SCSK CORPORATION

VDMTools

ダイナミックリンク機能

How to contact SCSK:

http://www.vdmtools.jp/	VDM information web site(in Japanese)
http://www.vdmtools.jp/en/	VDM information web site(in English)
http://www.scsk.jp/	SCSK Corporation web site(in Japanese)
http://www.scsk.jp/index_en.html	SCSK Corporation web site(in English)
vdm.sp@scsk.jp	Mail

ダイナミックリンク機能 *2.0*

— Revised for VDMTools v9.0.2

目 次

1 導入

本書は、ダイナミックリンク機能と呼ばれる VDMTool の特性を記述している。この特性が、 VDM-SL 仕様と C++で記述されたコードとの連結可能性の実現である。この連結で、インタープリタで翻訳中の VDM-SL 仕様で C++で書かれた部分を利用したり実行したりすることが、可能となる。これはダイナミックリンク機能と呼ばれるが、 C++コードがダイナミックに Toolbox とリンクされるからである。

システムの一部のみ VDM-SL を用いて開発した場合に、この連結の利用が興味深いものとなる。次のような場合には、役に立つ可能性がある:

- 既に実装されているシステムに対し、VDM-SL を用いて新しい構成要素を開発すべきであるとき;
- VDM-SL で仕様の定められたシステムで、その VDM-SL 自体では利用できない機能を用いる必要があるとき; そして
- VDM-SL を用いて、単にシステムの一部を開発することが求められているとき。

このダイナミックリンク機能を用いることで、開発過程の初期段階で VDM-SL で仕様が定められた部分と C++で実装された部分との、相互作用を探しだすことが可能となる。

1.1 本書の利用

本書は、*VDM-SL Toolbox* ユーザーマニュアル [SCSd] を拡張したものである。本書を読み進める前に、 *VDM-SL* 言語 [SCSb] のダイナミックリンクモジュール章を読まれることを推奨する。C++ [Str91] および *VDM C++* ライブラリ [SCSa] の知識も、仮定されている。

ダイナミックリンク特性の使用を始めるにあたって、本書すべてを読む必要はない。この特性を利用した例題の詳細な記述を得るには、第 2 章を読むことから始めよう。この例題にあるすべてのファイルは、付録にも載せてある。第 3 章では、ダイナミックリンク機能を利用する場合に含めることとなる構成要素を、1

つ１つ記述する。Toolbox の中にこの機能を実装する方法に興味をもつならば、[FL96] を参照するべきであろう。

付録 A では、システム要求について詳細情報を載せる。

2 はじめに

形式仕様と C++によるコード記述の結合は、それらを結ぶ共通の枠組みの設定を行ってこそ、実現可能となる。もっとも基本的な問題は、これら２つの異なる世界の値が異なる表現をすることにある。２つの世界でやり取りを行うためには、仕様からコードの世界に値を変換したり、またその逆も行うことが不可欠となる。この章では小さな例題を用いて、要求される機能性獲得のため、コードを仕様と統合する考え方を提示する。

例題では、VDM-SL Toolbox を用いて、三角関数の１つの仕様への統合を提供する。三角関数は VDM-SL に含まれず、したがってこのような関数を必要とするシステム開発を行うためには、ダイナミックリンク機能が用いられる。

2.1 基本概念

この手引きの目的は、仕様と実装の連結を分析できることにある。コードと仕様を結びつけることにより最も行いたいことは、コードの実行を仕様翻訳に統合して組み入れることで、機能のプロトタイプを提供することである。形式仕様と統合されるよう、コードレベルでなされた定義を有効にして、仕様のコード定義の翻訳が実行可能であるようにする。したがって、 VDM-SL 仕様とその仕様内の統合部分である仕様レベル、および、コードとその実行のための統合部分であるコードレベル、との間の相違を区別する。

図 1 では、仕様とコードの一体化に伴う要素を示す。第一に VDM-SL 仕様部分と外部 C++ コード部分であり、これらは図中の淡色グレーの四角で表される。次に仕様とコード部分のインターフェイス部分であり、図中では濃色グレーの四角である。このインターフェイス部は、仕様レベル部分 (**ダイナミックリンクモジュール** または単に *DL* **モジュール**と呼ばれる) とコードレベル部分 (**型変換関数**と呼ばれる) からなる。

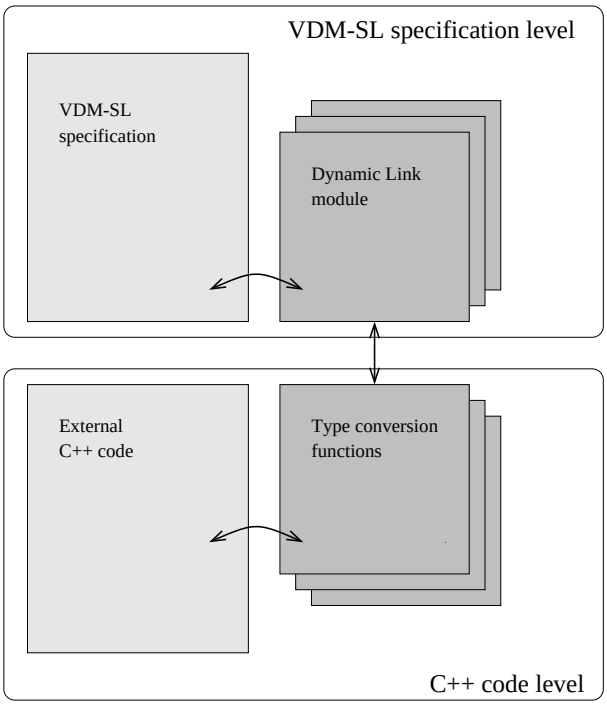

図 1: コードと VDM-SL 仕様の連結

DL モジュールは、 VDM-SL 仕様中で用いられる C++ コードのすべての定義に対する、 VDM-SL インターフェイスを記述する。この情報は、 VDM-SL の関数、操作、値の定義として記述されている。

C++ コードと VDM-SL Toolbox は値に対する表現が異なるために、２つの表現間で変換を行うための型変換関数が定義されなければならない。これらはコードレベルで定義されていなければならない。これらの型変換関数のために、たくさんの慣例が定義されている。一般的に、換えられる側の値は、VDM C++ ライブラリからの型である。(VDM C++ライブラリは、マップ型、集合型、列型、文字型、ブール型、等といった全 VDM 型を実装するクラスを含む。このライブラリは、 [SCSa] に詳細が記述され、ライブラリファイル `libvdm.a` で定義がなされていて、 Toolbox 配布に含まれるものである。)

パラメータを取り値を１つ返す C++関数呼出しのための慣例は、以下の通り:

- パラメータは、VDM C++ ライブラリから “列” 型 の１つの値として渡されるが、各パラメータは列の１要素で: 最初の要素は最初の引数、２番目の要素は２番目の引数、等となる。
- 返される値は一般型 (任意の VDM ライブラリクラス型はこれに変換できる)。

C++ 値や定数を呼び出すために、値／定数を変換し返す関数が、定義されなければならない。

VDM C++値と C++ コード間で、変換を行うための型変換関数が定義されなければならないのに加えて、C++コードの修正を行う必要はない。

2.2 三角関数の例題

概念の説明のために、三角関数を仕様に統合する例題を示す。

VDM-SL 仕様に統合されるべき C++コードが、*math* と呼ばれる数学標準 C ライブラリで与えられている。例題では、三角関数 *sin*、*cos* や 定数 *pi* の実装を、利用している。最初にこれらの定義を適用する仕様が示され、その後にコードレベルと同様の仕様レベルでの拡張が示される。

VDM-SL 仕様

以下の VDM-SL 仕様では、どのように DL モジュール MATHLIB の定義がモジュール CYLINDER にインポートされるかという例題が与えられている。DL モジュールからのインポートが通常モジュールからのインポートといかに同一であるか、つまり、インポートするモジュールはインポートされるモジュールが通常のモジュールであるか DL モジュールであるかが分からない、ということに注意しよう。

```
module CYLINDER

imports
  from MATHLIB
    functions
      ExtCos : real -> real;
      ExtSin : real -> real

    values
      ExtPI : real

 definitions
   functions
     CircCyl_Vol : real * real * real -> real
     CircCyl_Vol (r, h, a) ==
       MATHLIB‘ExtPI * r * r * h * MATHLIB‘ExtSin(a)

end CYLINDER
```

モジュール CYLINDER は、 DL モジュール MATHLIB から、関数 ExtCos と ExtSin および値 ExtPI をインポートする。関数 CircCyl_Vol は円柱の容積を評価し、関数 ExtSin と同じく定数 ExtPI を利用する。

VDM-SL レベルのインターフェイス

上記で述べた通り、コードと仕様間のインターフェイスは、２つの異なるレベルで提供されなければならない。VDM-SL レベルのインターフェイスは、外部世界

にエクスポートされる関数の VDM-SL 型を宣言する。コードレベルのインターフェイスは、上記で述べた型変換関数の定義に基づく。.

VDM-SL レベルでのインターフェイスは次のようなものである:

```
dlmodule MATHLIB

  exports
    functions
      ExtCos : real -> real;
      ExtSin : real -> real

    values
      ExtPI : real

    uselib
     "libmath.so"

end MATHLIB
```

DL モジュールは常に、外部世界で利用できる全構成要素を宣言するエクスポートセクションを含まなければならない。エクスポートセクションにある構成要素は、他のモジュールによるインポートが可能である。エクスポートセクションは、関数と操作定義のためのシグニチャと、C++コードに対し VDM-SL インターフェイスを定義する値定義の型情報からなる。DL モジュール内の値宣言は、コード内の定数または変数定義に関連している。

uselib 項は共有オブジェクトファイルの名称を含むが、DL モジュールを通してアクセスされたコードを含むものである。

C++レベルのインターフェイス

コードレベルのインターフェイスは、C++で開発され、たくさんの宣言と型変換関数の定義から成り立っている。宣言部分には標準数学ライブラリを含める。型変換関数は、Toolbox が用いる VDM C++ 列値を C++ コードが受け取る値に

変換し、またその逆も行う。一般的な VDM C++ 型は、任意の VDM 型に内在する値をもつことができる (たとえば一般的な VDM C++型である 列 は、任意の VDM 要素を含めることが可能な VDM-SL 列を表す。)

インターフェイスは次のようになる:

```
#include "metaiv.h"
#include <math.h>

# Platform specific directives/includes...

extern "C" {
  Generic ExtCos(Sequence sq);
  Generic ExtSin(Sequence sq);
  Generic ExtPI ();
}

Generic ExtCos(Sequence sq)
{
  double rad;
  rad = Real(sq[1]);
  return (Real( cos(rad) ));
}

Generic ExtSin(Sequence sq)
{
  double rad;
  rad = Real(sq[1]);
  return (Real( sin(rad) ));
}

Generic ExtPI (Sequence sq)
{
  return(Real(M_PI));
}
```

ExtPI は、 VDM-SL インターフェイス中では VDM-SL 値、しかしコードレベ

ルインターフェイス中では関数、として定義されていることに注意しよう。

Toolbox は、呼び出された関数の引数を一般的な VDM C++ 列型の値として取り入れて、それを型変換関数に渡す。型変換関数は、列の要素を抽出し変換し、統合された C++コードで求められる値にする。たとえば型変換関数 `ExtSin` は、列の最初の要素を抽出し、それを VDM C++ 型 `Real` に型変換するが、これは自動的に C++型 `double` になる。これが C++レベルで `sin` 関数に対する引数として与えられ、結果は VDM C++ 値に変換されて Toolbox に返される。

この場合、C 標準ライブラリはコードとして利用されたが、ユーザー定義の C++ パッケージであったとしても同様であった。この方法は、C++で開発される任意のモジュールに公開されている。ユーザーにはただ、 DL モジュールを開発し型変換関数を定義することだけ要求される。型変換関数は 外部 `"C"`関連仕様として取り込まれていなければならず、C++ パラメータ型絞込みは、共有オブジェクトライブラリにある関数標識名称から一部つくられたものではないことに注意したい。

共有ライブラリの生成

ダイナミックリンクライブラリの生成には、上記のように 外部 `"C"`関連仕様に型変換関数が取り込まれていることが求められる。加えて、上記のように`metaiv.h` ヘッダーファイルを含むことが求められる。さらに C++ コードは、1 つのサポートされるコンパイラ (第 A 章を参照) で、共有ライブラリ生成に必要なフラグを用いてコンパイルされなければならない。ライブラリを Make ファイルで生成できる方法は、記録が推奨される; 例題の Make ファイルは不可欠なフラグと共に、付録 B.3 で示されている。Solaris 10 に対し、この例題の Make ファイルは次のようなものとなるであろう[1]:

```
all: libcylio.so libmath.so

%.so:
        ${CXX} -shared -mimpure-text -v -o $@ -Wl,-B,symbolic\
        $^  ${LIB}
```

[1]ただし、C++ レベルのインターフェイスが `tcfmath.cc` という名称のファイルにおかれると仮定する。

```
libcylio.so: cylio.o tcfcylio.o

cylio.o: cylio.cc
        ${CXX} -c -fpic -o $@ $< ${INCL}

tcfcylio.o: tcfcylio.cc
        ${CXX} -c -fpic -o $@ $< ${INCL}

libmath.so: tcfmath.o

tcfmath.o: tcfmath.cc
        ${CXX} -c -fpic -o $@  $<  ${INCL}
```

他のサポートされるプラットフォーム／コンパイラで用いるオプションについては、第 3.7 章で見ることができる。

例題の実行

共有ライブラリ `libmath.so` を生成したら、Toolbox を始めることができる。`CYLINDER` モジュールと `MATHLIB` DL モジュールは、他のモジュール同様、構文チェックを行うことで Toolbox に読み込むことができる。今現在の仕様を初期化することで、`CircCyl_Vol` 関数が翻訳できるというような標準ライブラリへのリンクを構築することが可能となるわけで、ここでは `MATHLIB‘ExtPI` や `MATHLIB‘ExtSin` の呼び出しがライブラリの呼び出しとなる。 VDM-SL 部分もまた **VDMTools** デバッガを用いることでデバッグ可能であるが、一方で明白なことだが、コード部分はデバッグできない。

例題の拡張

本書の付録の中で、この小例題にはもう 1 つの DL モジュールを追加して拡張している。この追加モジュールでは、円柱の寸法を入力するための簡単な入出力インターフェイスの使用方法を説明する。例題リポジトリ

(http://www.csr.ncl.ac.uk/vdm/examples/examples.html) にある他の例題では、Tcl/Tk を使用したより一般的なユーザーインターフェイス利用を説明している。

3 ダイナミックリンク構成要素

この章では、ダイナミックリンク機能を使用する場合に含まれる様々な構成要素について、１つ１つ説明する。目的は、ここまでの章を一度読んだ後、この章を参照ガイドとして利用することができるようにすることである。

本書では一般的な名称慣例を用いてきた。ここでは自身で慣例をつくることを推奨することになる。これにより、どこで構成概念が定義されたか見つけ出すのがずっと容易になる。使用してきた名称慣例の中には、外部で (つまり、C++ コード中で) 定義された構成要素すべてに ‘‘Ext’’ 接頭辞をつける、というものがある 。変換関数をもつすべてのファイルは ‘‘tcf’’ 接頭辞をつける。最後に、Unix 上では、すべてのダイナミックリンクライブラリが ‘‘lib’’ 接頭辞をつけてすべてが .so 拡張子をとり; Windows 上では、すべてのダイナミックリンクライブラリが .dll 拡張子をとる。

3.1 ダイナミックリンクモジュール

DL **モジュール** は、コード部分への VDM-SL インターフェイスを提供する。モジュール名称に加えて、インポート節、エクスポート節、そしてダイナミックリンクを行うためのライブラリ (C++コード含む) の名称、が含まれている。

- インポート節はオプションであるが、DL モジュール内で使用される VDM-SL 型を記述する。この例題は、付録 B.1.2 の CYLIO DL モジュールで提示されている。
- エクスポート節は、DL モジュールから利用できる関数、操作、値を記述する。DL モジュール内の通常のモジュールとは異なり、エクスポートされた構成要素のすべてを明示的に記述しなければならない、つまり exports all はない、ということに注意しよう。

- 共有ライブラリの名称は、ダイナミックリンクを行うライブラリを示す。これは残しておくこともできて、この場合の仕様は、構文および型のチェックはできるが、初期化や翻訳はできない。

  第 3.5 章では、Toolbox はライブラリを探す場所をどのように定義するのか、を述べる。

DL モジュールの正確な構文と動作がさらに詳細に、*VDM-SL* **言語** [SCSb] の**ダイナミックリンクモジュール**章に記述されている。

関数と操作の違いは、通常モジュールに対すると同じく DL モジュールに対して: 関数は、値を返さなければならず副次的作用があってはならない。反対に、操作は副次的作用を起こす可能性があるので、値を返す必要はない。当然ながらこれは単に、Toolbox ではチェックできない実際的な相違である。

3.2 型変換関数

型変換関数の目的は、Toolbox により使用される値（これらは *VDM C++* ライブラリのオブジェクトである）を、統合されたコードで求められる値に変換したりその逆を行ったりすることである。型変換関数は、異なる引数の数をもった関数を取り扱うために、引数として *VDM C++* クラス 列 型オブジェクトを常に取らなければならない。ダイナミックリンクモジュールからエクスポートされた各々の構成要素に対し、型変換関数が定義されなければならない。

C++ コードで定義された関数（たとえばdouble Volume (double radius, double height)）を統合する場合、型変換関数が定義されるべきである。この場合、 DL モジュールは相当の関数や操作、たとえば functions ExtVolume: real * real -> real、を含むべきである。Toolbox は、翻訳された仕様によって与えられる評価済みの引数を Sequence クラスのオブジェクトに取り入れ、型変換関数に渡す。この小さな例題の定義は以下の通り:

```
Generic ExtVolume (Sequence sq1)
{
  double rad, height;
  rad = (Real) sq[1];
  height = (Real) sq[2];
```

```
    return( (Real) Volume(rad, height) );
  }
```

関数 Volume に対する引数は、列から抽出され、倍精度値に変換される。Volume からの返り値は、クラス Real のオブジェクトに変換され、これは自動的に Generic に型変換される。

Table 1 では、 DL モジュールで関連する定義の概観、型変換関数、そして C++ コードを、つまり様々なレベルで値がどのように表されるかを、提示する。付録 B は、三角関数例題のための型変換関数を含む。

3.3 レコード変換

レコードに対しては、特に注意を払う必要がある。VDM-SL において、仕様レコードには名称がタグ付けされるが、一方 VDM C++ライブラリにおいては、整数がタグ付けされる。これらのタグは以下の章で、各々‘‘記号名’’ および ‘‘数値タグ’’ として参照される。

VDM C++ ライブラリは、数値タグから記号名への写像であるマップ（レコード情報マップと呼ぶ）の管理ができる。ダイナミックリンクモジュールのコード部分への受け渡しを行う記号名はいずれも、このマップに登録されたものでなければならない。

この登録は、InitDLModule という関数中で定義されなければならない。この関数が存在するなら、 Toolbox は初期化中に (init コマンドを実行するとき) これを呼び出す。以下の例題は、InitDLModule 関数内でどのようにレコード情報マップがつくられるかを明らかする:

- 用いられる各記号名に対して、レコード情報マップ内で各々に相当する数値タグが追加されなければならない。これは次の実行で行われる:

  ```
  VDMGetDefaultRecInfoMap().NewTag(numtag, recordsize);
  ```

  numtag は数値タグ、 recordsize はレコード中の項目数である。

- 記号名は数値タグと結び付かなければならない。これは次の実行で行われる:

  ```
  VDMGetDefaultRecInfoMap().SetSymTag(numtag, "symbname");
  ```

DL モジュール定義	型変換関数	*C++* コード
値定義	引数なし結果型 Generic の関数	変数または定数
values ExtLength: real ExtMax:nat	Generic ExtLength () Generic ExtMax ()	double Length = 5.0; const int max = 20;
関数または操作定義	引数 Sequence 結果型 Generic の関数	結果型 void でない関数
functions ExtVolume: real * real -> real or operations ExtVolume: real * real ==> real	Generic ExtVolume(Sequence sq1)	double Volume(double radius, double height)
結果なしの操作	引数 Sequence 結果型 void の関数	結果型 void の関数
operations ExtShowItem: nat ==> ()	void ExtShowItem(Sequence sq1)	void ShowItem (int item)

表 1: 様々なレベルの値表現

ここでも numtag は数値タグ、 symbname は記号名である。記号名は引用符を用いて囲われていなければならないことに注意しよう。

例題の中で、数値タグとレコードサイズは定数 (TagA_X と TagA_X_Size) で定義される。

例題:

```
extern "C" void InitDLModule(bool init);
const int TagA_X = 1;
const int TagA_X_Size = 2;
void InitDLModule(bool init) {
  if (init) {
    VDMGetDefaultRecInfoMap().NewTag(TagA_X, TagA_X_Size);
    VDMGetDefaultRecInfoMap().SetSymTag(TagA_X, "A`X");
  }
}
```

Toolbox は、C++ コードに送る A`X という名称のレコード値を生成するとき、DL モジュール中のレコード情報マップで、記号名 A`X に相当する数値タグを見つけることになる。

レコード情報マップを生成する場合は以下の問題に注意しよう:

- SetSymTag コマンドの記号名は、相当するモジュール名で修飾された現存のレコード名でなければならないし、NewTag コマンドのレコードサイズは正確にレコード項目数でなければならない。
- レコード情報マップ中で各記号タグが現れるのは、各々一回でなければならない。

3.3.1 コードジェネレータと組み合わせたレコード利用

コードジェネレータは、コード生成されたモジュールに対して、レコード情報マップを構成するコードを生成する。このように、生成コードがコンパイルされてダイナミックリンクライブラリにリンクされるならば、 InitDLModule 関数が単

に関数 init_*Module*() を呼び出すことで、レコード情報マップ中の *Module* 内でレコード定義を行うことができる。

3.4 トークンの変換

トークン型は C++ クラス レコードとして、ツールボックスからエクスポートされる。しかし、トークンレコードのタグは常にファイル cg_aux.h におけるマクロ宣言である TOKEN と同等となり、トークンレコードの項目数は常に 1 である。外部コードから toolbox へトークンを送ろうとする場合、VDM-SL 値 mk_token(<HELLO>) を、たとえば以下の方法で、構成できる:

```
Record token(TOKEN, 1);
token.SetField(1, Quote("HELLO"));
```

3.5 uselib パス環境

仕様中で、ライブラリ名称は uselib オプションで与えられる。この場所については、いくつかの方法で与えることができる:

- ライブラリへの絶対パス名称（たとえば /home/foo/libs/libmath.so) で与える。
- パスなしで環境変数 VDM_DYNLIB の集合で与える。VDM_DYNLIB 環境変数の全ディレクトリ名称に対しては、ライブラリが検索を行う。環境変数は次のようなものとなる: /home/foo/libs:/usr/lib:. (変数設定において、カレントディレクトリも捜す検索の場合は、‘.’ が必要)。
- パスなしで VDM_DYNLIB 環境変数の集合も与えない。ライブラリがカレントディレクトリに置かれていると仮定することを意味する。

3.6 DL モジュールの初期化

Toolbox コマンド ‘‘init’’ が初回に用いられるときに、全モジュールは読み込まれる。２度目には、現在読み込まれているモジュールがまずは領域解放され、その後にモジュールは改めて読み込まれる。

3.6.1 モジュール搭載

共有ライブラリファイルが Toolbox に読み込まれると、以下が起きる:

- グローバル変数が初期化される。
- toolbox が各搭載モジュールにある関数 InitDLModule(true) を実行する。

3.6.2 モジュール領域解放

共有ライブラリファイルが Toolbox により領域解放されると、以下が起きる:

- toolbox が各搭載モジュールの関数 InitDLModule(false) を実行する。
- グローバル変数が破棄される。

3.7 共有ライブラリの生成

共有ライブラリは、第 A 章に示される適切なコンパイラを用いてコンパイルされなければならない。実行可能な共有ライブラリを生成するために、型変換関数を含むコードは以下の要求を満たす必要がある:

- 型変換関数は 外部 "C"関連仕様の中に入っていなければならないが、そうでない場合に、これらの関数がライブラリの外から連絡可能でなくなるからである。

- VDM C++ ライブラリの一部である metaiv.h ヘッダーファイルは、型変換関数を含むファイル中になければならないが、これが *VDM C++* ライブラリの型仕様関数のプロトタイプを含んでいるからである。

実際に利用するコンパイラフラグについては、B.3 にある Make ファイル例題を参照しよう。

Unix 上で、共有ライブラリならばファイル拡張子は ".so" とし、接頭辞 "lib" をつけるべきである。(共有ライブラリでは版番号は必要でない。) Windows 上では、共有ライブラリはファイル拡張子を ".dll"とするべきである。

参考文献

[FL96] Brigitte Fröhlich and Peter Gorm Larsen. Combining VDM-SL Specifications with C++ Code. In Marie-Claude Gaudel and Jim Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 179--194. Springer-Verlag, March 1996.

[SCSa] SCSK. *The VDM C++ Library*. SCSK.

[SCSb] SCSK. *The VDM-SL Language*. SCSK.

[SCSc] SCSK. *The VDM-SL to C++ Code Generator*. SCSK.

[SCSd] SCSK. *VDM-SL Toolbox User Manual*. SCSK.

[Str91] B. Stroustrup. *The C++ Programming Language, 2nd edition*. Addison Wesley Publishing Company, 1991.

A　システム要求

ダイナミックリンク機能を利用するためには、ダイナミックリンク機能のライセンスを含む *VDM-SL Toolbox* と *VDM C++* ライブラリ が必要である。さらに、実行可能な統合コード共有ライブラリを生成するために、コンパイラが必要である。

この機能は次の組み合わせで動く：

- Microsoft Windows 2000/XP/Vista 上の Microsoft Visual C++ 2005 SP1
- Mac OS X 10.4, 10.5
- Linux Kernel 2.4, 2.6 上の GNU gcc 3, 4
- Solaris 10

実行可能な統合コード共有ライブラリを生成するために、コンパイラが必要である。Toolbox 自身のインストールには、 [SCSd] 第 2 章を、 *VDM C++* ライブラリ のインストールには、 [SCSc] 第 2 章を参照しよう。

B　三角関数例題の概観

この例題中で円柱の容積を計算するために、仕様にある三角関数のための簡単なユーザーインターフェイスが統合されている。この例題は、ディレクトリ example 中に配布されていて利用可能である。例題の実行には以下を行うこと：

- 適切な Make ファイルを編集し、ライブラリがある場所の先頭をパス設定する。
- ライブラリをコンパイルする。

  Linux `make -f Makefile.Linux`

  Solaris 10 `make -f Makefile.solaris2.6`

  Microsoft Windows 2000/XP/Vista `make -f Makefile.win32`

- Toolbox をスタートし、仕様を読み込み、関数の初期化と呼び出しを行う。Toolbox コマンドライン版では以下のようになる:

```
vdm> r cylio.vdm
Parsing "cylio.vdm" ... done
vdm> r cylinder.vdm
Parsing "cylinder.vdm" ... done
vdm> r math.vdm
Parsing "math.vdm" ... done
vdm> init
Initializing specification ...
vdm> p CYLINDER`CircCylOp()


 Input of Circular Cylinder Dimensions
  radius: 10
  height: 10
  slope [rad]: 1


 Volume of Circular Cylinder

Dimensions:
   radius: 10
   height: 10
   slope: 1

 volume: 2643.56

(no return value)
```

VDM-SL 文書は、DL モジュール *MATHLIB* と *CYLIO* をインポートするモジュールである *CYLINDER* から構成される。DL モジュール *MATHLIB* は、三角関数 *sinus* と値 π に対するインターフェイスを提供している。モジュール *CYLIO* は、円柱の寸法を設定しその容量を表示するための関数を含める。モジュール *CYLINDER* もまた、DL モジュールによってインポートされた円柱の寸法を表示するためのレコードを定義している。

各 DL モジュールに対して相当する共有ライブラリが、次の通り存在する: libmath.so/math.dll に対して *MATHLIB*、また libcylio.so/cylio.dll に対して *CYLIO*。共有ライブラリ libmath.so/math.dll は、三角関数 *sine*、 *cosine*、や値 π を提供するが、すべて C 標準ライブラリで定義されているものである。コードは型変換関数から構成される。共有ライブラリ libcylio.so（あるいは Windows 上ならば cylio.dll）は、簡単なユーザーインターフェイス関数を提供する: GetCircCyl が円柱の寸法を訊ねると ShowCircCylVol が円柱の寸法とその容積を画面上に表示する。円柱の寸法は、レコード型 *CircCyl* の仕様のなかで同等の定義がなされているが、この構造 CircCyl によって表わされる。

B.1 仕様

この付録では、モジュール仕様を含む。

B.1.1 モジュール *CYLINDER*

```
module CYLINDER
  imports
    from MATH
      functions
        ExtCos : real -> real;
        ExtSin : real -> real
      values
        ExtPI : real,

    from CYLIO
      functions
        ExtGetCylinder : () -> CircCyl

      operations
        ExtShowCircCylVol : CircCyl * real ==> ()

   exports
     operations
```

```
      CircCylOp:() ==> ()
    types
      CircCyl

  definitions
    types
      CircCyl :: rad : real
                 height : real
                 slope : real

    functions
      CircCylVol : CircCyl -> real
      CircCylVol(cyl) == MATH‘ExtPI * cyl.rad * cyl.rad *
                         cyl.height * MATH‘ExtSin(cyl.slope);

    operations
      CircCylOp : () ==> ()
      CircCylOp() == ( let cyl = CYLIO‘ExtGetCylinder() in
                         let vol = CircCylVol(cyl) in
                           CYLIO‘ExtShowCircCylVol(cyl, vol))


end CYLINDER
```

B.1.2 ダイナミックリンクモジュール *MATHLIB* と *CYLIO*

```
dlmodule MATH
  exports
    functions
      ExtCos : real -> real;
      ExtSin : real -> real
```

```
   values
      ExtPI : real

   uselib
      "math.dll"

end MATH

dlmodule CYLIO
  imports
    from CYLINDER
      types
        CircCyl

   exports
     functions
       ExtGetCylinder : () -> CYLINDER`CircCyl

     operations
       ExtShowCircCylVol : CYLINDER`CircCyl * real ==> ()

   uselib
      "cylio.dll"

end CYLIO
```

B.2 共有ライブラリ

この付録は、共有ライブラリを構築するのに用いられるソースファイルを含む。

B.2.1 MATHLIB 共有ライブラリ

```
//-------------------------------------------------------------------
// tcfmath.cc   type conversion functions for libmath.so
//-------------------------------------------------------------------
#include "metaiv.h"
#include <math.h>

#ifdef WIN32
#define DLLFUN __declspec(dllexport)
#define M_PI   3.14
#else
#define DLLFUN
#endif

extern "C" {
  DLLFUN void InitDLModule(bool init);
  DLLFUN Generic ExtCos(Sequence sq);
  DLLFUN Generic ExtSin(Sequence sq);
  DLLFUN Generic ExtPI ();
}

void InitDLModule(bool init)
{
  // This function is called by the Toolbox when modules are
  // initialised and before they are unloaded.
  // init is true on after load and false before unload.
}

Generic ExtCos(Sequence sq) {
  return (Real( cos((Real)sq[1]) ));
}

Generic ExtSin(Sequence sq) {
  return (Real( sin((Real) sq[1]) ));
}
```

```
Generic ExtPI () {
  return(Real(M_PI));
}
```

型変換関数 ExtSin は、引数としてクラス Sequence のオブジェクトを取り込み、その引数を抽出してクラス Real のオブジェクトに変換する。このオブジェクトは、 *sin* 関数のアプリケーションによって、自動的に double 値に型変換される。結果の値は Meta-IV に変換され、Toolbox に返される。

B.2.2 CYLIO 共有ライブラリ

この章に、共有ライブラリ libcylio.so/cylio.dll のソースを含む。ファイル cylio.cc が、円柱の寸法の入力と容量の表示出力のための簡単なインターフェイスを提供する。

ファイル tcfcylio.cc は型変換関数を含む。型変換関数もまた、C++ や Toolbox で用いられる型 ‘‘CircCyl’’ の様々な表現間における円柱値の転換を表す。

```
//---------------------------------------------------------------
// cylio.h        Definition of the structure used.
//---------------------------------------------------------------
struct CircularCylinder {
  float rad;
  float height;
```

```
  float slope;
};

typedef struct CircularCylinder CircCyl;

extern CircCyl GetCircCyl();
extern void ShowCircCylVol(CircCyl cylinder, float volume);


//-----------------------------------------------------------------
// cylio.cc        Circular Cylinder simple I/O functions
//-----------------------------------------------------------------
#include <iostream.h>
#include "cylio.h"

CircCyl GetCircCyl() {
  CircCyl in;

  cout <<"\n\n Input of Circular Cylinder Dimensions";
  cout <<"\n  radius: ";
  cin >> in.rad;

  cout <<"  height: ";
  cin >> in.height;

  cout <<"  slope [rad]: ";
  cin >> in.slope;

  return in;
}


void ShowCircCylVol(CircCyl cyl, float volume) {
  cout << "\n\n Volume of Circular Cylinder\n\n";
  cout << "Dimensions: \n   radius: " << cyl.rad;
  cout << "\n   height: " << cyl.height;
  cout << "\n   slope: "<< cyl.slope << "\n\n";
```

```
  cout << " volume: " << volume<< "\n\n";
}
```

```
//--------------------------------------------------------------------
// tcfcylio.cc  type conversion functions for circular cylinder io
//--------------------------------------------------------------------

#include "metaiv.h"
#include "cylio.h"

#ifdef WIN32
#define DLLFUN __declspec(dllexport)
#else
#define DLLFUN
#endif

extern "C" {
  DLLFUN void InitDLModule(bool init);
  DLLFUN Generic ExtGetCylinder(Sequence sq1);
  DLLFUN void ExtShowCircCylVol(Sequence sq1);
}

const int tag_CYLINDER_CircCyl = 1;
const int size_CYLINDER_CircCyl = 3;

void InitDLModule(bool init)
{
  if (init) {
    VDMGetDefaultRecInfoMap().NewTag(tag_CYLINDER_CircCyl,
```

```
                                        size_CYLINDER_CircCyl);
    VDMGetDefaultRecInfoMap().SetSymTag(tag_CYLINDER_CircCyl,
                                        "CYLINDER`CircCyl");
  }
}

Generic ExtGetCylinder(Sequence sq1)
{
  CircCyl cyl;
  Record Rc(tag_CYLINDER_CircCyl,size_CYLINDER_CircCyl );

  cyl = GetCircCyl();   // input of cylinder dimension
  Rc.SetField(1, (Real)cyl.rad); // conversion in VDM C++ record class
  Rc.SetField(2, (Real)cyl.height);
  Rc.SetField(3, (Real)cyl.slope);
  return(Rc);    // return Record to the interpreter process
}

void ExtShowCircCylVol(Sequence sq1)
{
  CircCyl cyl;
  Record Rc(tag_CYLINDER_CircCyl,size_CYLINDER_CircCyl);
  float vol;

  // extract cylinder dimension and volume from sequence
  Rc =  sq1[1];
  vol = (Real) sq1[2];

  // convert Record in a C++ structure
  cyl.rad = (Real) Rc.GetField(1);
  cyl.height = (Real) Rc.GetField(2);
  cyl.slope = (Real) Rc.GetField(3);

  ShowCircCylVol(cyl, vol); //make output
  return;
}
```

B.3 Makeファイル

B.3.1 Linux上

ここにあるのは Linux 上の Make ファイル例題。

```
##--------------------------------------------------------------
##                    Make file for Linux
##--------------------------------------------------------------

#VDMLIB  = /opt/toolbox/cg/lib
VDMLIB  = /local2/paulm/vice
#INCL    = -I/opt/toolbox/cg/include
INCL    = -I/local2/paulm/vice

CC      = /opt/gcc-3.0.1/bin/g++
LIB     =  -L$(VDMLIB) -lvdm -lm # -lstdc++ -lgcc
# -Wl,-Bstatic

## Nothing below this line should be changed.

all: libcylio.so libmath.so

%.so:
${CC} -shared -fpic $(EXCEPTION) -v -o $@ $^ $(LIB)

libcylio.so: cylio.o tcfcylio.o

cylio.o: cylio.cc
${CC} -c $(EXCEPTION) -fpic -o $@ $< ${INCL}

tcfcylio.o: tcfcylio.cc
${CC} ${INCL} -c $(EXCEPTION) -fpic -o $@ $<
```

```
libmath.so: tcfmath.o

tcfmath.o: tcfmath.cc
${CC} -c $(EXCEPTION) -fpic -o $@  $<  ${INCL}


clean:
rm *.o *.so

# target for debuging
debug: cylio.o tcfcylio.o main.cc
${CC} -c main.cc -o main.o ${INCL}
${CC} -o debug cylio.o tcfcylio.o main.o $(LIB)
```

ここにあるのは Solaris 10 上の Make ファイル例題。

```
##-----------------------------------------------------------------
##                     Make file for Solaris 2
##-----------------------------------------------------------------
VDMLIB  = /opt/toolbox/cg/lib
INCL    = -I/opt/toolbox/cg/include

## Compiler flags
CC      = g++
LIB     = -L$(VDMLIB)  -lvdm -lm

all: libcylio.so libmath.so

%.so:
${CC} -shared -mimpure-text -v -o $@ -Wl,-B,symbolic $^  ${LIB}

libcylio.so: cylio.o tcfcylio.o

cylio.o: cylio.cc
${CC} -c -fpic -o $@ $< ${INCL}
```

```
tcfcylio.o: tcfcylio.cc
${CC} -c -fpic -o $@ $< ${INCL}

libmath.so: tcfmath.o

tcfmath.o: tcfmath.cc
${CC} -c -fpic -o $@  $<  ${INCL}


clean:
rm *.o *.so
```

B.3.2 Windows 上

ここにあるのは Microsoft Windows 上の Make ファイル例題で、GNU Make を用いている。

```
##-------------------------------------------------------------------
##                      Make file for Windows 32bit
##              This Makefile can only be used with GNU make
##-------------------------------------------------------------------

CC       = cl.exe
CFLAGS   = /nologo /c /MT /W0 /GD /GX /D "WIN32" /D "_USRDLL" /TP
INCPATH  = /I//hermes/georg/toolbox/winnt

LINK     = link.exe
LPATH    = /LIBPATH:C:/work
LFLAGS   = /dll /incremental:no /DEFAULTLIB:vdm.lib

# IMPLICITE RULES

%.obj: %.cc
```

```
$(CC) $(CFLAGS) $(INCPATH) /Fo"$@" $<

%.dll:
$(LINK) $(LPATH) $(LFLAGS) /out:"$@" $^

# TARGETS

all: math.dll cylio.dll

math.dll: tcfmath.obj

cylio.dll: tcfcylio.obj cylio.obj

clean:
rm -f math.lib math.exp math.dll
rm -f cylio.lib cylio.exp cylio.dll
rm -f tcfmath.obj tcfcylio.obj cylio.obj
rm -f *~
```

C トラブルシューティング

一旦独自の型変換関数の生成を始めると、見慣れないエラーメッセージが VDM C++ライブラリから出されることに気づくだろう。この付録は、ライブラリをどのようにデバッグするか考える際に、助けとなるはずである。

一番よい方法は、通常通り gdb を始めて、どこがうまくいかないのか理解するため、コードを通してデバッグすることである。これは早急にはできないが、gdb のスタート時には、ライブラリ中で定義される記号がまだわからないからである。

解決法としては、第一に vdmde の仕様をデバッグすることであり、そしてどの外部関数がうまくいかないかを発見したとき、特定の関数を呼び出す main() が生

成され、ライブラリと共にリンクされる。主な手順は次のようになるだろう:

```
#include "metaiv.h"

extern "C" {
  Generic ExtGetCylinder(Sequence);
}

main() {

  Sequence sq; // empty sequence to call ExtGetCylinder with.
  Generic result;

  result = ExtGetCylinder(sq);
}
```

C.1 分割障害

もう 1 つの問題として、 toolbox が分割障害でクラッシュすることがある。これは、外部ライブラリの問題として大変よく問題になる。どの関数で問題が起きるのかを確かめるため、 vdmde を再び始める。

Toolbox と結果としてこのようなエラーとなったコードとの間のインターフェイスで、うまくいかない可能性が明らかに 3 つある:

1. 関数が正しいシグニチャをもたない:
   void *function*(Sequence *args*)
   または Generic *function*(Sequence *args*)
2. Generic を返すべき関数が値を返さない。
3. 外部コードが、グローバルに初期化されていない値またはオブジェクトを含める (第 C.3 章を参照)。

C.2 Tcl/Tk の使用

ダイナミックリンクライブラリで Toolbox と共に Tcl/Tk の使用を望む場合は、この結合のために作成された他の例題を見るべきである[2]。vdmgde (Toolbox のグラフィカル版) を用いている場合は Tk_MainLoop を呼び出さないことが重要で、 vdmgde が Tcl/Tk を実装しているからであり、そのため Tk_MainLoop が呼び出された場合は Tcl/Tk がメインの Toolbox ウィンドウがクローズされるのを待つというデッドロック状況に入ってしまうからである。

1 つの解決法 (他の例題で用いられている) として、Tk_DoOneEvent を呼び出して Tcl コードが Tcl 変数 Done をゼロ以外に設定した場合に終了するような、ループを生成することである。最初は Tcl コードが Done をゼロに設定し、ダイアログウィンドウがクローズされたときには Done は 1 に設定される。

C.3 グローバルオブジェクトとグローバル値初期化

サポートされているすべてのプラットフォーム (Linux、Solaris 10、Windows) 上で、上記のような共有オブジェクトのコンパイルやリンクが成されるとき、 C++ オブジェクト構成要素が呼び出される。

C.4 Microsoft Windows の標準入出力

Windows のもとで、Toolbox のグラフィカル版は GUI アプリケーションとして実行されるため、標準入出力は利用できない。したがって、cin、cout、cerr を使用しようとする共有ライブラリは、期待したようには動かないはずである。

[2] 現在提供されている例題はない。