variables that are almost 0 or 1 at these values (“rounding”).

Solve the resulting MIP problem and retrieve the solution value.

Restore the original MIP problem and solve it using the solution value of the modified problem as bound (“cutoff” value).

この solution heuristic を実行するファンクション solve は、ファイル fixbv_solv.mos で定義されます。

```

function solve:real

```

```

declarations
  TOL=5.0E-4
  osol: array(RF,RT) of real
  bas: basis
end-declarations
setparam("zerotol", TOL) ! Set Mosel comparison tolerance
setparam("XPRS_CUTSTRATEGY", 0)
setparam("XPRS_HEURSTRATEGY", 0)
setparam("XPRS_PRESOLVE", 0)
maximize(XPRS_TOP, MaxProfit) ! Solve the LP problem
savebasis(bas) ! Save the current basis
forall(f in RF, t in RT) do ! "Round" binaries
  osol(f,t):= getsol(open(f,t))
  if osol(f,t) = 0 then
    setub(open(f,t), 0)
  elif osol(f,t) = 1 then
    setlb(open(f,t), 1)
  end-if
end-do
maximize(MaxProfit) ! Solve the modified MIP
ifgsol:=false
if getprobstat=XPRS_OPT then ! If an integer feas. solution was found
  ifgsol:=true
  solval:=getobjval ! Get the value of the best solution
end-if
forall(f in RF, t in RT) ! Restore the original problem
  if ((osol(f,t) = 0) or (osol(f,t) = 1)) then
    setlb(open(f,t), 0); setub(open(f,t), 1)
  end-if
loadbasis(bas) ! Reload the basis
if ifgsol then ! Set the "cutoff" to the
  setparam("XPRS_MIPABSCUTOFF", solval) ! best known solution
end-if
maximize(MaxProfit) ! Solve the original MIP
returned:=if(getprobstat=XPRS_OPT,getobjval,solval)
end-function

```

5.3 *mmquad* : QP を定義し、解く

前に述べたように、QP 問題を定式化し、解くには、モジュール *mmquad* を使います。このモジュールは、新しいタイプ *qexp* を定義し、これは、Xpress-MP QP ソルバーによってインプットとして受け入れられます。これは、モジュール *mmquad* によって提供される Mosel 言語への拡張は、モジュール間のコミュニケーションによって、他のモジュールでも使えることを意味します。

以下は、資産額についての upper bounds と選択できる値の数の制限のもとで、総費用を最小にするポートフォリオの構成を決定したい、という例です。考慮の対象となっている資産の間の依存関係により、費用関数は二次関数になります。

```

model Portfolio
  uses "mmxprs", "mmquad"
  parameters
    DATAFILE = "portf.dat"           ! Name of the data file
    LIMIT = 20                         ! Maximum number to be chosen
  end-parameters
  declarations
    NVAL = 30                          ! Total number of assets
    RV = 1..NVAL
    LCOST: array(RV) of real           ! Coeff. of linear part of the obj.
    QCOST: array(RV,RV) of real       ! Coeff. of quadratic part of the obj.
    UBND: array(RV) of real           ! Upper bound values
    n: integer                          ! Counter for chosen assets
    x: array(RV) of mpvar              ! Amount taken into the portfolio
    y: array(RV) of mpvar              ! 1 if asset i is chosen, else 0
    Cost: qexp                        ! Objective function
  end-declarations
  initializations from DATAFILE
    UBND LCOST QCOST
  end-initializations
  Cost:= sum(i in RV) ( LCOST(i)*x(i) +           ! Define the (quadratic) cost function
    QCOST(i,i)*x(i)^2 +
    sum(j in i+1..NVAL) QCOST(i,j)*x(i)*x(j) )
  sum(i in RV) x(i) = 100                ! Amounts chosen must add up to 100%

```

```

sum(i in RV) y(i) <= LIMIT ! Limit on total number of values
forall(i in RV) do
    x(i) <= UBND(i)*y(i) ! Upper limits          ! Upper limits
    y(i) is_binary ! Variables are binary        ! Variables are binary
end-do
minimize(Cost) ! Minimize the total cost
writeln("Solution: ", getobjval) ! Solution printing
writeln("quadratic part: ",
    getsol(sum(i in RV) ( QCOST(i,i)*x(i)2 +
    sum(j in i+1..NVAL) QCOST(i,j)*x(i)*x(j)), ") )
forall(i in RV)
    if(getsol(y(i)) > 0.000001) then
        writeln(i, ": ", getsol(x(i)))
        n+=1
    end-if
writeln("%n", n, " assets have been selected")
end-model

```

6 パッケージ

Mosel 言語は、ユーザによって書かれたライブラリによる拡張にオープンです。これらのライブラリは、下記の2つの形を取ります。

- パッケージ – Mosel 言語で書かれているライブラリで、Mosel 言語のために新しい定数、サブルーチン、および、タイプを定義する。
- モジュール – C プログラミング言語で書かれているダイナミック・ライブラリ (Dynamic Shared Object, DSO)。

このセクションの残りで、ユーザパッケージの例について議論します。次のセクションで、ユーザモジュールの2つの例を説明します。

パッケージの構造は、model に似ていて、キーワード model を package に置き換えます。

パッケージは、module と同じように、uses ステートメントと共に、モデルに含められます。include ステートメントと共にモデルに組み込まれる Mosel コードとは異なって、packages は別個にコンパイルされます。つまり、これらの内容は、ユーザには見えないということです。

パッケージの典型的な使い方には、下記のようなものがあります。

- 個人用のツールボックスを作る。
- Mosel で書かれた、コンテンツを明らかにしないで分配したいモデルの部分(例えば、再定式化)とかアルゴリズム
- モジュールへのアドオンで、Mosel 言語で、より容易に書かれるもの

6.1 新しいサブルーチンを定義する

問題を解いた後に、ユーザは、ソリューションを取り出して、それを格納しておきたい、とう場合があります。Mosel には、ソリューションの値にひとつひとつにアクセスするためのファンクション `getsol` があります。しかし、一度に、すべての値にアクセスして、ソリューションを決定変数の array にセーブするサブルーチンがありません。しかし、このようなサブルーチンは、モジュールという形式で、容易に実行できます。前のセクションの例では、以下のようにして、ソリューションをプリントできます。

```

-----
model Portfolio
  uses "solarray", "mmxprs"
  ...                               ! Data initialization
  declarations
  x: array(RV) of mpvar             ! Amount taken into the portfolio
  sol: array(RV) of real            ! Solution values
  end-declarations
  ...                               ! Formulate and solve the problem
  solarray(x,sol) ! Retrieve the solution for all variables
  writeln(sol)                    ! Print the solution
end-model
-----

```

以下の Mosel コードは、前に議論したモジュール 'solarray' の場所で使えるパッケージ 'solarray' を生成します。モジュールが、array にたいして、どのようなタイプの、そして、どんな数のインデックスセットにも対処できる、一つの C ファンクションを定義するのにたいして、ここでは、使いたい array index sets の構成 (configuration) ごとに、明示的に、サブルーチン `solarray` の、

ひとつの overloaded version を定義する必要があります。

```

package solarraypkg
  public procedure solarray(x:array(R:set of integer) of mpvar,
    s:array(set of integer) of real)
    forall(i in R) s(i):=getsol(x(i))
  end-procedure
  public procedure solarray(x:array(R1:set of integer,
    R2:set of integer) of mpvar,
    s:array(set of integer,
    set of integer) of real)
    forall(i in R1, j in R2) s(i,j):=getsol(x(i,j))
  end-procedure
  public procedure solarray(x:array(R1:set of integer,
    R2:set of integer,
    R3:set of integer) of mpvar,
    s:array(set of integer,
    set of integer,
    set of integer) of real)
    forall(i in R1, j in R2, k in R3) s(i,j,k):=getsol(x(i,j,k))
  end-procedure
end-package

```

solarray.mos として保存されたこのパッケージコードは、どんな標準 Mosel モデルとも同じように、BIM ファイル(solarray.bim、ここで、filename はパッケージの名前を示す)にコンパイルされます。

それがワーキング・ディレクトリに含まれていないと、environment variable MOSEL_DSO が、その位置に設定される必要があります。

7 ユーザのモジュールを書く

オペレーティングシステムの観点から見ると、モジュールは、C プログラミング言語で書かれているダイナミックなライブラリ(Dynamic Shared Object、DSO)です。Mosel Native Interface は、1 セットのコンベンションで、DSO は、モジュールとして Mosel によって認められるためには、これを

尊重しなければなりません。C ライブラリの形で実行できるどんな機能性も、Mosel 言語からアクセスできるようにできます。このセクションでは、どのように、(1) 新しいサブルーチンを定義するか、(2) 新しいタイプをどのように定義するかを示します。

7.1 新しいサブルーチンを定義する

前のセクションで説明した、パッケージ'solarray'の solarray ファンクションは、モジュール'solarray'によっても、同じように定義できます。このファンクションを使うと、Mosel モデルへ変更を加えることは必要ありません。'solarray'モジュールは、一つの C ファンクションを定義しますが、これは、array のタイプがどのようなものであると、また、インデックスセットの数がいくつであっても、対応できますが、Mosel 言語とでは、使いたい array index sets の構成 (configuration) ごとに、明示的に、サブルーチン solarray の、ひとつの overloaded version を定義する必要があります。

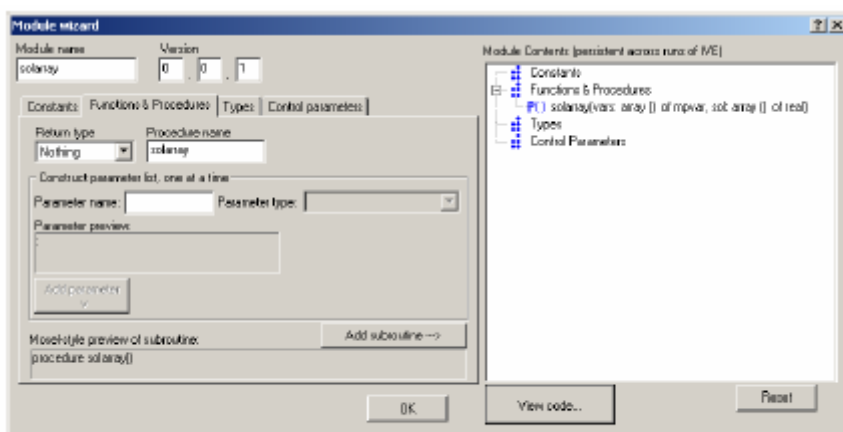


Figure 4: Module definition with IVE

この新しいファンクションを提供するモジュールを実行するコードを、以下に示します。様々なインタフェース構造、イニシャライゼーション・ファンクション、"solarray"を実行するライブラリファンクション ar_getsol は、Xpress-IVE のモジュール生成機能 (Figure 4) で、自動的に生成されます。ユーザに残されている唯一の仕事は、ファンクション ar_getsol に記入することです。簡単にするため、このファンクションによって実行されるエラー処理を省き、単に、必要な操作のみを示します。

1. Mosel スタックから、2つのアレイ(決定変数の array を1つ、実数の array をの1つ)の参照を得る。

2.変数の array のエンTRIESを、MAXDIM dimensionsまで、すべて、エミュレートする(array は、粗であるか、もしかすると、密度が高いかもしれない)。

3.各エンTRIESの、ソリューションの値を得て、そして、それぞれ、実数の array にコピーする。

```

#include <stdlib.h>
#include "xprm_ni.h"
#define MAXDIM 20
static int ar_getsol(XPRMcontext ctx,void *libctx);
/* List of subroutines */
static XPRMdsofct tabfct[]=
{
    {"solarray", 1000, XPRM_TYP_NOT, 2, "A.vA.r", ar_getsol}
};
/* Interface structure */
static XPRMdsointer dsointer=
{
    0, NULL,
    sizeof(tabfct)/sizeof(XPRMdsofct), tabfct,
    0, NULL,
    0, NULL
};
/* Structure for getting function list from Mosel */
static XPRMnifct mm;
/* Module initialization function */
DSO_INIT solarray_init(XPRMnifct nifct, int *interver, int *libver,
                      XPRMdsointer **interf)
{
    m=nifct;                /* Get the list of Mosel functions */
    *interver=XPRM_NIVERS;  /* Mosel NI version */
    *libver=XPRM_MKVER(0,0,1); /* Module version: must be <= Mosel NI version */
    *interf=&dsointer;      /* Pass info about module contents to Mosel */
    return 0;
}
static int ar_getsol(XPRMcontext ctx,void *libctx)
{
    XPRMarray varr, solarr;

```

```

XPRMmpvar var;
int indices[MAXDIM];
/* Get variable and solution arrays from stack in the order that they are
used as parameters for 'getsol' */
varr=XPRM_POP_REF(ctx);
solarr=XPRM_POP_REF(ctx);
/* Error handling:
- compare the number of array dimensions and the index sets
- make sure the arrays do not exceed the maximum number of dimensions MAXDIM
*/
/* Get the solution values for all variables and copy them into the solution
array */
if(!mm->getfirstarrtrumentry(varr,indices))
do
{
mm->getarrval(varr,indices,&var);
mm->setarrvalreal(ctx,solarr,indices,mm->getvsol(ctx,var));
} while(!mm->getnextarrtrumentry(varr,indices));
return XPRM_RT_OK;
}

```

このコードを、ダイナミックなライブラリにコンパイルし、拡張子.dso を与えます。そのあと、そのロケーションを、環境変数 MOSEL_DSO を設定して、Mosel に教えなければなりません。そうすると、Mosel の他のモジュールと同じように、使えるようになります。

この例からわかるように、モジュールの形式で言語の拡張として実行するか、パッケージとして使うかの選択があります。パッケージは、標準の Mosel モデルのように動作しますが、モジュールは下位レベルの言語で書かれるので、これは、一般に、開発に手間が掛かることを意味しますが、同時に、速い実行速度も意味します。この例の arbitrary array indices や下の複素数の例でのオペレータの定義のような一部の機能性は、モジュールでのみ得られます。

7.2 新しいタイプを創る

このセクションでは、Mosel で下記のようなモデルを書けるようにするため、どのように新しいタ

イブ complex を実行し、複素数を表したらよいかを示しています。

```

model Complex numbers
  uses "complex"
  declarations
    c:complex                                ! Define a single complex number
    t:array(1..10) of complex                ! Define an array of complex numbers
  end-declarations
  forall(j in 1..10)
    t(j):=complex(j,10-j)                  ! Initialize with 2 integers or reals
  t(5):=complex("5+5i")                   ! Initialize with a string
  c:=prod(i in 1..5) t(i)                   ! Aggregate PROD operator
  if c<>0 then                              ! Comparison with an integer or real
    writeln("Product: ", c)                ! Printing a complex number
  end-if
  writeln("Sum: ", sum(i in 1..10) t(i))    ! Aggregate SUM operator
                                           ! Arithmetic operators
  c:=t(1)*t(3)/t(4) + if(t(2)=0, t(10), t(8)) + t(5) - t(9)
  initializations to "complex_out.dat"     ! Output to a file
    c t
  end-initializations
end-model

```

いくつかの標準イニシライゼーション、および、新しいタイプを創るファンクションに加え、この新しいタイプを実行するモジュールは、コンストラクター、基礎的な計算操作、equality comparison オペレータ、および、印刷ファンクションを定義します。

ひとたび、タイプが定義されると、この例に示されているように、タイプは、自動的に Mosel データ構造(セット、アレイ)の中で使われます。基礎的な計算操作の定義から、Mosel は、aggregate product、sum のような aggregate operator の定義を推論し、それが適切であるところでは、operand や negation の転換を行います。同じように、equality comparison の定義は、不等式を導くのに十分です。そして、(このケースでは一つも定義されていませんが)、基本的な論理演算子から、aggregate operator の定義が生成されます。印刷ファンクションの定義により、ファイルへの初期化処理による出力も、利用可能になります。

module¹の完全なコードを、すべて示すには、数ページが必要なので、ここでは、このモジュールの、いくつかの重要なフィーチャーに限定し、それのみを示します。

-
- module context
 - type creation and deletion
 - type transformation to and from string
 - overloading of arithmetic operators

前の例と同じように、ここでは、必要なインタフェース構造を作成するのに、IVE のモジュールコード生成ファシリティが使われ、したがって、対応するファンクション本体に記入するだけ済むようになっている、と仮定しています。

7.2.1 モジュールのコンテキスト

モジュールは、実行が終了するとき、割り当てられたスペースが、すべて、解放されるように、モデルの実行の間に作成されたすべてのオブジェクトを記録しておかなければなりません。このファンクションは、モジュールのコンテキストで実行されます。この例では、コンテキストは、複素数の連鎖したリストです。

```
typedef struct
{
    s_complex *firstcomplex;
} s_cxctx;
```

これは、複素数が、下記の構造で示されていると仮定しています。

```
typedef struct Complex
{
    double re, im;
    int refcnt;
    struct Complex *next;
} s_complex;
```

¹ Complete source code available from the authors.

また、モデルの実行の間、モジュールコンテキストは、コントロール・パラメータの現行の値や、モジュールファンクションへのコール間で保存される必要がある、あらゆる情報をストアしておくことにも使用できます。

リセットサービス・ファンクションは、モジュールを使用する Mosel プログラムの実行の開始時と終了のときに呼ばれます。最初のコールで、リセット・ファンクションは、モデルのためのコンテキストを作成して、初期化します。そして、2 番目のコールで、このコンテキスト(および、このモデルのためにモジュールが使った、他の、あらゆる資源)を削除します。

7.2.2 タイプの生成と削除

タイプ・インスタンスの作成、および、削除の目的は、外部のタイプを示す C 構造を扱う(作成、初期化、削除、リセット)こと、および、モジュールコンテキストに格納されている情報をそれ相応にするようにアップデートをおこなうことです。この例では、モジュールで作成されたオブジェクト(複素数)のための、基本的なメモリ管理を実行します。数がつくられるたびに、対応するスペースを割当て、それが削除されるたびに、割当てを解除します。より現実的に言えば、モジュールは、メモリの塊を割り当て、このモジュールによって、以前に割り当てられたスペースをリサイクルします。

一つ一つの複素数のため、クリエーション・ファンクションを、以下の通りに定義します。すなわち、その数が、既に、存在していれば、参照カウンタを増加させ、そうでなければ、この数字に新しい C 構造を割り当て、初期化します。

```
static void *cx_create(XPRMcontext ctx, void *libctx, void *todup, int typrnum)
{
    s_cxctx *cxctx;
    s_complex *complex;
    if(todup!=NULL)
    {
        ((s_complex *)todup)->refcnt++;
        return todup;
    }
    else
    {
        cxctx=libctx;
```

```

    complex=(s_complex *)malloc(sizeof(s_complex));
    complex->next=cxctx->firstcomplex;
    cxctx->firstcomplex=complex;
    complex->re=complex->im=0;          /* Initialize the complex number */
    complex->refcnt=1;
    return complex;
}
}

```

ディリート・ファンクションは、複素数によって使われているスペースを解放し、モジュールコンテキストによって保持されたリストからそれを取り除きます。しかし、この複素数の参照が、まったくない場合以外は、ディリート・ファンクションは、参照カウンタを減少させるだけです。

7.2.3 スtringへの、および、Stringからのタイプの変換

新しいタイプ `complex` を、`initializations` ブロックで使用できるようにするためには、「数をStringに変える」ためのファンクション、「それをStringから初期化する」ためのファンクションの、2つのファンクションを定義しなければなりません。writing ファンクションは、このタイプをプリントするために、手順 `write`、および、手順 `writeln` によっても使用されます。type instance creation function がStringを与えられると、reading ファンクションが適用されます。Stringのフォーマットは、明らかに、タイプに依存します。この例の場合、フォーマットが"re+imi"であるのは明らかです。以下のファンクションは、一つの複素数をプリントします。

```

static int cx_tostr(XPRMcontext ctx, void *libctx, void *toprt, char *str,
                  int len, int typnum)
{
    s_complex *c;
    if(toprt==NULL)
    {
        strcpy(str, "0+0i");
        return 4;
    }
    else

```

```

    {
    c=toprt;
    return sprintf(str, "%g%+gi", c->re, c->im);
    }
}

```

下のファンクションは、文字列から、複素数を読み見込みます。

```

static int cx_fromstr(XPRMcontext ctx, void *libctx, void *toint, const char *str,
                    int typnum)
{
    double re,im;
    s_complex *c;
    if(sscanf(str,"%lf%lf",&re,&im)!=2)
        return XPRM_RT_ERROR;
    else
    {
        c=toint;
        c->re=re;
        c->im=im;
        return XPRM_RT_OK;
    }
}

```

7.2.4 演算子の Overloading

Mosel によって自動的に行われる唯一のタイプ変換は、整数から実数への変換です。しかし、この逆は自動的に変換されません。また、external type にかかわるものも同様です。したがって、2 つの複素数の間の操作は、すべて、定義することが必要です。しかし、可換的演算(添加、乗法、比較)にたいしては、2 つのタイプを結合する 1 つのバージョンを定義することが必要で、もう片方は、Mosel によって推論されます。

例えば、乗法を例にとって説明すると、2 つの複素数の乗法を定義しなければなりません。 $(a + bi) \cdot (c + di) = ac - bd + (ad + bd)i$

```

static int cx_mul(XPRMcontext ctx, void *libctx)
{
    s_complex *c1,*c2;
    double re,im;
    c1=XPRM_POP_REF(ctx);
    c2=XPRM_POP_REF(ctx);
    if(c1!=NULL)
    {
        if(c2!=NULL)
        {
            re=c1->re*c2->re-c1->im*c2->im;
            im=c1->re*c2->im+c1->im*c2->re;
            c1->re=re;
            c1->im=im;
        }
        else
            c1->re=c2->im=0;
    }
    cx_delete(ctx,libctx,c2,0);
    XPRM_PUSH_REF(ctx,c1);
    return XPRM_RT_OK;
}

```

そして、複素数と実数の乗法 $(a + bi) \cdot r = ar + bri$ は、

```

static int cx_mul_r(XPRMcontext ctx, void *libctx)
{
    s_complex *c1;
    double r;
    c1=XPRM_POP_REF(ctx);
    r=XPRM_POP_REAL(ctx);
    if(c1!=NULL)
    {

```

```

        c1->re*=r;
        c1->im*=r;
    }
    XPRM_PUSH_REF(ctx,c1);
    return XPRM_RT_OK;
}

```

実数を複素数と掛けるときは、この操作が可換的演算なので、したがって、Mosel がこのケースを推論するので、乗法を定義する必要はありません。2 つの複素数の加算、実数と複素数の加算は、乗法と非常によく似た方法で実行されます。

2 つのタイプの加算がひとたび行われると、Mosel が引き算 (subtraction: 実数 - 複素数、複素数 - 実数) を推論できるようにするためには、ただ単に、否定 (negation: - complex) を実行する必要があるだけです。割り算の場合は、この操作が可換的演算でないので、すべての 3 つのケース (複素数/複素数、複素数/実数、そして、実数/複素数) を実行する必要があります。

さらに、加算と乗法には、単位元 (identity element) を定義する必要があります。

```

static int cx_zero(XPRMcontext ctx, void *libctx)
{
    XPRM_PUSH_REF(ctx,cx_create(ctx,libctx,NULL,0));
    return XPRM_RT_OK;
}
static int cx_one(XPRMcontext ctx, void *libctx)
{
    s_complex *complex;
    complex=cx_create(ctx,libctx,NULL,0);
    complex->re=1;
    XPRM_PUSH_REF(ctx,complex);
    return XPRM_RT_OK;
}

```

ひとたび、加算と 0-element が定義されると、Mosel は、aggregate operator SUM を推論します。乗法と 1-element で、新しいタイプに、aggregate operator PROD が得られます。

このモジュールで実行される他の演算子は、constructor 演算子、assignment 演算子、comparison 演算子です。

8 むすび

Mosel は、最適化問題をモデル化して、解くためのフレキシブルな環境を提供します。このペーパーで議論した例は、それぞれ異なった Mosel の可能な使い方を示しています。OR に実務家は、皆、自分たちのモデルを速く実行し、そして、容易に維持できる方式に興味を持っています。このペーパーは、Mosel 言語での数学的モデルの定式化の方式が、これらの OR の実務家が行う代数形式に近いという事実を皆様にお伝えする目的で書かれています (see [Guéret et al., 2002] for a collection of examples)。また、Mosel は、数行のコードで、様々なソースのデータを読み書きするデータハンドリングの十分なサポートを提供しています。モデルが大きい場合、ソリューション・アルゴリズムとヒューリスティクスを、直接、Mosel 言語で書くと便利です。アドバンスト・ユーザーや研究者は、例えば、特定の外部のデータソース、新しいソルバー、または、外部のソリューション・アルゴリズムへのアクセスなどのような、自分たちで、それがどのようなフィーチャーであったとしても、自分のアプリケーションが必要とするものを追加できる可能性を、当然、評価してください。

これは、Mosel の重要な特性の一つである、モジュールのアーキテクチャーがあって初めてできることです。すなわち、ソフトウェアとかアプリケーション特有の機能は、Mosel 言語を拡張するモジュールという形式で、容易に加えられます。こうして、このスキームでは、モジュールが、例えば、線形計画法、混合整数計画法のツール以外のソルバーをサポートするのに必要である、新しいデータと変数タイプの定義に使うこととできます。これらは、(二次計画問題などを解く) マトリクスベースのソルバーやソフトウェアで、ここでは、finite domain constraint solver のような、異なった方法を使います。

Bibliography

- [Dash, 1999] Dash Associates. Xpress-MP Reference Manual, 1999.
- [Fourer et al., 1993] R. Fourer, D. Gay, and B. W. Kernighan. AMPL: A Modeling Language for Mathematical Programming. The Scientific Press, San Francisco, CA, 1993.
- [Guéret et al., 2002] C. Guéret, S. Heipcke, C. Prins, M. Sevaux (2002). Applications of Optimization with Xpress-MP. Dash Optimization, Blisworth, UK.
- [Maximal, 2001] Maximal Software. MPL for Windows Reference Manual, 2001.
- [Van Hentenryck, 1998] P. Van Hentenryck. The OPL Optimization Programming Language. MIT Press, Cambridge, MA, 1998.

Conclusion 22 Mosel