九州大学
KYUSHU UNIVERSITY

VDMTools

VDMTools APIマニュアル

ver.1.0

How to contact:

http://fmvdm.org/	VDM information web site(in Japanese)
http://fmvdm.org/tools/vdmtools	VDMTools web site(in Japanese)
inq@fmvdm.org	Mail

VDMTools APIマニュアル 1.0

— Revised for VDMTools v9.0.6

© COPYRIGHT 2016 by Kyushu University

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement.

This document is subject to change without notice

目 次

1 導入

本書は、VDM Toolboxに含まれるCORBA対応APIの使い方について記述したものである。

VDM Toolbox APIでは、グラフィカルな画面からあるいはコマンドラインから、VDM Toolbox実行インスタンスの一定プロパティの参照や修正を行うような、クライアントプログラムの作成が許されている。VDM Toolboxでは、同時に複数のCORBAクライアントからのアクセスが可能である。これらクライアントはAPIを通して、プロジェクトにアクセスし設定を行う、個々のファイルの構文分析や型検査をする、インタープリタを通した式評価を行う、等々が可能である。クライアントプロセスとVDM Toolboxは分離したプロセスであり、ネットワーク上の異なるマシン上において、場合によっては異なるオペレーティングシステム上で、実行され得るものである。その結果、 あるクライアントプロセスでサーバーとして用いられているVDM Toolboxを、ユーザーインターフェィスを通してこのユーザーが利用できる。

APIはCORBA (see [4])を基本とする。この理由により、 CORBA2.0準拠実装を行う言語ならばこのAPIへのアクセスが可能である。たとえばC++かJavaで簡単にクライアントの記述が可能であるが、それはこれらの言語に対しては、フリーのものを含めいくつかのCORBAが実装可能だからである。第 3章では、C++で書かれた小さな例題コードが提供される。しかし第 4章と 5章では、C++ とJava各々で完全なクライアントの記述法を述べる。

本書と本書で述べるAPIは、 Toolboxの VDM-SL版とVDM++版の双方へ適用される。ほんのわずかなケースにおける、 APIの VDM-SLと VDM++間での相違については、はっきりとAPIの定義に述べられている。このマニュアル中での“module”は一般的に、VDM-SLにおけるモジュールまたはVDM++におけるクラスを示している。

2 CORBA - 基本原則

CORBA の本旨は **オブジェクトの分散**である。クライアントプロセスは、ネットワーク上にローカルあるいはリモートに配置された別々のサーバープロセスで取り扱われ物理的に保存されたオブジェクトに対して、状態を再現し、それにアクセスし、場合によっては修正したりすることができる。クライアントはサーバー中に保存されたオブジェクトに対して “ハンドル” を持ち、あたかも分散オブジェクトがクライアントのアドレス空間に配置されているかのように、 このハンドルをメソッドの呼出しに用いる。CORBA 標準仕様では、メソッドの発動方法や異なるオブジェクト間での値のやり取りの方法と同様に、どのように分散オブジェクトに対するハンドルを獲得できるかについての仕様を定めている。

CORBA は単にオブジェクト分散に対する標準であるため、CORBA サーバーおよびクライアントを記述するためにはCORBA の実装 (いわゆる *ORB*) が必要である。現在においてCORBA 実装は、異なるプラットフォームや言語の多くに対して利用可能である。

2.1 IDL

サーバーで動作しCORBA で公開されるオブジェクトは、インターフェイス定義言語（IDL）を用いて記述される。IDL は、実装向き言語でプラットフォームに中立にインターフェイスを記述するための、オブジェクト指向言語である。CORBA インターフェイスを持つツールを提供するベンダーは、顧客にそのツールと共にIDL 記述を配布することでこのインターフェイスの存在を知らせている。IDL の文法は [4] に記述されている。

クライアント実装を行うとき、IDL 記述はIDL コンパイラ (選択されたCORBA 実装に伴って提供される) により、任意に選択された実装言語にマップされる。IDL 記述から生成されるコードはコンパイルされ、サーバーのCORBA インターフェイスを用いることができた実行可能なクライアントと共にリンクされる。

3 VDM Toolbox API（アプリケーションインターフェイス）

VDM Toolbox のCORBA インターフェイスは、2つの IDL ファイル `corba_api.idl` と `metaiv_idl.idl` で記述されていて、VDM Toolbox と共に分散配布される。1番目のファイルでは実際の VDM Toolbox のインターフェイスが記述され、2番目のファイルではクライアントと VDM Toolbox 間でやり取りされるさまざまな VDM 値のインターフェイスが記述されている。以下で両ファイルの詳細を述べ、第 6 章でこれらインターフェイスの参照マニュアルを提供する。

3.1 API ツールの IDL 記述

VDM Toolbox の API は、クライアントプロセスからアクセス可能な多くの様々なオブジェクト (IDL におけるインターフェイス) から構成される。オブジェクトである `VDMApplication` メインの入口であり、ここから API の他のインターフェイスすべての利用が可能である。このオブジェクトは VDM Toolbox へのクライアントハンドルであり、API の他の任意の機能の利用に先立ち構築されている必要がある。第 4 章と第 5 章で、C++と Java 各々において VDM Toolbox に対するこのハンドルをどう獲得するかを述べる。

`VDMApplication` インターフェイスを図 1 に示す。メソッド `Register` と `Unregister` は、サーバー上での処理を登録および解除するために、クライアントによって用いられる。`VDMApplication` インターフェイスはさらにたくさんの、他のインターフェイスをリターン値として戻すメソッドから構成されている。たとえば VDM Toolbox で最新プロジェクトを構成しようと望むなら、そのプロジェクトインターフェイスに対するハンドル、これは以下で述べる `VDMProject` インターフェイスのハンドルである、を得るため `GetProject` を用いるべきである。加えて、インターフェイスの `Tool` 属性は、サーバーとして用いるツール型、つまりクライアントが VDM-SL と VDM++ Toolbox のどちらに接続されているか、を決定するのに用いることができる。

ToolboxAPI::VDMApplication
Tool: ToolType
Register ()
Unregister(in VDM::ClientID id)
GetProject()
GetInterpreter()
GetCodeGenerator()
GetParser()
GetTypeChecker()
GetPrettyPrinter()
GetErrorHandler()
GetModuleRepos()
GetVDMFactory()
PushTag(in VDM::ClientID id)
DestroyTag(in VDM::ClientID id)

図 1: VDMApplication インターフェイス

3.1.1 VDMProject

VDMProject インターフェイスを図 ¯ 2 に示す。このインターフェイスを用いて、VDM Toolbox の最新プロジェクトへのアクセスと修正が可能となる。GetFiles と GetModules は、最新プロジェクトのファイル名とモジュール名の列を (パラメーターを通して) 返す。AddFile と RemoveFile はプロジェクト構成に用いられる。

3.1.2 VDMModuleRepos

VDMModuleRepos インターフェイスを図 ¯ 3 に示す。

VDMModuleRepos インターフェイスは既知のモジュールやクラスにおける追加情報獲得に用いられる。FilesOfModule は特定モジュールのファイルを返し、 一方、 Status は任意モジュールの最新状態を取り出しユーザーインターフェイス中で S、 T、 C、 P という指示子で表示する。残る 4 つのメソッドは、VDM++ Toolbox のみから利用できる。これらはクラスの継承や接続といった関係の問合

ToolboxAPI::VDMProject
New()
Open(in FileName name)
Save()
SaveAs(in FileName name)
GetModules(out ModuleList modules)
GetFiles(out FileList files)
AddFile(in FileName name)
RemoveFile(in FileName name)

図 2: VDMProject インターフェイス

ToolboxAPI::VDMModuleRepos
FilesOfModule(out FileList files, in ModuleName name)
Status(out ModuleStatus state, in ModuleName name)
SuperClasses(out ClassList classes, in ClassName name)
SubClasses(out ClassList classes, in ClassName name)
Uses(out ClassList classes, in ClassName name)
UsedBy(out ClassList classes, in ClassName name)

図 3: VDMModuleRepos インターフェイス

せに用いる。これらのメソッドは、クラス相互の参照を調べるのと同様に、スーパークラスやサブクラスを調べるためにも用いるとよい。

ただしこの IDL 記述は VDM++と VDM-SL Toolbox の両者に共通するため、*ModuleName* や *ModuleList* を定義で用いる場合は、(VDM-SL における) モジュール と (VDM++における) クラスの両方に適用されるものとなることに注意したい。しかし *ClassName* や *ClassList* が明白に用いられていれば、適用は VDM++のみに限定される。

3.1.3 VDMParser

VDMParser インターフェイスを図 ¯ 4 に示す。このインターフェイスを用いて、単一ファイルまたは一連のファイル群に対して VDM Toolbox 構文解析を行う

ToolboxAPI::VDMParser
Parse(in FileName name) ParseList(in FileList names)

図 4: VDMParser インターフェイス

ことができる。後者の一連のファイル群は、手作業で一覧を構築する代わりに、VDMProject::GetFiles の呼び出しを行うことで通常は得られる。

ファイルの構文解析中にエラーが起きる場合は、検出されたエラーを記述する詳細情報を得るために続けて VDMErrors インターフェイス (第 ¯ 3.1.5 章参照) に問合せを行うことができる。

型チェックツール、コード生成ツール、清書印刷ツール (VDMCodeGenerator、VDMTypeChecker、VDMPrettyPrinter) に対するインターフェイスの構造は VDMParser と非常によく似ているが、唯一の違いは、これらのインターフェイスはクライアントが読み込み修正できるたくさんの種々の属性を持つということだ。この属性の設定は、特定のインターフェイスの機能を制御する。これらの 3 つのインターフェイスについてはここでは詳しく述べないが、個々の属性の更なる詳細と説明は第 ¯ 6 章の IDL 記述を参照とする。

3.1.4 VDMInterpreter

VDMInterpreter インターフェイスを図 ¯ 5 に示す。このインターフェイスはインタープリタを使用して、VDM 式の評価とデバッグをし、仕様中の関数や演算を起動する。EvalExpression(client_id, expr) を呼び出すと、文字列引数である *expr* 内の式が評価され、クライアントに結果が返される。結果は、第 ¯ 3.2 章に述べる VDM 値 Generic として表わされる。たとえば、

ToolboxAPI::VDMInterpreter
DynTypeCheck: boolean DynInvCheck: boolean DynPreCheck: boolean DynPostCheck: boolean PPOfValues: boolean Verbose: boolean Debug: boolean
Initialize () EvalExpression (in VDM::ClientID id, in string expr) Apply (in VDM::ClientID id, in string f, in VDM::VDMSequence arg) EvalCmd (in string cmd) SetBreakPointByPos (in string file, in long line, in long col) SetBreakPointByName (in string mod, in string func) DeleteBreakPoint (in long num) StartDebugging (in VDM::ClientID id, in string expr) VDM::VDMTuple DebugStep (in VDM::ClientID id) VDM::VDMTuple DebugStepIn (in VDM::ClientID id) VDM::VDMTuple DebugSingleStep (in VDM::ClientID id) VDM::VDMTuple DebugContinue (in VDM::ClientID id)

図 5: VDMInterpreter インターフェイス

```
EvalExpression(client_id, "[e | e in set {1,...,20}
            & exists1 x in set {2,...,e} & e mod x = 0 ] ")
```

これは、1から20の間のすべての素数からなる列をもった Generic を返す。列からすべての素数を抽出するかわりに、(VDM においては) より効率的な関数である Primes を指定し、インターフェイスの Apply メソッドを通して発動することもできる：

```
Apply(client_id, "Primes", s)
```

ここでの s は関数の引数である。Apply もまたある Generic に含まれる VDM 値と同様に、与えられた引数に対して関数を適用した結果を戻す。(この例題は、

Apply を用いたときの関数に対して引数をどのように渡すかについて、実際には多少単純化している。）Apply の正確な使用法は、 C++ クライアントに対しては第 ¯ 4.2.7 章に、Java クライアントに対しては第 ¯ 5.2.6 章に、述べる。

この例題では、整数列の構築にインタープリタ利用が便利である:

```
s = EvalExpression(client_id, "[e|e in set {1,...,20}]")
```

さらに戻り値 s を Apply の引数として用いる。この代わりに、手作業で列を構築することも可能ではある。

すでに言及した関数とは別に、インタープリタインターフェイスはクライアントから修正が可能なたくさんの属性（ブール値）をもつ。これら属性の設定がインタープリタの動作を制御する。最初の 5 つの属性 (DynTypeCheck,DynInvTheck, DynPreCheck, DynPostCheck, PPOfValues) は、このインタープリタのオプションに相当し、VDM Toolbox のユーザーインターフェイスから設定が可能である。これらは、不変数、事前条件、事後条件等の動的な型チェックといった場面の制御を行う。残る 2 つの属性、 Verbose と Debug は、API がインタープリタを使用する方法を制御する。Verbose は、インタープリタを使用した結果をそのまま VDM Toolbox のユーザーインターフェイスに反映すべきかどうか、の制御を行う。Verbose が偽であれば、ユーザーインターフェイスに結果を反映させることなしに、クライアントはインタープリタを “silently” に使用することになる。Debug 属性は、仕様のブレイクポイントが評価中に生かされるかどうかを制御する。Debug が真と設定されていれば、評価はそれぞれのブレイクポイントで一時停止し、ユーザーは仕様をデバッグできる。この場合、 Apply や EvalExpression への呼び出しは、ユーザーのデバッグ終了までは戻らない。

SetBreakPointByPos と SetBreakPointByName のメソッドを用いて、ブレイクポイントを設定することができる。先のメソッドはファイルと位置 (行、列) を引数にとる一方、後者はモジュール名称と関数名称を求める。両メソッド共に設定されたブレイクポイントの数を返す。この数はブレイクポイントを削除するときに再び利用できる (DeleteBreakPoint)。デバッグはその後の StartDebugging の呼び出しで始まる。このメソッドは引数に ClientID と式 (列) をとる。StartDebugging は、評価が終了するかブレイクポイントに出会うと戻る。VDMTuple を戻すが、評価状態 (<BREAKPOINT>、 <INTERRUPT>、 <SUCCESS> 、 <ERROR>のいずれか) と、 <SUCCESS>の場合ならば MetaIV 値となる評価結果が含まれる。DebugStep、DebugStepIn、 DebugSingleStep、DebugContinue のメソッドは仕様を通して用いることができる。

２つの関数を含むモジュール A を仮定する:

```
module A
  ...
  functions
     foo: nat -> nat
     foo (a) == a + 1;

     bar: nat -> nat
     bar (b) = foo (b)
  ...
```

関数 foo に対してブレイクポイントを設定するために、 SetBreakPointByName ("A", "foo") を用いることもできる。StartDebugging (id, "A`bar(1)) の呼び出しはその後、bar における foo (b) の呼び出しに出会った後に戻される。結果は mk_(<BREAKPOINT>,<<UNDEFINED>>) となるである。DebugContinue の呼び出しは評価を続け、 mk_(<SUCCESS>, 2) が返されるであろう。

3.1.5 VDMErrors

ToolboxAPI::VDMErrors
NumErr: short NumWarn: short
GetErrors(out ErrorList err) GetWarnings(out ErrorList err)

図 6: VDMErrors インターフェイス

VDMErrors インターフェイスを図 6 に示す。このインターフェイスの状態は、構文解析、型チェック、コード生成、清書、の最中にエラーが起きた場合に、更新される。属性 n_err と n_warn を通してエラーと警告の数を求めるためには、このインターフェイスを使おう。この２つのメソッドが、詳細な情報を得るためのエラーや警告の列を返す。

3.2 VDM値のIDL記述

VDM Toolboxのインターフェイスの説明を行ったので、次はどのようにVDM値がAPIを通りぬけるか、つまりどのようにしてVDM値はVDM Toolboxからクライアントに渡されるのか、また逆はどうかといったことに話を移す。

与えられている例題コードはC++で記述されている。Java構文については第5章を参照する。

すでに述べたが、VDMInterpreterインターフェイスのEvalExpressionメソッドは評価の結果を1つのVDM値として返し、Applyメソッドは、関数または操作に対する引数の一覧と同じように、VDM値のVDMの列を引数とする。このようなVDM値を操作する方法については第6.2章に記述されていて、IDLファイルであるmetaiv_idl.idlにも記されている。IDLインターフェイスの構造は、VDM C++ Libraryの構造に([1]に記されているとおり)可能な限り近づけてある。VDM C++ライブラリの各々のクラスは、IDL記述中の1つのインターフェイス(同じ名前のもの)に一致する。図7では、すべての具体的なVDM値が同じスーパークラスGenericから継承されているという事実が描かれる。図では有効なVDM値のほんの一部だけが示されているということに注意を与えておこう。

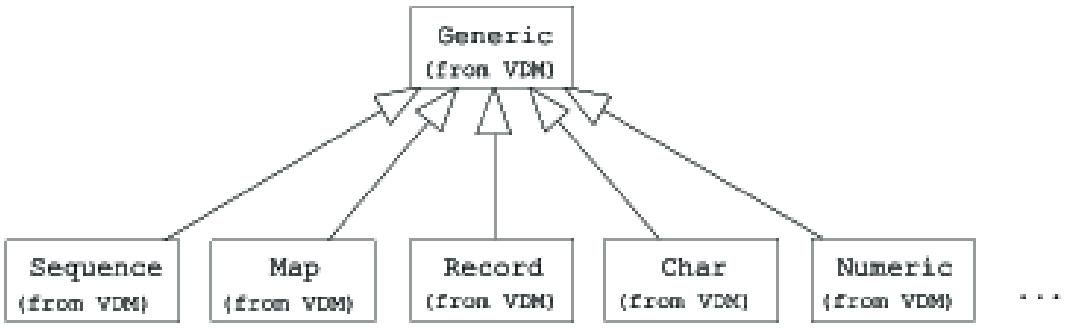

図7: VDM値の継承構造

以下は、インタープリタより返されたVDM値の内容をどう読み込むかの例題である:

```
01  VDM::VDMGeneric_var g;
02  g = interp->EvalExpression(client_id,
```

```
                                    "{ e |-> 2**e | e in set {1,...,16}}");
03  if(!g->IsMap()){
04    // エラーを示す...
05    ...
06  }
07  else{
08    VDM::VDMMap_var m;
09    m = VDM::VDMMap::_narrow(g);
10    VDM::VDMGeneric_var iter;
11    for(int i = m->First(iter); i; i = m->Next(iter)){
12      VDM::VDMGeneric_var rng = m->Apply(client_id, iter);
13      cout << iter->ToAscii() << "-->" << rng->ToAscii() << "\n";
14      iter->Destroy();
15      rng->Destroy();
16    }
17  }
18  g->Destroy();
```

インタープリタは VDMGeneric における評価結果を返す。このため、変数 g は VDM::Generic_var[1] として宣言され、クライアントで使用される特別な _var 型の記述に対して EvalExpression からの結果保持のために用いられる。インタープリタで評価される式は写像内包であり; したがって g に含まれる値は Map 型であるとすべきであろう。Generic インターフェイスの IsMap メソッドは、これが本当に 3 行目に示されるような場合であるのかをチェックするために用いることができる。g が Map 型でなければエラー信号が伝えられる。そうでない場合 Generic の Map 型への変換はうまくいく。オブジェクト参照をキャスト (または narrow) するには、 ORB 実装により供給された_narrow メソッドを用いる。9 行目で VDMGeneric から VDMMap へ制限する方法を示す。VDMMap 型のオブジェクト参照である m と共に、 Map インターフェイスのすべてのメソッドが現在利用可能である。First と Next を用いることで、マップ (11 行目) の定義域を通しての繰り返しが可能であり、 Apply (12 行目) を用いればマップ中のキーとなる値を取り出すことができる。これらのメソッドへの同時アクセスは *one* クライアントのみとすべき、と覚えておきたい。同時に使用したクライアントは、 VDMMap に含まれる全値を得ることができないかもしれない。

[1] クライアントで使用される特別な _var 型の記述、また CORBA オブジェクト参照の使用、について第 4 章を参照のこと。

VDM 値の印刷を提供するため、 Generic インターフェイスは (したがって他のすべての VDM 値は) VDM 値の ASCII 表現を返す ascii メソッドを準備している。13 行目で、このメソッドがマップ m の各要素の印刷に用いられる。

まとめ: この簡単な例題で、インタープリタを用いてマップを構築するが、このマップを通して繰り返すための VDMMap のさまざまなメソッドを使用して連続して印刷が行われる。上記の例題の出力は次の通り:

```
1-->2
2-->4
3-->8
...（短縮のため行を削除）...
16-->65536
```

3.2.1 分散オブジェクトとしての VDM 値

VDM 値がサーバーからクライアントに渡されるとき、常にサーバー内の分散オブジェクトの “ハンドル” あるいはオブジェクト参照として渡される。つまり、クライアントに使用される実際の VDM 値は、実は VDM Toolbox によって操作されたサーバーのアドレス空間に含まれるものである。こうした理由から、クライアントが保持する **すべての** VDM 値は、クライアントが値をもう使用しないというときにはサーバー中で明示的に解放されなければならない。クライアントは VDM 値の Destroy メソッドを呼び出すことでこれを行う。上記例題の 14 から 15 行目で、m の定義域の各要素を表すのに用いられたオブジェクト iter と 、m の値域の相当する要素を保持するために用いられた rng は、破棄される。最後に、18 行目でインタープリタにより生成された VDM 値 g は破棄される。クライアントが値をもう必要としなくなったとき、この方法で破棄がなされないとしたら、 その後に VDM Toolbox で破棄されることはない。この結果、VDM Toolbox 処理ではメモリー使用量が増大する。

そのオブジェクトに対して Destroy メソッドを呼ぶといった明白なオブジェクト破棄の代わりに、PushTag と DestroyTag という VDMApplication インターフェイスの 2 つのメソッドを用いる方法がある。PushTag を呼ぶとタグを 1 つ生成するのでそれを内部タグスタックに押し込む。タグスタック最上位のタグが、VDM Toolbox で連続して作成される全オブジェクトのタグ付けに用いられる。DestroyTag を呼び出すたびにタグスタック最上位のタグが取り出されて、この

値でタグ付けされた各オブジェクトに対して破棄の呼出しがなされる。結果として、PushTag に対する最後の呼出しで生成されたすべてのオブジェクトは破棄される。PushTag と DestroyTag の組み合わせを用いることで、前の例題は次のように読める:

```
01  app->PushTag(client_id);
02  VDM::VDMGeneric_var g;
03  g = interp->EvalExpression(client_id,
                               "{ e |-> 2**e | e in set {1,...,16}}");
04  if(!g->IsMap()){
05    // エラーを示す...
06    ...
07  }
08  else{
09    VDM::VDMMap_var m;
10    m = VDM::VDMMap::_narrow(g);
12    VDM::VDMGeneric_var iter;
12    for(int i = m->First(iter); i; i = m->Next(iter)){
13      VDM::VDMGeneric_var rng = m->Apply(client_id, iter);
14      cout << iter->ToAscii() << "-->" << rng->ToAscii() << "\n";
15    }
16  }
17  app->DestroyTag(client_id);
                  // 最後の PushTag() 以降に生成された全オブジェクトが
                  // ここで破棄される。
```

PushTag と DestroyTag に対する呼び出しは、任意の深さの入れ子にされて DestroyTag 呼び出しの全数が PushTag の呼び出しの全数を超えない限りは、任意の深さの入れ子にできることは覚えておきたい。

3.2.2 インタープリタから戻ってきた値の利用

上記の例題では、インタープリタによって生成されたマップの定義域と値域を印刷するために ToAscii メソッドを用いた。ASCII 表現に対するものとして、

GetValue メソッドを通しての値が利用できる。たとえば以下のコードでは、列の要素１つ１つを二乗して結果を印刷出力する：

```
01  app->PushTag(client_id);
02  g = interp->EvalExpression(client_id,
                               "[ e | e in set {1,...,10}]");
03  if(!g->IsSequence()){
04    exit(-1);
05  }
06  else{
07    VDM::VDMSequence_var s = VDM::VDMSequence::_narrow(g);
08    for(int i = 1; i <= s->Length(); i++){
09      VDM::VDMGeneric_var e = s->Index(i);
10      if(e->IsNumeric()){
11        VDM::VDMNumeric_var ii = VDM::VDMNumeric::_narrow(e);
12        cout << ii->GetValue() * ii->GetValue() << " ";
13      }
14    }
15  }
16  app->DestroyTag(client_id);
```

この例題でも PushTag と DestroyTag のメソッドが用いられているが、クライアントにより使用された値はクライアントがもうそれらを必要としなくなったときに VDM Toolbox の中で解放される、ことが確実に遂行されるためであることに注意しよう。これ以降は、すべてのコード例が PushTag と DestroyTag の呼び出しで “wrapped”であると仮定し、明示的に Destroy を呼出すことはしない。

3.2.3 クライアントの VDM 値構築

これまで用いてきた VDM 値は、すべてインタープリタが生成しクライアントへ返したものであった。しかしながら、クライアントは VDMFactory インターフェイスを用いれば VDM 値を直接構築できる。以下の例題では、VDMFactory インターフェイスに対するハンドルを獲得して数値集合を構築する：

```
VDM::VDMFactory_var fact = app->GetVDMFactory();
```

```
VDM::VDMSet_var s = fact->MkSet(client_id);
VDM::VDMNumeric_var elem;
for(int j=0; j<20; j++){
  elem = fact->MkNumeric(client_id, j);
  s->Insert(elem);
}
```

この集合に挿入される各数値と同様に、factory interfaceが集合全体の構築に用いられることを覚えておこう。さらにfactoryで構築された各数値は、集合に挿入された後に破棄されることも覚えておきたい。なぜならこれら要素はVDMSequence、VDMSet、VDMMap、VDMRecord、VDMTupleといった合成型に挿入されるとき、暗黙にコピーされて用いられるからである。

3.2.4 分散されたVDM値の“実数型”VDM C++値への変換

クライアントはインタープリタからいつVDM値を受け取るのかに留意することが重要であり、これは単にVDM Toolbox内のVDM値に対するハンドルを保持するということである。クライアントがVDM値に対するメソッドを1つ発動するたび、この呼び出しがVDM Toolboxに分散されたVDM値に対しての仲介となる。こういった理由から、たとえばクライアントがインタープリタから戻された長い列に対して繰り返し処理を行う場合、VDM Toolboxは列中の各要素ごとに呼び出されている。もちろんこの方法が特別に効率的だということではない。クライアントに保持されるVDM値に対してより効率的なアクセスを行うために、corba_client.hファイル中に宣言されたGetCPPValue関数を用いることができる。この関数は分散されたVDM値をバイト構成のIDL列に変換するものであり、もしmetaiv.hを含めてVDMライブラリである[1]とリンクするまらば、順番に正しいVDM C++値に変換されることが可能だ。使いやすさを優先すれば、分散されたVDM値からVDM C++値への変換はcorba_client.hで利用できる。単にGetCPPValueを呼び出し、それを引数として変換したいと望むオブジェクト参照に渡せばよい：

```
#include "metaiv.h"

g = interp->EvalExpression(client_id, "[ e | e in set {1,...,10}]");
Generic cpp_g;  // 一般的なC++値   cpp_g = GetCPPValue(g);
```

```
// ここで C++ 値である cpp_g はクライアントプロセスに局所的であり、
// 効率的なアクセスが行える。
```

VDM C++ 値からオブジェクト参照への変換は、関数 FromCPPValue を通して可能であり、これは記述と同様 corba_client.h に宣言もなされている。

Java クライアントは上記の変換を手作業で行う必要があり、クライアント [2] のコンパイルおよび実行時には、クラスパスにVDM Java ライブラリを含めなければならない。例題として、付録 ¯ A.2 の Java プログラムに含まれる関数 EchoPrimes2 を参照しよう。

3.3 例外の取り扱い

IDL インターフェイスである metaiv_idl.idl では、CORBA の 2 つの例外 APIError と VDMError を宣言する。この 2 種の例外は、サーバー処理で何か具合の悪いことが起きた場合に、クライアント処理に信号を送るために用いられる。VDMError が VDM 値使用中のエラー信号伝達に専念する一方で、APIError は VDM Toolbox の API (corba_api.idl) の使用中に起きるかもしれないエラー信号伝達に用いられる。APIError の内容は何が失敗したかを記述する簡単なメッセージ文字列であるが、 VDMError はエラーを示す整数を保持する。この整数の意味は第 ¯ 6.3 章で示す。

付録 ¯ A.1 と ¯ A.2 の例題では、C++と Java それぞれの標準的な例外の取扱いを利用し、クライアントがこのような例外をどう取り扱うことができるかを示す。

4 C++ クライアントの記述

ここでは、Windows または Linux 上で C++ を使ってクライアントを記述する方法について述べよう。

4.1 CORBA 実装の選択

この例題で使用する CORBA 実装 は CORBA 2 準拠 ORB *omniORB3* である。実装は *AT&T* **ケンブリッジ研究所**により開発されたもので、 GNU 一般公有使用許諾の条件や状況下において自由に利用できる。IDL から C++への完全マッピングを実装する omniORB3 によって、いくつかのプラットフォームと C++コンパイラーがサポートされている。ORB は以下のサイトからダウンロードできる:

```
http://www.uk.research.att.com/omniORB/omniORB.html
```

また Windows 2000/XP/Vista や Linux 2.4 または 2.6 を含む様々な手近のプラットフォームに対しては、C++の純粋ソースコードやプレコンパイルされたものが利用できる。特殊なプラットフォームで分散物が利用できない場合は、ソースコードをダウンロードすることで実行ファイルやライブラリ類を構築することが可能である。

クライアントの実装のためには、Win32 か Linux のどちらか向けの onmiORB3 配布をダウンロードし (圧縮を解凍して) インストールしなければならない。omniORB をインストールしたら、すぐにシステムパスの環境変数に omniORB のバイナリディレクトリへの絶対パスを追加すべきである。このディレクトリは様々な CORBA ツール (実体のための IDL コンパイラ) を含む。Windows 向けには、実行時のクライアント実装に用いられるライブラリ群も含まれている。プリコンパイルされた配布をダウンロードする場合なら、必要なものはこれですべてである。これ以外の場合は、配布されたもののコンパイルを成功させるために omniORB のインストール取扱い説明書を参考にする必要がある。

omniORB の代用としては、たとえば **オブジェクト指向構想**による *Orbacus* や Sun's Java IDL からの *idlj* である。OMG CORBA 2.x 仕様や IIOP を実装する ORB であればどのようなものであれ、 VDM Toolbox API と互換性をもつべきであるが、今までこのような検査はなされていない。

Orbacus は IDL から Java や C++ への完全なマッピングを 実装しているため、この ORB はクライアントを Java で記述したい場合に 1 つの選択肢となる。

```
http://www.ooc.com/ob/
```

Java IDL comes with an IDL to Java compiler and is available at Java IDL は 1 つの IDL と共に Java コンパイラとなり、次で利用可能である

```
http://java.sun.com
```

しかしながら、VDM Toolbox と共に配布されるクライアント例題 (付録 ¯ A.1 に一覧がある) は omniORB3 に特有であるため、 OmniBroker が用いられるのであればクライアント例題の多少の修正が必要である。

4.2 クライアントの実装

この章では、C++で記述されたクライアントから VDM Toolbox を使用する例題を通して、詳細を見ていこう。以下は完全な例題からの抜粋であり、付録 ¯ A.1 で全体を見ることができる。

4.2.1 CORBA サービスの初期化

クライアントから VDM Toolbox のアクセスを可能とするためには、基礎となる CORBA 実装の初期化がまずなされる必要がある。様々な CORBA 実装は必ずしも同じように初期化されるわけではないため、ここで述べる CORBA 初期化は omniORB に限ったものであることに注意して欲しい。主なアプリケーションオブジェクトの初期化と獲得は、簡単に利用するため `corba_client.cc` に実装されていて、 `corba_client.h` をクライアント実装に含むことで利用可能となる。異なる CORBA 実装上のクライアントを実装したいと望むのであれば、 `corba_client.cc` の内容を移植することはそれほど難しくはないであろう。VDM Toolbox 配布の `api/corba` サブディレクトリ中に、`corba_client.cc` と `corba_client.h` が見つけられるはずである。

CORBA サービスの初期化に必要なすべては次の通りである:

```
#include "corba_client.h"

main(int argc, char *argv[])
{
  init_corba(argc, argv);
  ...
}
```

4.2.2 アプリケーションオブジェクトの獲得

VDMApplication CORBA オブジェクトに対する CORBA-参照を保持するため、もっとも簡単な方法は get_app メソッドを用いることで、上記の corba_client.h ファイルに見つけることができる。実装は omniORB-特化であるため、任意の ORB に対しては動かない可能性がある。そのため、COS NamingService と文字列化参照によるサポートも行われる。

COS NamingService とは標準化された CORBA オブジェクトサービスで、オブジェクトの実体とその名称を扱うために用いられる。これは CORBA オブジェクトに対して、ディレクトリ階層に保存された名称をマップする。文字列化されたオブジェクト参照と違い、クライアントはたとえ VDM Toolbox とファイルシステムを共有していなくともオブジェクトに対してアクセスが可能である。したがってこれを用いることで柔軟性が得られる。オブジェクト管理グループ (http://www.omg.org) により、使用が推奨されているものである。

COS NamingService と CORBA オブジェクトサービスについてのより詳しい情報は、OMG CORBA ホームページ上にある (http://www.corba.org)。

VDM-SL あるいは VDM++-Toolbox を始動するとき、 COS NameService が動作しているかどうかチェックされる。ORB は設定ファイルを探そうとする。設定ファイルの配置場所は、 OMNIORB_CONFIG 環境変数 (Windows を使用している場合にはレジストリの対処について omniORB-説明書 を参照) を用いて指定できる。典型的な omniORB.cfg ファイルとして、以下のエントリーを含む:

```
ORBInitialHost gandalf
ORBInitialPort 2809
```

これはポート 2809 の gandalf というホスト上で、NamingService が動作していることを意味している。

omniORB はこのような NameService (omniORB-配布の一部として omniNames という名の実行形式が含まれるはずである) を提供するが、事実上 omniORB.cfg ファイルを用いて omniORB に情報を与えているという限りにおいて、他の CORBA-準拠 NameService の使用も可能である。更なる詳細については omniORB 説明書を参照のこと。VDM-SL Toolbox は、 VDMApplication である VDMApplication オブジェクトを名称 SL_TOOLBOX にバインドする一方で、VDM++ Toolbox のアプリケーションオブジェクトでは名称 PP_TOOLBOX が VDMApplication として使用される。これによりクライアントはオブジェクトを区別することが可能となり、したがって各 Toolbox 実体の同時実行に問題はない。

ただし同じ Toolbox の 2 実体を実行させないことに気をつけるべきで、なぜなら先に始動した VDMApplication オブジェクトしかクライアントからアクセスできないことになるからだ。同じ VDMApplication オブジェクトを使用しての 2 クライアント以上の実行は全く問題ないが、それらは互いに影響し合うということを覚えておこう。

VDM Toolbox に対する主導権を獲得するためのもう 1 つのアプローチ- VDMApplication CORBA オブジェクト - では、クライアントに文字列化オブジェクト参照 (最新の VDMToolbox で生成される) を読み込ませ、これを CORBA オブジェクト参照に変換する。すべての ORB 実装は 2 つの関数 object_to_string と string_to_object を実装する必要があり、これらはオブジェクト参照をコード化および非コード化するのに用いられる。VDM Toolbox は 、object_to_string を用いてアプリケーションオブジェクトを文字列にコード化し、この文字列をファイルに書き出す。したがってクライアントはこのファイルを読み込んだ後に、 string_to_object を用いて文字列をオブジェクト参照に変換しなければならない。VDM Toolbox により生成されたファイルは、VDM-SL Toolbox では vdmref.ior、 VDM++ Toolbox では vppref.ior という名称が与えられる。そして VDM_OBJECT_LOCATION 環境変数によって指定された場所に書き込まれる。環境変数が設定されていない場合には、 VDM Toolbox が Unix 上で実行されているならばユーザーのホームディレクトリ ($HOME で指定されている) のルートに配置され、Windows 上で実行されているならばユーザーのプロファイルディレクトリ (%USERPROFILE%で指定されている) へ配置される。

クライアントからアプリケーションオブジェクトを獲得する最も簡単な方法は、corba_client.h に宣言されている get_app を用いることである。

```
main(int argc, char *argv[])
{
 ...
 /* toolType は SL_TOOLBOX か PP_TOOLBOX に設定する */
 ToolType toolType = ...;
 VDMApplication_var app;
 get_app(app, NULL, toolType);
 ...
```

この関数は最初に COS NameService は動いているかどうかチェックし、さらに、VDMApplicationであればSL_TOOLBOX またVDMApplication であればPP_TOOLBOX (どちらかは toolType フラグに依存する) という名のオブジェクトが存在するかどうかをチェックする。NameService を通してこのオブジェクトを見つけることができない場合は自動的に、VDMApplication オブジェクトへの IOR 参照を含むファイルを探し出そうとする。get_app の呼出しを行った後は、変数 app が VDM Toolbox の主ハンドルとなる。

クライアントはサーバーに対し何らかの呼び出しを行う前に、サーバー内に自身を登録しておかなければならない。同様に、終了時には登録解除を行う必要がある。これは VDMApplication クラスの Register と Unregister メソッドを呼出して行う。

```
client_id = app->Register();
...
app->Unregister(client_id);
```

さてこれで実行中の VDM Toolbox のサービスにアクセスできる。

4.2.3 C++でのオブジェクト参照

C++ では、IDL 記述のオブジェクトインターフェイスのためのハンドルが **オブジェクト参照**に含まれる。オブジェクト参照にはインターフェイス名称に _var をつけた名前である。この種のオブジェクト参照は **オブジェクト参照変数**[2]と呼ばれ

[2]オブジェクト参照にはもっと簡単な形式である、 _ptr オブジェクト参照、での利用もある。オブジェクト参照の 2 つの型の違いの詳細は、 [3] と [4] で参照するものとする。ほとんどの場合、 _var オブジェクト参照だけの利用で十分である。

る。たとえば VDMApplication_var は (適切に初期化されている場合)、サーバーの VDMApplication インターフェイスに対するハンドルとなる。インターフェイスの操作は、 _var オブジェクト参照で矢印 “arrow” (->) を用いて呼び込まれる、たとえば VDMApplication インターフェイス app の GetProject メソッドを呼出すのには app->GetProject() 、という具合である。

4.2.4 最新プロジェクトの設定

以下のコード行では、 VDM Toolbox の VDMProject インターフェイスに対するハンドルを獲得し、このインターフェイスの New メソッドと AddFile メソッドを使う。結果として、VDM Toolbox のこのプロジェクトは単一ファイル sort.vdm を含む構成となる。この方法でプロジェクトに追加されるファイルは、VDM Toolbox が始動したと同じディレクトリに配置されなければならない。そうでない場合、絶対パスの与えられたファイル名である必要がある。クライアントが存在しないファイルを追加しようとすると、サーバーはエラーを示して APIError 型の例外処理を実行する。これら例外処理については第 ¯ 3.3 章に記述されている。

```
VDMProject_var prj = app->GetProject();
prj->New();  // 新規プロジェクト
prj->AddFile("sort.vdm");
```

4.2.5 構文解析ツールの使用

クライアントが構文解析ツールを使用するために、 VDMParser インターフェイスへのハンドルを獲得する必要があり、あるファイルを構文分析する場合はこのインターフェイスの Parse メソッドを呼び、唯一の引数にファイル名を指定する。たとえば

```
VDMParser_var parser = app->GetParser();
parser->Parse("sort.vdm");
```

これで sort.vdm ファイルを構文解析する。

代わりに VDMProject インターフェイスを用いれば、最新プロジェクトに構成されたファイルの一覧の取得や、この各ファイルの構文解析が可能となる:

```
FileList_var fl;
prj->GetFiles(fl);

for(int i=0; i<fl->length(); i++){
  cout << (char *)fl[i] << "...Parsing...";
  if(parser->Parse(fl[i]))
    cout << "done.\n";
  else
    cout << "error.\n";
}
```

この例題では、API のいくつかの重要な面を表わしている。初期設定として fl をファイル一覧と宣言し、最新プロジェクトのファイル一覧取り出しに GetFiles を用いる。FileList 型は無限の文字列列と定義され、39 ページに示される。結果として、ファイルの一覧である fl は、CORBA 仕様 [4] により提示された IDL 列のすべてのメソッドを持つ。IDL 列の長さは length メソッドを通して入手でき、個々の要素には C++の通常の列と同様に添え字を付けることができる。

まとめ: 上記のコード行は最新プロジェクトのファイル一覧を取り出し、その一覧を繰り返し、各項目について各ファイルを構文解析するための Parse メソッド呼出しを行う。Parse はファイルの構文解析が成功したことを知らせるブール値を返すことに注目しよう。ファイル一覧を繰り返すことでプロジェクトの全ファイルを構文解析する方法は、実は必要以上の複雑な作業となる。このかわりに ParseList メソッドを用いることができる:

```
FileList_var fl;
prj->GetFiles(fl);
parser->ParseList(fl);
```

4.2.6 型チェックツールの使用

型チェックツールのインターフェイスは構文解析ツールのインターフェイスと似ている。クライアントが利用したり修正したりできるたくさんの属性をもつ。属

性の読取りと修正は、たとえば次のように行うことができる:

```
// DefTypeCheck の値を得る:
int dtc = tpck->DefTypeCheck();

// ExtendedTypeCheck の値を true に設定する
tpck->ExtendedTypeCheck(true);
```

もちろんここでの tpck は、型チェックツールインターフェイスに対する有効なハンドルである。

4.2.7 インタープリタの利用

以下の例題では、インタープリタに任意のVDM式を評価させるために、インタープリタインターフェイスの EvalExpression をどのように利用できるかを示す。

```
VDMInterpreter_var interp = app->GetInterpreter();
VDM::Generic_var g;

g = interp->EvalExpression(client_id, "[e|e in set {1,...,20} & \
  exists1 x in set {2,...,e} & e mod x = 0 ]");

if(g->IsSequence())
  cout << "All primes below 20:\n" << g->ascii() << "\n";
```

EvalExpression に渡される文字列が評価され、評価結果は VDM::Generic における1つのVDM値として返される。これは後に Apply 呼び出しで利用されたり、あるいは第 3.2 章に記述されたVDM値のインターフェイスにより提供されるメソッド群によって、読取／修正を行うことができる。行の最後にあるバックスラッシュの中に EvalExpression への呼出しが置かれるが、これはC++構文の一部である。VDM-SL 式を含む文字列は行替え (\n) を含まないことを示すために用いられる。以下の例題では、VDM Toolbox にすでに読み込まれているVDM仕様の関数をどう使用するかを表す:

```
interp->Init();
g = interp->EvalExpression(client_id, "MergeSort([6,4,9,7,3,42])");
```

仕様のどんな関数でも呼出しが行われる前に、インタープリタ初期化の確認が不可欠であることに注意しよう。

EvalExpression の代わりとなるのが Apply メソッドの使用であり、これは引数として適用する関数または操作の名称をあて、関数やメソッドに対しては引数の列をあてる。以下の例題では MergeSort を用いてソートされた整数の VDM 列を生成する:

```
VDMFactory_var fact = app->GetVDMFactory();

VDM::Sequence_var list = fact->MkSequence(client_id);
VDM::Int_var elem;
for(int j=0; j<20; j++){
  elem = fact->MkInt(client_id, j);
  list->ImpPrepend(elem);
}
```

結果の列は list となり、整数 19 から減じて 0 までを含む。VDMFactory インターフェイスの使用によって、クライアント側で VDM 値がどう構築されるか注意しよう。

Apply を通して MergeSort を呼び出すために、引数の一覧を構築しなければならない。Apply を通して呼び出される関数に対する引数は、 VDMSequence に含まれる。呼び出したい関数は唯 1 つの引数をとり、それは構築したばかりの整数の列である:

```
VDM::Sequence_var arg_l = fact->MkSequence(client_id);
arg_l->ImpAppend(list);
```

ここで MergeSort は以下のように適用される:

```
g = interp->Apply(client_id, "MergeSort", arg_l);
```

もちろんここでのインタープリタは、すでに初期化されていなければならない。引数一覧である arg_l もまた factory interface を用いて構築される。

4.2.8 例題に付加される解釈

ここまでで付録 ¯ A.1 の例題の大半を取上げた。さらにこの例題で対象としているのは、エラーの詳細情報が API を通してどのように検索できるか、どのように個々のモジュール状態の追加情報を得るか、ということである。ここでは例題の詳細に踏みこみこむことはしないが、付録 ¯ A.1 のソースコードやコメントと共に、更なる情報として IDL 記述のインターフェイスである VDMErrors と VDMModuleRepos を参照しよう。

4.3 クライアントのコンパイル

client_example.cc ファイルのコンパイルを成功させるために、以下の各要求が満たされなければならない:

- omniORB はうまくインストールされていなければならない。バイナリー配布が特殊なプラットホームで利用できないという場合には、同様にコンパイルがなされなければならない。さらに、PATH 環境変数は omniORB のバイナリーディレクトリを指定していることが必要である。
- $TOOLBOX/api/corba に以下のファイルが、存在していなければならない (ここで$TOOLBOX は、Toolbox がインストールされたディレクトリを表している)。
  - client_example.cc
  - corba_client.h, corba_client.cc
  - corba_api.idl, metaiv_idl.idl
  - Makefile, Makefile.nm
- VDM Toolbox には VDM C++ ライブラリ、つまりインクルードファイル metaiv.h と、ライブラリ libvdm.a (Unix) または vdm.lib (Windows)、が含まれる必要がある。

例題をコンパイルするため、簡単に Makefile が利用できる。Linux では Makefile と共に make を実行させ、 Windows では Makefile.nm と共に nmake を用いる。

omniORB と VDM Toolbox のそれぞれのインストールディレクトリを指定するため、make ファイルのマクロ `OMNIDIR` と `TBDIR` を修正する必要がある。

win32 下の Microsoft's Foundation Classes を使用したい場合は、 MFC ライブラリが静的にリンクされるべきであることを覚えておこう。

4.3.1 サポートされるコンパイラ

付録 ¯ A.1 にあるクライアント例題は、Microsoft Windows 2000/XP/Vista 上では Microsoft Visual C++ 2005 SP1 を用いて、Linux 上では GNU gcc 3 または 4 を用いて、コンパイルおよびテストされている。

4.4 クライアントの実行

クライアント例題を実行する前に、 サーバーとして使用されるべき VDM Toolbox が実行中であることを確認しておく必要がある。client where to look for the `[vdm|vpp]ref.ior` file. クライアントに `[vdm|vpp]ref.ior` ファイルを探すための場所を伝えるのに、`VDM_OBJECT_LOCATION` 環境変数を用いること。

5 Javaクライアントの記述

5.1 CORBA 実装の選択

Java 1.3 API には org.omg.CORBA と呼ばれるパッケージが含まれ、OMG CORBA API から Java プログラム言語へのマッピングを提供している。このパッケージは ORB クラスを含むが、プログラマが十分な機能をもつ Object Request Broker として使用できるように実装を行ったものだ。

以下の例題ではこの CORBA 実装を用いる。

CORBA の実装に加えて、ユーザーはすでに述べた IDL モデル:corba_api.idl と metaiv_idl.idl にアクセスする必要がある。これらは Java のパッケージやクラスに翻訳されていて、クラスパスに ToolboxAPI.jar ファイルを含めれば使用できる。このファイルは Toolbox 配布の一部であり、api/corba サブディレクトリに置かれている。

次の３つのパッケージが含まれる:

- jp.co.csk.vdm.toolbox.api.corba.VDM このパッケージは metaiv_idl.idl で定義された VDM モジュールを含む。その結果、各 VDM 値に対して１つの Java インターフェイスを含む。
- jp.co.csk.vdm.toolbox.api.corba.ToolboxAPI このパッケージは corba_api.idl からのインターフェイスを含む。
- jp.co.csk.vdm.toolbox.api このパッケージは ToolboxClient というクラス１つのみを含む。VDM Toolbox CORBA API を通して、クライアントアプリケーションを VDM Toolbox に接続するためのメソッドを実装する。

３つのパッケージすべては、*javadoc* プログラムで生成された HTML 文書で説明がなされている。ToolboxAPI.jar ファイルと HTML 文書は、両方とも VDM Toolbox で配布される。

Java 1.3 に続く CORBA 実装を使用しない場合でも、自分で IDL ファイルを Java に翻訳する必要がある。ToolboxAPI.jar のファイルは idltojava コンパイラ (Java Developer Connection からダウンロード可能) を用いて生成される:

http://developer.java.sun.com. Sun JDK 1.3を使用しているならば、実行可能形式の idlj が配布に含まれている。これがJavaコンパイラに対応するSUN IDLで、 Javaのスタブとスケルトンを生成する。

5.2 クライアントの実装

この章では、Javaで記述されたクライアントからどのようにVDM Toolboxを使用するかを示す例題を通して、詳細を見ていこう。以下は完全な例題からの抜粋で示すが、全体は付録 ¯ A.2で見ることができる。

5.2.1 CORBA サービスのインポート

クライアントプログラムは、 上記の3つのパッケージと共に org.omg.CORBA パッケージをインポートすることから始めるとよいだろう:

```
import org.omg.CORBA.*;

import jp.co.csk.vdm.toolbox.api.ToolboxClient;
import jp.co.csk.vdm.toolbox.api.corba.ToolboxAPI.*;
import jp.co.csk.vdm.toolbox.api.corba.VDM.*;
```

5.2.2 アプリケーションオブジェクトの獲得

C++実装においてと同様に、VDM Toolboxに対するメインハンドル－VDMApplication CORBAオブジェクト－獲得のために用いられるアプローチは、クライアントにCOS NamingServiceからの参照を決断させるか、文字列化されたオブジェクト参照を読み込みこれをCORBAオブジェクト参照に変換するか、である。

VDMApplication CORBAオブジェクトに対するCORBA-参照を簡単に手に入れるために可能性が最も高い方法として、 前述の ToolboxClient.java ファイルに見る getVDMApplication メソッドの利用がある。

VDM-SL あるいは VDM++ Toolbox を始めるとき、 COS NameService が動作しているかどうかがチェックされる。ORB が設定ファイルを探そうとする。このファイルの配置位置は OMNIORB_CONFIG 環境変数を用いて指定することができる (Windows 使用の場合に、このためどのようにレジストリを使用するのかは omniORB-説明書を参照のこと)。典型とされる omniORB.cfg ファイルでは、以下のエントリが含まれる:

```
ORBInitialHost gandalf
ORBInitialPort 2809
```

これはポート 2809 の gandalf と呼ばれるホスト上で、 NamingService が動いていることを意味している。omniORB はこのような NameService (omniORB-配布の一部として、omniNames と呼ばれる実行形式があるはずである) を提供しているが、 omniORB.cfg ファイルを用いて omniORB に知らせている限りであれば他のどのような CORBA-compliant NameService でも実際上用いることが可能だ。さらに詳しくは omniORB 説明書を参照のこと。Toolbox は、Sun JDK1.3 からもまた tnameserv-NamingService を用いてテストを行ってきている。クライアントアプリケーションに対しては、 NameService をどこで見つけることができたかを知らせる必要が出てくるであろう。また、これをコマンドラインパラメーター -ORBInitialPort <port> -ORBInitialHost <host>で行うことも、直接ソースコードに相当するプロパティを設定することも、できる。

```
Properties props = new Properties ();
props.put ("org.omg.CORBA.ORBInitialHost", "gandalf");
props.put ("org.omg.CORBA.ORBInitialPort", 2809);
orb = ORB.init (args, props);
```

VDM-SL Toolbox はその VDMApplication オブジェクトに、VDMApplication の中の SL_TOOLBOX という名を付け、一方で VDM++ Toolbox のアプリケーションオブジェクトには、VDMApplication の中の PP_TOOLBOX という名称を用いる。これによりクライアントはオブジェクトを区別することができるため、各 Toolbox のインスタンスを実行させることについての問題はなくなる。

以下のコードは VDMApplication-オブジェクト を NameService から決定するのに用いられる:

```
org.omg.CORBA.Object obj =
   orb.resolve_initial_references ("NameService");
NamingContext ctx = NamingContextHelper.narrow (obj);

NameComponent nc = null;

if (toolType == ToolType.SL_TOOLBOX)
   nc = new NameComponent ("SL_TOOLBOX", "VDMApplication");
else
   nc = new NameComponent ("PP_TOOLBOX", "VDMApplication");

NameComponent[] name = {nc};

org.omg.CORBA.Object obj = // 完全修飾のクラスパスを使用しよう!
   ctx.resolve (name);
VDMApplication app = VDMApplicationHelper.narrow (obj);
```

同じ Toolbox の２つのインスタンスを実行させないよう気をつけるべきで、なぜなら先に始動した VDMApplication オブジェクトしかクライアントからアクセスできないことになるからである。同じ VDMApplication オブジェクトを使用しての２クライアント数以上の実行は全く問題ないが、それらが互いに影響しあうということを覚えておこう。

getVDMApplication メソッドが NamingService を配置できない場合、文字列参照ファイルを使用して VDMApplication-参照の決定を試みる、。すべての ORB 実装では２つの関数 object_to_string と string_to_object を実装する必要があり、オブジェクト参照をコード化および非コード化するのに用いられる。VDM Toolbox は object_to_string を使用してアプリケーションオブジェクトを文字列としてエンコードし、この文字列をファイルに書き出す。したがってクライアントはこのファイルを読み込んだ後に、 string_to_object を用いて文字列をオブジェクト参照に変換しなければならない。VDM Toolbox により生成されたファイルは vdmref.ior あるいは vppref.ior という名称が与えられ、VDM_OBJECT_LOCATION 環境変数によって指定された場所に書き込まれる。環境変数が設定されていない場合に、VDM Toolbox が Unix 上で実行されているならばホームディレクトリ ($HOME で指定されている) のルートに配置され、Windows 上で実行されているならばユーザーのプロファイルディレクトリ (%USERPROFILE%で指定されている) に配置される。

ToolboxClient クラスの readRefFile メソッドは、Toolbox により生成されたこの vdmref.ior あるいは vppref.ior ファイルを読み込むために getVDMApplication で使用される。Toolbox クライアントクラスの getVDMApplication メソッドを呼び出すことで、オブジェクト参照文字列で示された Toolbox に対して接続を構築することができる。このメソッドは CORBA VDMApplication オブジェクトへのオブジェクト参照を返してくれる。

```
VDMApplication app = ToolboxClient.getVDMApplication(args,ref);
```

getVDMApplication の呼び出しの後、変数 app が VDM Toolbox に対する主ハンドルとなる。

クライアントはサーバーに向けての呼び出しを行う前に、サーバー中に自身を登録しなければならない。同様にして、終了時には自身を登録解除しなければならない。これは VDMApplication クラスの Register および Unregister メソッドの呼び出しを行うことでなされる。

```
short client_id = app.Register();
...
...
app.Unregister(client_id);
```

ここで、実行中の VDM Toolbox のサービスにアクセスする立場にたつ。

5.2.3 最新プロジェクトの設定

以下のコード行では、VDM Toolbox の VDMProject インターフェイスに対するハンドルを獲得して、このインターフェイスの New および AddFile メソッドを使用している。その結果として VDM Toolbox のプロジェクトは唯 1 つのファイル sort.vdm を含むように構成されている。VDM Toolbox was started. この方法でプロジェクトに追加されるファイルは、 VDM Toolbox が始動した同じディレクトリに配置されていなければならない。そうでない場合、ファイル名は絶対パスと共に与えられなければならない。クライアントが存在しないファイルを追加しようとすると、サーバーはエラーを示す APIError タイプの例外処理を実行しようとする。例外処理については 第 ¯ 3.3 章に記述されている。

```
VDMProject prj = app.GetProject();
prj.New();
prj.AddFile("sort.vdm");
```

5.2.4 構文解析ツールの利用

クライアントから構文解析ツールを使用するためには、VDMParserインターフェイスに対するハンドルを獲得しなければならず、ファイルを構文解析するためには、そのファイル名を唯1つの引数として与えてこのインターフェイスのParseメソッドを呼び出すのである。つまり、

```
VDMParser parser = app.GetParser();
parser.Parse("sort.vdm");
```

これでファイルsort.vdmを構文解析する。

替わりとしてカレントプロジェクトのために構成されたファイルの一覧を得るためにVDMProjectインターフェイスを使用し、その後この一覧の各ファイルを構文解析することもできる:

```
FileListHolder fl = new FileListHolder();
int count = prj.GetFiles(fl);
String flist[] = fl.value;

for(int i=0; i<flist.length; i++){
   System.out.println("...Parsing" + flist[i] + "...");
   if(parser.Parse(flist[i]))
        System.out.println("done.");
   else
        System.out.println("error.");
}
```

この例題ではAPIのいくつかの重要な面を表している。初期処理としてflをファイル一覧として宣言し、最新プロジェクトのファイル一覧を取り出しGetFiles

を用いる。APIのIDL記述をみれば、 FileList 型は無限の文字列列として定義されている。結果としてファイル一覧 fl は、CORBA仕様 [4] により提出されたIDL列のすべてのメソッドをもつ。IDL列の長さは length メソッドを通して入手でき、個々の要素にJavaの通常の列と同様に添え字を付けることができる。

さらにこの例題で示されるように、Javaで状態を手渡す出力入出力のパラメーターに対するサポートとして、付加的な "holder" クラスの使用が必要となる。これらのクラスは、 org.omg.CORBA パッケージの基本IDLデータ型のすべてに対してできる上、typedefで定義する型以外の型を定義したすべての名前つきユーザーに対して生成される。

ユーザー定義のIDL型に対しては、その型のマップ（Java）名に対して Holder を付けることで holder クラス名がつくられる。各 holder クラスは パブリックな実体要素、 value、をもち、これは型付きの値である。

まとめ: 上記のコード行は最新プロジェクトのファイル一覧を取り出しその一覧を繰り返して、各項目について各ファイルを構文解析するための Parse メソッド呼出しを行う。Parse は、ファイルの構文解析の成功を知らせるブール値を返すことに注目しよう。ファイル一覧を繰り返すことでプロジェクトのファイルを構文解析する方法は、実は必要以上の複雑な作業となる。この代わりに ParseList メソッドを用いることができる:

```
FileListHolder fl = new FileListHolder();
int count = prj.GetFiles(fl);
String flist[] = fl.value;
parser.ParseList(flist);
```

5.2.5 型チェックツールの使用

型チェックツールのインターフェイスは構文解析のインターフェイスと似ている。クライアントが利用したり修正したりできるたくさんの属性をもつ。属性の読取と修正はたとえば次のように行うことができる:

```
// DefTypeCheck の値を得る:
boolean dtc = tpck.DefTypeCheck();
```

```
// ExtendedTypeCheck の値を true に設定する
tpck.ExtendedTypeCheck(true);
```

もちろんここでの tpck は、型チェックツールインターフェイスに対する有効なハンドルである。

5.2.6 インタープリタの利用

以下の例題では、インタープリタに任意のVDM式を評価させるために、インタープリタインターフェイスの EvalExpression をどのように利用できるかを示す。

```
VDMInterpreter interp = app.GetInterpreter();
Generic g;
g = interp.EvalExpression(client_id,
        "[e|e in set {1,...,20} & \
              exists1 x in set {2,...,e} & e mod x = 0]");

if(g.IsSequence()){
     System.out.println("All primes below 20": " + g.ascii());
}
```

EvalExpression に渡される文字列が評価され、評価結果はある Generic における1つのVDM値として返される。これは後に Apply 呼び出しで利用されたり、あるいは第 3.2 章で記述されたVDM値のインターフェイスにより提供されるメソッド群によって、読取／修正を行うことができる。

以下の例題では、VDM仕様の関数をどう使用するかを表す:

```
interp.Init();
g = interp.EvalExpression(client_id, "MergeSort([6,4,9,7,3,42])");
```

仕様のどのような関数でも呼出しが行われる前は、インタープリタ初期化の確認が不可欠である。

EvalExpressionの代わりとなるのがApplyメソッドの使用であり、これは引数として適用する関数または操作の名称をあて、関数やメソッドに対しては引数の列をあてる。以下の例題ではMergeSortを用いてソートされた整数のVDM列を生成する:

```
VDMFactory fact = app.GetVDMFactory();
Sequence list = fact.MkSequence(client_id);
Numeric elem;
for(int j=0; j<20; j++){
    elem = fact.MkNumeric(client_id, j);
    list.ImpPrepend(elem);
}
```

結果となる列listは、整数19から減じて0までを含む。VDMFactoryインターフェイスの使用により、クライアント側でVDM値がどう構築されるのか注意しよう。

Applyを通してMergeSortを呼び出すために、引数の一覧を構築しなければならない。Applyを通して呼び出される関数に対する引数は、Sequenceに含まれる。呼び出したい関数は唯1つの引数を取るが、それは構築したばかりの整数の列である:

```
Sequence arg_l = fact.MkSequence(client_id);
arg_l.ImpAppend(list);
```

ここでMergeSortは以下のように適用される:

```
g = interp->Apply(client_id, "MergeSort", arg_l);
```

もちろんインタープリタは、すでに初期化されていなければならない。引数一覧であるarg_lもまたfactoryインターフェイスを用いて構築されたものであることは、覚えておこう。

最後に、列の要素すべての合計を計算するために、返された列を通して繰り返しをどのように行うか示す:

```
Sequence s = SequenceHelper.narrow(g);
GenericHolder eholder = new GenericHolder();
int sum=0;
for (int ii=s.First(eholder); ii != 0; ii=s.Next(eholder)) {
  Numeric num = NumericHelper.narrow(eholder.value);
  sum = sum + (int) num.GetValue();
}
```

5.2.7 例題に追加される解釈

ここまでで、付録 ¯ A.2 の例題の大半を取り上げた。さらにこの例題で対象としているのは、エラーの詳細情報が API を通してどのように検索できるか、どのように個々のモジュール状態の追加情報を得るか、ということである。その上で、配布された VDM 値を “実数型” VDM Java 値に変換する方法も示す。ここでは例題の詳細にさらに踏み込むことはしないが、付録 ¯ A.2 のソースコードやコメントと共に、更なる情報として IDL 記述のインターフェイスである VDMErrors と VDMModuleRepos を参照しよう。

5.3 クライアントのコンパイル

client_example.java ファイルは、以下のコンパイラを用いてコンパイルされなければならない:

```
jdk1.3
```

次のように記述することで、主プログラムのコンパイルを行うことができる:

```
javac client_example.java
```

CLASSPATH 環境変数が ToolboxAPI.jar ファイルを含むことを確認すること。もし Unix Bourne シェルまたは互換性のあるシェルを用いているならば、次のコマンドが利用できる:

```
CLASSPATH=ToolboxAPI_Library/ToolboxAPI.jar:$CLASSPATH
export CLASSPATH
```

ToolboxAPI_Library を、 ToolboxAPI.jar ファイルがインストールされたディレクトリ名に置きかえること。

Windows ベースのシステムで作業をしている場合は、 Windows コマンドプロンプト内で以下のコマンドを用いることができる:

```
set CLASSPATH=ToolboxAPI_Library/ToolboxAPI.jar;$CLASSPATH
```

Windows に対しては、区切り文字は “:” ではなく “;” を使用しなければならないことに注意しよう。

5.4 クライアントの実行

クライアント例題を実行する前に、サーバーとして使用されるべき VDM Toolbox が実行中であることを確認しておく必要がある。

既定では、例題プログラムは Linux 上で VDM-SL Toolbox と共に実行されていると仮定する。この場合、単に次の実行を行う

```
java client_example.java
```

Windows 上の実行では WIN property を設定しなければならず、 VDM++ Toolbox を用いた実行では、 VDMPP プロパティを設定しなければならない。

```
java -DVDMPP -DWIN client_example
```

6 API 参照ガイド

6.1 Corba API

6.1.1 型

以下の型同義語が定義されている:

名称	同義語の対象
ModuleName	string
ModuleList	sequence<ModuleName>
ClassName	string
ClassList	sequence<ClassName>
FileName	string
FileList	sequence<FileName>
ErrorList	sequence<Error>

以下の列挙が定義されている

```
enum ToolType {SL_TOOLBOX, PP_TOOLBOX};
```

以下の構造が定義されている:

6.1.2 エラー構造

変数	意味
FileName fname	エラー／警告の検出されたファイル名
unsigned short line	エラー／警告の行番号
unsigned short col	エラー／警告の列番号
string msg	エラー／警告のテキスト

6.1.3 ModuleStatus 構造

変数	意味
`boolean SyntaxChecked`	モジュール（またはクラス）が構文チェック済みであるかどうかを示す属性
`boolean TypeChecked`	モジュール（またはクラス）が型チェック済みであるかどうかを示す属性
`boolean CodeGenerated`	モジュール（またはクラス）がコード生成済みであるかどうかを示す属性
`boolean PrettyPrinted`	モジュール（またはクラス）が清書済みであるかどうかを示す属性

6.1.4 VDMApplication インターフェイス

名称	説明
`readonly attribute ToolType Tool`	サーバー側 Toolbox に対するツールの型を返す。
`ClientID Register()`	unique client id を返す。クライアントは何かの API を実行する前にサーバー側に自身を登録するべきである。
`void Unregister(ClientID id)`	クライアント id と結びついたリソースを解放する。クライアントが終了するときはこれが呼ばれるべきである。
`VDMProject GetProject()`	現プロジェクトに対するハンドルを返す。
`VDMInterpreter GetInterpreter()`	Toolbox インタープリタに対するハンドルを返す。

名称	説明
VDMCodeGenerator GetCodeGenerator()	Toolbox コード生成ツールに対するハンドルを返す。
VDMParser GetParser()	Toolbox 構文解析に対するハンドルを返す。
VDMTypeChecker GetTypeChecker()	Toolbox 型チェックツールに対するハンドルを返す。
VDMPrettyPrinter GetPrettyPrinter()	Toolbox 清書ツールに対するハンドルを返す。
VDMErrors GetErrorHandler()	Toolbox エラーインターフェイスに対するハンドルを返す。
VDMModuleRepos GetModuleRepos()	Toolbox モジュール (またはクラス) リポジトリに対するハンドルを返す。
VDMFactory GetVDMFactory()	VDM value factory に対するハンドルを返す。 CORBA 2.x では remote object instantiation をサポートしないため、この factory が CORBA VDM オブジェクト (たとえば VDMSequence、VDMToken...) の生成のために用いられる必要がある。
void PushTag(in ClientID id)	クライアント id に対する unique tag を生成し、Toolbox の内部タグスタック上にこれを押し込む。このクライアントに対して Toolbox で生成されたすべてのオブジェクトは、その後このタグでタグ付けされる。
void DestroyTag(in ClientID id) raises APIError	ClientID に対するタグスタック上の一番上のタグを取り出し、この値でタグ付けされたすべてのオブジェクトを破壊する。

6.1.5 VDMCodeGenerator インターフェイス

名称	説明
attribute boolean GeneratePosInfo	位置情報の生成を有効化または無効化する。これによって生成コード中の構築物すべては仕様にまでさかのぼることを許可される。 既定は false。
enum LanguageType CPP, JAVA	コード生成ツールの可能な対象言語
boolean GenerateCode(in ModuleName name, in LanguageType targetLang) raises APIError	モジュール (またはクラス) name に対して C++/Java コード (targetLang flag に従う) を生成する。name が現プロジェクト内で有効なモジュール名またはクラス名でなかった場合、また VDM-SL モジュールに対して Java コードを生成しようとした場合、例外を起こす (Java コード生成は VDM++-クラスに対してのみ利用可能なものであるから)。
boolean GenerateCodeList(in ModuleList names) raises APIError	names 中の各モジュール (またはクラス) 名 に対して C++/Java コードを生成する。 names 中の名称で現プロジェクトで有効なモジュール名やクラス名でないものがあった場合、また VDM-SL モジュールに対して Java コードを生成しようとした場合、例外を起こす。

6.1.6 VDMErrors インターフェイス

名称	説明
readonly attribute unsigned short NumErr	直前の動作で生成されたエラーの数を返す。
readonly attribute unsigned short NumWarn()	直前の動作で生成された警告の数を返す。
unsigned short GetErrors(out ErrorList err)	err 中の直前の動作で生成されたエラーの一覧を返す。
unsigned short GetWarnings(out ErrorList err)	err 中の直前の動作で生成された警告の一覧を返す。

6.1.7 VDMInterpreter インターフェイス

名称	説明
attribute boolean DynTypeCheck	動的型チェックを有効化または無効化する。既定は false。
attribute boolean DynInvCheck	動的不変数チェックを有効化または無効化する。既定は false。真の場合は、 DynTypeCheck 属性が自動的に真に設定される。
attribute boolean DynPreCheck	動的事前条件チェックを有効化または無効化する。既定は false。
attribute boolean DynPostCheck	動的事後条件チェックを有効化または無効化する。既定は false。
attribute boolean PPOfValues	清書機能を有効化または無効化する。既定は true。
attribute boolean Verbose	Toolbox との冗長な対話、つまり API を通して実行された動作結果のインタープリタウィンドウでの繰り返し、を有効化または無効化する。既定は false。

名称	説明
attribute boolean Debug	デバッグモードを有効化または無効化する。このモードでは仕様中のブレイクポイントが生かされる。ブレイクポイントが評価されるときは一時停止を行い、ユーザーは実際上のデバッグを行うためにグラフィカルユーザーインターフェイスと付き合う必要がある。既定は false。
void Initialize()	インタープリタを初期化する。評価の前になされる必要がある。
VDMGeneric EvalExpression(in ClientID id, in string expr) raises APIError	ID id のクライアントの代わりに、expr を評価する。評価結果はメソッドの結果として返ったものだ。Verbose が true であるならば、結果はモニターに投影される。実行時エラーは例外を引き起こす。
VDMGeneric Apply(in ClientID id, in string f, in VDMSequence arg) raises APIError	クライアント id の代わりに、関数（または操作） f を引数 arg に適用する。 関数（または操作）呼び出しの結果は、メソッドの結果として返される。実行時エラーは例外を引き起こす。
void EvalCmd(in string cmd)	インタープリタに直接書き込まれたように、コマンド cmd を評価する。
long SetBreakPointByPos(in string file, in long line, in long col)	指定ファイルの指定位置（行、列）にブレイクポイントを設定し、新しいブレイクポイントの数を返す。例外または -1 の戻り値が、エラーが起きたことを示す (たとえば、ファイルが存在しないまたは指定された行番号が有効でない場合)。

名称	説明
long SetBreakPointByName(in string mod, in string func) raises APIError	指定モジュールの指定関数 (func) にブレイクポイントを設定し、新しいブレイクポイントの数を返す。例外または-1 の戻り値は、エラーが起きたことを示す (たとえばモジュールまたは関数が存在しない場合)。
void DeleteBreakPoint(in long num) raises APIError	ブレイクポイントの削除に使用される。ブレイクポイントで返された数を、１つのパラメーターとして、メソッドを設定する。
VDMTuple StartDebugging (in ClientID id, in string expr) raises APIError	式のデバッグを始める。式の評価が終了した場合、またはブレイクポイントに出会った場合、このメソッドが返る。評価状態 (<BREAKPOINT>、<INTERRUPT>、 <SUCCESS>、<ERROR>、のいずれか) を含める VDMTuple を返すことができて、<SUCCESS>の場合は、結果の評価を表す MetaIV 値も。
VDMTuple DebugStep (in ClientID id) raises APIError	このメソッドは toolbox における step コマンドと同等である。これは次の文を実行しその後中断する。関数や操作の呼び出しには立ち入らない。評価状態 (<BREAKPOINT>、<INTERRUPT>、 <SUCCESS> 、<ERROR>のいずれか) を含む VDMTuple を返し、<SUCCESS> (これは、式がうまく評価できたことを意味する) の場合には、MetaIV 値として評価の結果を返す。

名称	説明
VDMTuple DebugStepIn (in ClientID id) raises APIError	このメソッドはtoolboxにおけるstepinコマンドと同等である。これは次の文を実行しその後中断する。関数や操作の呼び出しにも立ち入る。評価状態 (<BREAKPOINT>、<INTERRUPT>、 <SUCCESS>、<ERROR>のいずれかになり得る) を含むVDMTupleを返し、<SUCCESS> (これは、式がうまく評価できたことを意味する) の場合にはMetaIV値として評価の結果を返す。
VDMTuple DebugSingleStep (in ClientID id) raises APIError	このメソッドはtoolboxにおけるsinglestepコマンドと同等である。これは次の式または文を実行しその後中断する。評価状態 (<BREAKPOINT>、 <INTERRUPT>、<SUCCESS>、<ERROR>のいずれかになり得る) を含むVDMTuple値を返し、<SUCCESS>(これは、式がうまく評価されたことを意味する) の場合は、MetaIV値として評価の結果を返す。
VDMTuple DebugContinue (in ClientID id) raises APIError	このメソッドはtoolboxにおけるcontコマンドと同等である。これはブレイクポイントに出会った後の実行を継続する。評価状態 (<BREAKPOINT>、 <INTERRUPT>、<SUCCESS>、<ERROR>のいずれかになり得る) を含むVDMTuple値を返し、<SUCCESS> (これは、式がうまく評価されたことを意味する) の場合はMetaIV値として評価の結果を返す。

6.1.8 VDMModuleReposインターフェイス

名称	説明
unsigned short FilesOfModule(out FileList files, in ModuleName name)	filesの中のモジュール（またはクラス）nameを含むファイルの名称を引き渡す。フラットモジュールの場合には、１つのモジュール名DefaultModはいくつものファイルに分散されているかもしれないが、これ以外の場合、文字列はきっちり１つの名称から構成されることになる。ファイル数を返す。
void Status(out ModuleStatus state, in ModuleName name) raises APIError	stateにモジュール（またはクラス）nameの状態を引き渡す。nameが現モジュールに存在しない場合は、例外を起こす。
unsigned short SuperClasses(out ClassList classes, in ClassName name) raises APIError	nameクラスのスーパークラスのclasses一覧に引き渡す。VDM++仕様では、 VDM-SL toolboxで呼ばれた場合には例外を起こす。
unsigned short SubClasses(out ClassList classes, in ClassName name) raises APIError	nameクラスのサブクラスのclasses一覧に引き渡す。VDM++仕様では、 VDM-SL toolboxで呼ばれた場合には例外を起こす。
unsigned short Uses(out ClassList classes, in ClassName name) raises APIError	nameクラスで使用されるクラスのclasses一覧に引き渡す。VDM++仕様では、VDM-SL toolboxで呼ばれた場合には例外を起こす。
unsigned short UsedBy(out ClassList classes, in ClassName name) raises APIError	nameクラスを使用するクラスのclasses一覧に引き渡す。VDM++仕様では、 VDM-SL toolboxで呼ばれた場合には例外を起こす。

6.1.9 VDMParser インターフェイス

名称	説明
boolean Parse(in FileName name) raises APIError	name ファイルがうまく構文解析された場合に true を返す; それ以外は false を返し VDMErrors インターフェイスの状態が修正される。このファイルが存在しない場合は例外を起こす。
boolean ParseList(in FileList names) raises APIError	names 内のファイルすべてがうまく構文解析された場合に true を返す。それ以外は false を返し VDMErrors インターフェイスの状態が更新される。いずれのファイルも存在しない場合は例外を起こす。

6.1.10 VDMPrettyPrinter インターフェイス

名称	説明
boolean PrettyPrint(in FileName name) raises APIError	name モジュール（またはクラス）がうまく清書を終えた場合に true を返す; それ以外は false を返して VDMErrors インターフェイスの状態が修正される。このモジュールが存在しない場合は例外を起こす。
boolean PrettyPrintList(in FileList names) raises APIError	names 内のモジュール（またはクラス）すべてがうまく清書を終えた場合に true を返す。それ以外は false を返して VDMErrors インターフェイスの状態が更新される。いずれのモジュール（またはクラス）も存在しない場合は例外を起こす。

6.1.11 VDMProject インターフェイス

名称	説明
`void New()`	新しいプロジェクトを生成する
`void Open(in FileName name) raises APIError`	引数 `FileName` で与えられた名前のプロジェクトを開く
`void Save() raises APIError`	現存の名前を使用しているプロジェクトを保存する。この時点でプロジェクトに名前がない場合には例外を起こす (たとえば新規の場合)
`void SaveAs(in FileName name)`	引数 `FileName` で与えられた名前を使い、プロジェクトを保存する
`unsigned short GetModules(out ModuleList modules)`	現プロジェクトに対し、`modules` 内のモジュール (VDM-SL) の一覧またはクラス (VDM++) の一覧を生成し、モジュール数あるいはクラス数を返す。
`unsigned short GetFiles(out FileList files)`	現プロジェクトに対し、`files` 内のファイルの一覧を生成し、ファイル数を返す。
`void AddFile(in FileName name) raises APIError`	プロジェクトに 1 つファイルを追加する。うまくいかなかった場合は `APIError` を起こす (たとえばファイルが見つからない場合)。
`void RemoveFile(in FileName name) raises APIError`	プロジェクトからファイルを取り除く。うまくいかなかった場合は `APIError` を起こす (たとえばファイルが現プロジェクトにない場合) 。

6.1.12 VDMTypeChecker インターフェイス

名称	説明
attribute boolean DefTypeCheck	型チェックモードが “def” (true) か “pos” (false) かを決定する。既定は “pos”。
attribute boolean ExtendedTypeCheck	拡張型チェックが有効かどうかを決定する。既定は false。
boolean TypeCheck(in ModuleName name) raises APIError	name モジュール（またはクラス）がうまく型チェックされた場合には true を；それ以外は false を返し、VDMErrors インターフェイスの状態は修正される。モジュール（またはクラス）が存在しない場合は、例外を起こす。
boolean TypeCheckList(in ModuleList names) raises APIError	names 内のすべてのモジュール（またはクラス）がうまく型チェックされた場合、true を返す。 それ以外は false を返し、VDMErrors インターフェイスの状態は更新される。いずれのモジュール（またはクラス）も存在しない場合は例外を起こす。

6.2 VDM API

章のタイトルはインターフェイス名で完全に修飾している (つまり VDM::Interface となる) が、以下における実際の説明では簡略のために短縮した名称を用いる。

6.2.1 型

型の同義語は次のように定義される:

名称	同義語の対象
ClientID	short
bytes	sequence<octet>

6.2.2 VDM::VDMGeneric インターフェイス

名称	説明
`string ToAscii()`	オブジェクトの文字列表現を返す。
`boolean IsNil()`	オブジェクトが型 `VDMNil` をもつ場合のみ true を返す。
`boolean IsChar()`	オブジェクトが型 `VDMChar` をもつ場合のみ true を返す。
`boolean IsNumeric()`	オブジェクトが型 `VDMNumeric` をもつ場合のみ true を返す。
`boolean IsQuote()`	オブジェクトが型 `VDMQuote` をもつ場合のみ true を返す。
`boolean IsTuple()`	オブジェクトが型 `Tuple` をもつ場合のみ true を返す。
`boolean IsRecord()`	オブジェクトが型 `Record` をもつ場合のみ true を返す。
`boolean IsSet()`	オブジェクトが型 `Set` をもつ場合のみ true を返す。
`boolean IsMap()`	オブジェクトが型 `Map` をもつ場合のみ true を返す。
`boolean IsText()`	オブジェクトが型 `VDMText` をもつ場合のみ true を返す。
`boolean IsToken()`	オブジェクトが型 `VDMToken` をもつ場合のみ true を返す。
`boolean IsBool()`	オブジェクトが型 `VDMBool` をもつ場合のみ true を返す。
`boolean IsSequence()`	オブジェクトが型 `Sequence` をもつ場合のみ true を返す。
`boolean IsObjectRef()`	オブジェクトが別の VDM オブジェクトの参照である場合のみ true を返す。

名称	説明
`void Destroy() raises APIError`	このメソッドを呼び出すことで、サーバーに対してクライアントがこのオブジェクトをこれ以上使用しないことを示す。それがサーバーオブジェクトに対する最後の参照であったという場合には、付随したリソースはすべて解放されることになる。
`bytes GetCPPValue()`	MetaIV 値のバイナリー表現を返す。この方法でクライアントアプリケーションを VDM ライブラリとリンクさせることで、クライアント側に 'real' MetaIV 値を生成することが可能である。大きな VDM 値を通しての繰り返し行う場合に、これがより効率的なアクセスを提供してくれることになる
`VDMGeneric Clone()`	このメソッドは、このメソッドが起動されたオブジェクトで保持されている値のコピーを返す。

6.2.3 基本 VDM 型

以下のインターフェイスは `VDMGeneric` インターフェイスを拡張する。唯一の違いは、既定のアクセスがあってこの VDM 値に相当する値を返す `GetValue()` メソッドが、追加されていることである。

インターフェイス	GetValue() の戻り値
VDM::VDMBool	boolean
VDM::VDMChar	char
VDM::VDMNumeric	double
VDM::VDMQuote	string
VDM::VDMText	string
VDM::VDMToken	string

The interface VDM::VDMNil は、継承している以上のパブリックメソッドやメンバー変数をもたない。

6.2.4 VDM::VDMMap インターフェイス

このインターフェイスは VDMGeneric を拡張する。

名称	説明
void Insert(in VDMGeneric key, in VDMGeneric val) raises VDMError	マップに対して値 val の新しいキー key を追加する。マップの定義域に key が既に存在する場合は例外を起こす。
void ImpModify(in VDMGeneric key, in VDMGeneric val)	key が値 val をもつようにマップを修正する。
VDMGeneric Apply(in VDMGeneric key) raises VDMError	key に相当する値を返す。マップの定義域に key が存在しない場合は例外を起こす。
void ImpOverride(in VDMMap m)	マップオブジェクト m でこのマップを上書きする。
unsigned long Size()	マップのキー数を返す。
boolean IsEmpty()	マップがキーをもたない場合 true を返す。

名称	説明
Set Dom()	マップの定義域（すべてのキー）を返す。
Set Rng()	マップの値域 (すべての値) を返す。
boolean DomExists(in VDMGeneric g)	マップの定義域に g がある場合のみ true を返す。
void RemElem(in VDMGeneric key) raises VDMError	マップからキー key を取り除く。key がマップの定義域に存在しない場合は例外を起こす。
short First(out VDMGeneric g)	マップの第一キーを g に引き渡す。マップが空でない場合は 1 を、空である場合は 0 を返す。
short Next(out VDMGeneric g)	マップの次のキーを g に引き渡す繰り返し。マップにまだ巡回していないキーが存在する場合に 1 を、繰り返しですべてのキーが引き渡されている場合は 0 を返す。

6.2.5 VDM::VDMRecord インターフェイス

このインターフェイスは Generic を拡張する。

名称	説明
void SetField(in unsigned long i, in VDMGeneric g) raises VDMError	フィールド i が値 g をもつように設定する。i がこのレコードで有効なフィールドでない場合 (つまり値域 $1, \ldots,$ *number of fields* 内にない場合) 例外を起こす。
VDMGeneric GetField(in unsigned long i) raises VDMError	フィールド i の値を返す。i がこのレコードで有効なフィールドでない場合 (つまり値域 $1, \ldots,$ *number of fields* 内にない場合) 例外を起こす。
string GetTag()	このレコードのタグを返す。
boolean Is(in string tag)	tag がこのレコードのタグに一致する場合のみ true を返す。

名称	説明
long Length()	このレコードのフィールド数を返す。

6.2.6 VDM::VDMSequence インターフェイス

このインターフェイスは VDMGeneric を拡張する

名称	説明
VDMGeneric Index(in long i) raises VDMError	列における添え字 i の値を返す。i が有効な添え字でない場合は、例外を起こす。
VDMGeneric Hd() raises VDMError	列の先頭の値を返す。列が空である場合は例外を起こす。
VDMSequence Tl() raises VDMError	この列を変更することなしに、最初の要素を取り除いた部分列を返す。列が空である場合は例外を起こす。
void ImpTl() raises VDMError	この列の先頭を削除する。列が空である場合は例外を起こす。
void RemElem(in long i) raises VDMError	列から添え字が i の要素を削除する。
long Length()	列の長さを返す
boolean GetString(out string s)	列が純粋な文字の連続であるならば、 true を返して、s に相当する文字列を引き渡すが、それ以外は false を返す。
boolean IsEmpty()	列が空である場合のみ true を返す。
void ImpAppend(in VDMGeneric g)	列の最後に値 g を付け加える。
void ImpModify(in long i, in VDMGeneric g) raises VDMError	添え字 i で保存されていた値を g で上書する。i がこの列に対して有効な添え字でない場合 (つまり範囲 $1,\ldots,length\ of\ sequence$ にない場合) 例外を起こす。
void ImpPrepend(in VDMGeneric g)	列の先頭に値 g を追加する。

名称	説明
void ImpConc(in VDMSequence s)	この列の後ろに列 s を連結する。
Set Elems()	列の要素で構成される集合を返す。
short First(out VDMGeneric g)	列の先頭の要素を g で渡す。列が空でなければ 1 を、空であれば 0 を戻り値として返す。
short Next(out VDMGeneric g)	列の次の要素を g で渡す。イテレータでまだ指し示していない要素がある場合は 1 を、すべて指し示した場合は 0 を戻り値として返す。

6.2.7 VDM::VDMSet インターフェイス

このインターフェイスは VDMGeneric を拡張する

名称	説明
void Insert(in VDMGeneric g)	値 g を集合へ挿入する。
unsigned long Card()	集合の要素数を返す。
boolean IsEmpty()	集合が空の場合のみ true を返す。
boolean InSet(in VDMGeneric g)	集合に g がある場合のみ true を返す。
void ImpUnion(in VDMSet s)	s のすべての要素をこの集合に加える。
void ImpIntersect(in VDMSet s)	この集合から s にない要素を削除する。
VDMGeneric GetElem() raises VDMError	集合の任意の要素を返す。集合が空である場合は例外を起こす。
void RemElem(in VDMGeneric g) raises VDMError	集合から要素 g を削除する。g が集合に現れない場合は例外を起こす。
boolean SubSet(in VDMSet s)	s がこの集合の部分集合である場合のみ true を返す。
void ImpDiff(in VDMSet s)	集合 s にも現れる要素は削除することでこの集合を修正する。

名称	説明
short First(out VDMGeneric g)	集合の最初の要素を g に渡す。集合が空でない場合 1 を返し、集合が空である場合 0 を返す。
short Next(out VDMGeneric g)	集合の次の要素を g に引き渡す繰り返し。まだ巡回していない集合の要素がある場合は 1 を返し、すべての要素が繰り返しで引き渡されている場合は 0 が返される。

6.2.8 VDMTuple インターフェイス

このインターフェイスは VDMGeneric を拡張する。

名称	説明
void SetField(in unsigned long i, in VDMGeneric g) raises VDMError	タプルのフィールド i を値 g に設定する。フィールド i が存在しない場合は例外を起こす。
VDMGeneric GetField(in unsigned long i) raises VDMError	フィールド i を返す。フィールド i が存在しない場合は例外を起こす。
unsigned long Length()	タプルのフィールド数を返す。

6.2.9 VDMFactory インターフェイス

名称	説明
VDMNumeric MkNumeric(in ClientID id, in double d)	クライアント id に値 d をもつ VDMNumeric オブジェクトを返す
VDMBool MkBool(in ClientID id, in boolean b);	クライアント id に値 b をもつ VDMBool オブジェクトを返す
VDMNil MkNil(in ClientID id);	クライアント id に VDMNil オブジェクトを返す

名称	説明
VDMQuote MkQuote(in ClientID id, in string s);	クライアント id に値 s をもつ VDMQuote オブジェクトを返す
VDMChar MkChar(in ClientID id, in char c);	クライアント id に値 c をもつ VDMChar オブジェクトを返す
VDMText MkText(in ClientID id, in string s);	クライアント id に値 s をもつ VDMText オブジェクトを返す
VDMToken MkToken(in ClientID id, in string s);	クライアント id に値 s をもつ VDMToken オブジェクトを返す
VDMMap MkMap(in ClientID id);	クライアント id に VDMMap オブジェクトを返す
VDMSequence MkSequence(in ClientID id);	クライアント id に VDMSequence オブジェクトを返す
VDMSet MkSet(in ClientID id)	クライアント id に VDMSet オブジェクトを返す
VDMTuple MkTuple(in ClientID id, in unsigned long length);	クライアント id に構成要素 length をもつ VDMTuple オブジェクトを返す
VDMGeneric FromCPPValue(in ClientID id, in bytes cppvalue)	'real' MetaIV 値をそのバイナリー表現から VDMGeneric に変換する。この関数は GetCPPValue() の '逆' である。

6.3 例外

2つの例外が定義されている:

例外	構成要素
ToolboxAPI::APIError	string msg
VDM::VDMError	short err

VDMError 例外パケットに返される値は、状態コードである。可能な状態コードの一覧とそれらの意味については、以下の通り。

値	説明
1	既に異なる値域値をもち存在するマップにキーを挿入しようとしている
2	定義域にない
4	添え字が範囲外
6	空集合に対して操作
7	集合に存在しない
10	空列に対し Hd() を実行
11	空列に対し Tl() を実行
12	範囲エラー

6.4 C++ API参照

この章では、第 6.1 章と 6.2 章に記述された IDL インターフェイスの翻訳を簡単に述べる。これは omniORB IDL コンパイラ (Version 2.6.1) で生成された翻訳に基づいている。

6.4.1 corba_client.h

ファイル `corba_client.cc` は ORB の初期化を簡単にするために準備されたものである。そのインターフェイスが `corba_client.h` に定義され、ここで一覧にしてある。列挙型 `GetAppReturnCode` は以下のテーブルに記載された値で宣言されている:

値	説明
VDM_OBJECT_LOCATION_NOT_SET	環境変数 VDM_OBJECT_LOCATION が設定されなかった。詳細は関数 GetAppReturnCode を参照。
OBJECT_STRING_NON_EXISTING	VDM Toolbox が実行していなかった
CORBA_SUCCESS	VDM Toolbox との交信が成功
CORBA_ERROR	VDM Toolbox との交信はエラー

定義された関数は以下の通り:

名称	説明
`void init_corba(int argc, char *argv[])`	CORBA ORB と BOA (Basic Object Adapter) を初期化する。何か他の CORBA 関連関数を用いる前に、この関数を呼び出すこと。Object Request Broker と Basic Object Adapter についの更なる情報としては、OMB CORBA ホームページ (http://www.corba.org) から利用できる CORBA 仕様を参照しよう。
`GetAppReturnCode get_app(VDMApplication_var app, char *path [, ToolboxAPI::ToolType toolType])`	CosNamingService からの VDMApplication を解決しようとする。NamingService が実行されていない場合、実行サーバーの ID を得るために 'vdmref.ior' または 'vppref.ior' という名称のファイルを読む。このファイルは、環境変数 VDM_OBJECT_LOCATION によって提示されたディレクトリに置かれている必要がある (自分でパスを準備する場合は別)。The ToolType (ToolboxAPI::SL_TOOLBOX または ToolboxAPI::PP_TOOLBOX はオプション、 SL_TOOLBOX は既定の設定である。戻り値は操作の結果を示す。
`Generic GetCPPValue(VDM::Generic_ptr g_ptr)`	MetaIV-IDL オブジェクト参照を相当する 'real' MetaIV C++ 値に変換する。
`VDM::Generic_ptr FromCPPValue(ClientID id, Generic g, VDMFactory_ptr fact);`	'real' MetaIV C++ 値を、サーバーの呼び出し時に渡すことが可能な VDMGeneric CORBA オブジェクトに変換する。この関数に VDMFactory に対するハンドルを渡さなければいけないことも、覚えておこう。

6.4.2 命名規則

C++において IDL インターフェイス *I* への参照を生成するために、I_var 型の変数が生成されるべきである。

インターフェイス *I* で定義された操作 *O* にアクセスするためには、オブジェクト参照の間接アクセスが用いられる、つまり I_var->M ということ。このような参照は、“in”（値）パラメーターと “out”（結果）パラメーターの両方に対して用いられるべきである。

6.4.3 キャスト操作

各々のインターフェイス *I* に対して、相当する C++ クラス I は_narrow という静的関数を含んでいる。

I::_narrow 関数は、型 CORBA::Object_ptr の引数をとり、I クラスの新しいオブジェクト参照を返す。ORB と交信するかもしれないオブジェクトはいずれも、型 CORBA::Object_ptr をもつ。

引数オブジェクト参照の実際の (実行時の) 型を I に限定することができるならば、 I::_narrow は有効なオブジェクト参照を返す。それ以外は何のオブジェクト参照も返さない。

6.5 Java API

この章では、 6.1 と 6.2 の章で述べられている IDL インターフェイスの Java への翻訳のこと、を簡単に説明する。これは IDL により生成された、Java Compiler (Version 1.3) への翻訳に基づくものである。javadoc で生成された記述は、Toolbox 配布と共に api/corba/javaapi-doc に 含まれていることを覚えておこう。

6.1 と 6.2 の章で述べている各々のインターフェイスに対し、相当する Java クラスが jp.co.csk.vdm.toolbox.api パッケージに存在する。インターフェイスで定義されているメソッドは、それに相当する Java クラス内で同じ名前をもっている。メソッドに対して、“In” パラメーター (値パラメーター) は相当するクラ

スの値として; “out” パラメーター (結果パラメーター) は *holder* オブジェクトとして、渡される – 6.5.2 章を参照。

これらのクラスや以下に述べるクラスに加えて、さらにもう１つクラス-ToolboxClient (これも jp.co.csk.vdm.toolbox.api パッケージに含まれる) がある。このクラスは次の２つのメソッドを提供している

名称	説明
String readRefFile()	[vdm|vpp]ref.ior ファイルの内容を返す。
public VDMApplication getVDMApplication(String[] args, ToolType toolType)	Toolbox に対する接続を (SL_TOOLBOX か PP_TOOLBOX という toolType に従って) 確立する。ORB に対しては、引数 (args) として各々いくつかのコマンドライン引数をとる。

追加として、 API を用いるためには Java Development Kit と共に配布される org.omg.CORBA パッケージを用いるべきであろう。

VDM_OBJECT_LOCATION プロパティは readRefFile で用いるということを覚えておきたいが、Java は通常の環境変数を許していないため、この値は実行時に -D フラグを用いて渡されなければならない。たとえば

```
java -DVDM_OBJECT_LOCATION=/tmp <java class>
```

6.5.1 Helper クラス

6.1 や *C* という名の 6.2 の章に述べられているインターフェイス各々に対して、相当する *CHelper* という名の *Helper* クラスが存在する。プログラマの視点からみると、これらクラスの主なる使用は *narrow* メソッドの提供で、任意の CORBA オブジェクトを *C* クラスのオブジェクトに制限（キャスト）することである。このような制限が可能でない場合は、 例外 org.omg.CORBA.BAD_PARAM が起こされる。

CHelper クラスに対しては、相当する制限関数が次のスキームを用いて宣言できる

```
public static C narrow(org.omg.CORBA.Object that)
```

ここでの C は特定のクラス名称に置き換えられるものである。

以下のクラスに対しての制限メソッドは提供されていないが、有意ではないからである:

ClassList
ErrorList
FileList
ModuleList
ModuleStatus
ToolType
_Error

6.5.2 Holder クラス

6.1 と *C* という名の 6.2 の章で述べたインターフェイスの各々に対して、*CHolder* という名の相当する *Holder* クラスが存在する。これらはメソッドが参照により結果を返すことを許するために用いられ、つまり holder クラスのオブジェクトが引数としてこのようなメソッドに渡されるということであり、このメソッドは結果をそのオブジェクト中に置くのである。

各々の holder クラスはパブリックな実体メンバー、 value、をもち、これは型値である。

これらの holder クラスに加えて、さらに４つの holder クラスが存在する:

例外	value 属性の型
ClassListHolder	String[]
ErrorListHolder	_Error[]
FileListHolder	String[]
ModuleListHolder	String[]

7 推奨文献

CORBA についての更なる情報提供を行うためまたこの標準規格が提供するサービスとして、omniORB3 ユーザーガイド、[3]、に言及するが、これは本題についての強力な導入となる。さらに詳しい情報としては、 CORBA 標準、[4]、の章が選択的に利用可能である。

参考文献

[1] CSK. *The VDM C++ Library*. CSK.

[2] CSK. The vdm++ to java code generator. Tech. rep.

[3] Lo, S.-L., Riddoch, D., and Grisby, D. *The omniORB version 3.0 User's Guide*. AT&T Laboratories Cambridge, May 2000.

[4] *The Common Object Request Broker: Architecture and Specification*. OMG, July 1996.

A 例題プログラム

A.1 C++ クライアントの例題

```
/***
*  * 概要
*  *    このファイルはVDM Toolboxの CORBA API を用いて
*  *    クライアント処理を実装する方法の例題である。
*  *
*  *    このファイルは windows NT/95 上の MS VC++ 6.0  および
*  *    Unix 上の gcc 2.95.2 でコンパイル可能である。
*  *
*  *    98/NT 上で nmake を使用する場合は makefile に Makefile.nm を用い
*  *    Linux 上でコンパイルする場合は Makefile を用いる
*  * ID
*  *    $Id: client_example.cc,v 1.27 2006/02/07 05:14:11 vdmtools Exp $
*  * AUTHOR
*  *    Ole Storm + $Author: vdmtools $
*  * COPYRIGHT
*  *    (C) Kyushu University
***/

#include <iostream>
using namespace std;

#include <string>
// CORBA 初期化 および omniORB4 のための他のこと
#include "corba_client.h"

#ifdef _MSC_VER
#include <direct.h> // getcwd のため
#else
#include <unistd.h> // getcwd のため
#endif // _MSC_VER

char ABS_FILENAME[200];
```

```
VDM::ClientID client_id;

#define ADD_PATH(p,s) strcat(strcpy(ABS_FILENAME, p), s)
#define SORT_NUM 20

void EchoPrimes(int, ToolboxAPI::VDMInterpreter_var,
                ToolboxAPI::VDMApplication_var);
void EchoPrimes2(int, ToolboxAPI::VDMInterpreter_var,
                 ToolboxAPI::VDMApplication_var);
void ListModules(ToolboxAPI::VDMApplication_var app);

int main(int argc, char *argv[])
{
  const char * source_path = getenv("VDM_SOURCE_LOCATION");
  string sdir;
  if( source_path == NULL )
  {
    char buf[1024];
    if( getcwd( buf, sizeof( buf ) ) != NULL )
    {
#ifdef _MSC_VER
      // Unix 上では
      // パス中のバックスラッシュはフォワードスラッシュに変換する。
     for (char* s = buf; s = strchr(s, '￥￥'); s++) {
        *s = '/';
      }
#endif
      sdir = buf;
      for( int i = 0; i < 2; i++ )
      {
        unsigned int index = sdir.find_last_of( '/' );
        if( index == string::npos ) break;
        sdir = sdir.substr( 0, index );
      }
      sdir += "/";
    }
```

```
    source_path = sdir.c_str();

    cerr << "Environment variable VDM_SOURCE_LOCATION not set" << endl;
    cerr << "Default location: " << source_path << endl;
  }

  //  VDM Toolboxに対する主ハンドル:
  ToolboxAPI::VDMApplication_var app;

  // ORBの初期化。これがどのように行われるかについての詳細は
  // corba_client.{h,cc}とomniORB3 ユーザーマニュアルを参考に。
  init_corba(argc, argv);

  // 最後にスタートしたVDMToolboxに対するハンドルを取り出す。
  // ハンドルはVDM Toolboxにより生成された
  // CORBAオブジェクトの文字列表示を通して獲得する。
  // 文字列はファイル名の付いたobject.stringとして書き出され
  // VDM_OBJECT_LOCATIONにより定義されたディレクトリに置かれる。
  // これが設定されない場合、get_app は自動的にホーム（Unix）または
  // プロファイルディレクトリ（Windows NT/95）にファイルを探しに行く。

#ifdef VDMPP
  GetAppReturnCode rt = get_app(app, NULL, ToolboxAPI::PP_TOOLBOX);
#else
  GetAppReturnCode rt = get_app(app, NULL, ToolboxAPI::SL_TOOLBOX);
#endif //VDMPP
  switch(rt){
  case VDM_OBJECT_LOCATION_NOT_SET:
    cerr << "Environment variable VDM_OBJECT_LOCATION not set" << endl;
    exit(0);
  case OBJECT_STRING_NON_EXISTING:
      cerr << "The file " + GetIORFileName() + " could not be located. ¥

               Make sure the Toolbox is running" << endl;
    exit(0);
  case CORBA_ERROR:
```

```
  cerr << "Unable to setup the CORBA environment" << endl;
  exit(0);
case CORBA_SUCCESS:
default:
  break;
}

try{
  // Toolboxにクライアントを登録:
 client_id = app->Register();

  // 現プロジェクトを構築するために、
  // VDMProjectインターフェイスに対するハンドルを最初に獲得:
  ToolboxAPI::VDMProject_var prj = app->GetProject();

  prj->New();  // 新規プロジェクト

  // 必要なファイルを含めるために、プロジェクトを構築する。
  // ファイルはVDM Toolboxがスタートした同じ場所に
  // 配置されなければならない。それ以外の場合は、
  // ファイルに対する絶対パスを用いるべきである
  if(app->Tool() == ToolboxAPI::SL_TOOLBOX)
    {
      prj->AddFile(ADD_PATH(source_path,"examples/sort/sort.vdm"));
    }
  else{
    prj->AddFile(ADD_PATH(source_path,"examples/sort/implsort.vpp"));
    prj->AddFile(ADD_PATH(source_path,"examples/sort/sorter.vpp"));
    prj->AddFile(ADD_PATH(source_path,"examples/sort/explsort.vpp"));
    prj->AddFile(ADD_PATH(source_path,"examples/sort/mergesort.vpp"));
    prj->AddFile(ADD_PATH(source_path,"examples/sort/sortmachine.vpp"));
  }
  // ファイルを構文解析:
  ToolboxAPI::VDMParser_var parser = app->GetParser();
  ToolboxAPI::FileList_var fl;
  prj->GetFiles(fl);
```

```
// ２つの違った方法でファイルを構文解析する。
// 第一の方法ではファイル一覧を１つ１つ検討し
// 各ファイルを個別に構文解析する
// （もちろん、SL_TOOLBOX に対しては構築ファイルが唯１つであるが、
// 説明ではこれで十分であろう）
cout << "Parsing files individually" << endl;
for(unsigned int i=0; i<fl->length(); i++){
  cout << (char *)fl[i] << "...Parsing...";
  if(parser->Parse(fl[i]))
    cout << "done." << endl;
  else
    cout << "error." << endl;
}

// そしてその後すべてのファイルを一度に構文解析:
cout << "¥nParsing entire list...";
parser->ParseList(fl);
cout << "done." << endl;

// 構文解析中にエラーが起きた場合には
// ここで調査:
ToolboxAPI::VDMErrors_var errhandler = app->GetErrorHandler();
// エラーハンドラー
ToolboxAPI::ErrorList_var errs;

// エラー列を取り出す
int nerr = errhandler->GetErrors(errs);
if(nerr){
  // エラーの印刷:
  cout << nerr << " errors:" << endl;
  for(int ierr=0; ierr<nerr; ierr++)
    cout << (char *) errs[ierr].fname << ", "
         << errs[ierr].line << endl
         << (char *) errs[ierr].msg << endl;
```

```
}
// 同様に警告を問い合わせることもできる。

// すべてのモジュールの名称と状態を一覧:
ListModules(app);

// すべてのモジュールの型チェック:
ToolboxAPI::VDMTypeChecker_var tchk = app->GetTypeChecker();
ToolboxAPI::ModuleList_var modules;
prj->GetModules(modules);
cout << "Type checking all modules...";
if(tchk->TypeCheckList(modules))
  cout << "done." << endl;
else
  cout << "errors." << endl;

// すべてのモジュールの最新状態の一覧:
ListModules(app);


// 最後にインタープリターの使用法を示す。

cout << endl << "Interpreter tests:" << endl << endl;
ToolboxAPI::VDMInterpreter_var interp = app->GetInterpreter();


// 素数計算をする関数の呼び出し:
EchoPrimes(20, interp, app);

// 第２の方法として問合せを用いる方法を示す:
// ソートされるべき整数の列を構築する。このために
// VDM値を作り出すためのVDMFactoryに対するハンドルを求める:
VDM::VDMFactory_var fact = app->GetVDMFactory();

app->PushTag(client_id);
// これ以降は生成されたオブジェクトすべてにタグ付けする
```

```
VDM::VDMSequence_var list = fact->MkSequence(client_id);
VDM::VDMNumeric_var elem;
for(int j=0; j<SORT_NUM; j++){
  elem = fact->MkNumeric(client_id, j);
  list->ImpPrepend(elem);
}
cout << "The sequence to be sorted: " << list->ToAscii() << endl;

// 呼び出しのための引数の一覧の構築。これは
// 右並びに全引数を含める VDM::Sequence の構築である:
VDM::VDMSequence_var arg_l = fact->MkSequence(client_id);
arg_l->ImpAppend(list);

// ユーザーインターフェイスでインタープリターを用いる結果を
// 表示するため、Verbose を true に設定:
interp->Verbose(true);
interp->Debug(true);

// 始めにインタープリターを初期化する
interp->Initialize();

VDM::VDMGeneric_var g;
if(app->Tool() == ToolboxAPI::SL_TOOLBOX){
  g = interp->Apply(client_id, "MergeSort", arg_l);
}
else{ // PP_TOOLBOX
  // 最初は、中心となるソートオブジェクの生成:
  interp->EvalCmd("create o := new SortMachine()");

  // 次は、このオブジェクト上で GoSorting メソッドを呼び出し:
  g = interp->Apply(client_id, "o.GoSorting", arg_l);
}

cout << "The sorted sequence: " << g->ToAscii() << endl;
```

```
    // 最後に、返された列を通して列中の全要素の
    // 合計を計算するための繰り返し:

    VDM::VDMSequence_var s = VDM::VDMSequence::_narrow(g);
    int sum=0;
    for(int k=1; k<=s->Length(); k++){
      VDM::VDMNumeric_var n = VDM::VDMNumeric::_narrow(s->Index(k));
      sum += (Int(GetCPPValue(n))).GetValue();
    }
    cout << "The sum of all the elements: " << sum << endl;

    EchoPrimes2(50, interp, app);

    app->DestroyTag(client_id);

    // クライアントの登録を解除:
    app->Unregister(client_id);
  }
  catch(ToolboxAPI::APIError &ex){
    cerr << "Caught API error " << (char *)ex.msg << endl;
  }
  catch(CORBA::COMM_FAILURE &ex) {
    cerr << "Caught system exception COMM_FAILURE, ¥
             unable to contact server"
         << endl;
  }
  catch(omniORB::fatalException& ex) {
    cerr << "Caught omniORB3 fatalException" << endl;
  }

  return 0;
}


void EchoPrimes(int n, ToolboxAPI::VDMInterpreter_var interp,
                ToolboxAPI::VDMApplication_var app)
```

```
  // n以下の素数の列を生成し、
  // それを stdout にそのまま送り返す。
{
  app->PushTag(client_id);

  interp->Initialize ();

  // この VDM::Generic はインタープリターからの結果の保持に
  // 用いられる。
  VDM::VDMGeneric_var g;

  // 20 以下の素数を計算するために EvalExpression を用いる
  char expr[200];
  sprintf(expr, "[e|e in set {1,...,%d} ¥
                 & exists1 x in set {2,...,e} & e mod x = 0 ]", n);
  g = interp->EvalExpression(client_id, expr);
  if(g->IsSequence()){
    cout << "All primes below " << n << ":" << endl << g->ToAscii() << endl;
  }
  VDM::VDMSequence_var s = VDM::VDMSequence::_narrow(g);
  int sum=0;
  for(int k=1; k<=s->Length(); k++){
    VDM::VDMNumeric_var n = VDM::VDMNumeric::_narrow(s->Index(k));
    sum += (Int(GetCPPValue(n))).GetValue();
  }
  cout << "The sum of all the primes: " << sum << endl;
  app->DestroyTag(client_id); // 仕上げ...
}

void EchoPrimes2(int n, ToolboxAPI::VDMInterpreter_var interp,
                 ToolboxAPI::VDMApplication_var app)
  // n以下の素数の列を生成し
  // その列を stdout にそのまま送り返す。
  // 追加として、VDM 値全体を toolbox からクライアント側へ移行し
  // これを metaiv.h に宣言されている"real"の C++値に変換するために
  // どう GetCPPValue を使用できるか、をこの関数が示している
```

```
{
  // この VDM::VDMGeneric はインタープリターからの結果を保持することに
  // 用いる。
  VDM::VDMGeneric_var g;

  // 20 以下の素数を計算するのには EvalExpression を用いる
  char expr[200];
  sprintf(expr, "[e|e in set {1,...,%d} & ¥
                 exists1 x in set {2,...,e} & e mod x = 0 ]", n);
  g = interp->EvalExpression(client_id, expr);

  // VDM::Generic g を"real" metaiv-Sequence 値に
  // 変換する:
  Sequence s(GetCPPValue(g));

  // ここで値全体がクライアント側へ移されたので安全に g を
  // 破棄することができる:
  g->Destroy();

  cout << "All primes below " << n << ":" << endl
       << wstring2string(s.ascii()) << endl;
  int i, sum=0;
  Generic gg;
  for(i = s.First(gg); i; i = s.Next(gg)){
    sum += (int)Real(gg).GetValue();
  }
  cout << "The sum of all the primes: " << sum << endl;
}


void ListModules(ToolboxAPI::VDMApplication_var app)
  // この関数はモジュールとそれらの状態を一覧にする。
{
 // プロジェクトハンドル
  ToolboxAPI::VDMProject_var prj = app->GetProject();
```

```
  // モジュールリポジトリ
  ToolboxAPI::VDMModuleRepos_var repos = app->GetModuleRepos();

  ToolboxAPI::ModuleList_var ml;
  prj->GetModules(ml);
  cout << "Modules:" << endl;
  for(unsigned int i=0; i<ml->length(); i++){
    // この構造はモジュール状態の保持に使用される:
    ToolboxAPI::ModuleStatus stat;
    // i 番目のモジュールの状態を得る
    repos->Status(stat, ml[i]);
    // 状態を印刷する。 0 = none, 1 = OK
    cout << (int) stat.SyntaxChecked
         << (int) stat.TypeChecked
         << (int) stat.CodeGenerated
         << (int) stat.PrettyPrinted
         << " " << (char *)ml[i] << endl;
  }
}
```

A.2 Java クライアントの例題

```
import org.omg.CORBA.*;
import java.io.*;

import jp.co.csk.vdm.toolbox.api.ToolboxClient;
import jp.co.csk.vdm.toolbox.api.corba.ToolboxAPI.*;
import jp.co.csk.vdm.toolbox.api.corba.VDM.*;

public class client_example
{
  private static short client;
  private static VDMApplication app;
```

```
private static final String VdmToolboxHomeWin=
    "C:\\Program Files\\The VDM-SL Toolbox v3.7.1\\examples";
private static final String VppToolboxHomeWin=
    "C:\\Program Files\\The VDM++ Toolbox v6.7.17\\examples";
private static final String VdmToolboxHome=
    "/home/vdm/toolbox/examples/sl";
private static final String VppToolboxHome=
    "/home/vdm/toolbox/examples/pp";

public static void main(String args[])
{
  try {
    //
    // ORB生成
    //

    String os = System.getProperty("os.name", "");
    String ToolboxHome = System.getProperty("TOOLBOXHOME", "");

    if (System.getProperty("VDMPP") == null) {
        app = (new ToolboxClient ()).getVDMApplication(args,
                                                       ToolType.SL_TOOLBOX);
if( 0 == ToolboxHome.length() ) {
          if (os.startsWith("Windows"))
            ToolboxHome = VdmToolboxHomeWin;
          else
            ToolboxHome = VdmToolboxHome;
        }
    }
    else {
        app = (new ToolboxClient ()).getVDMApplication(args,
                                                       ToolType.PP_TOOLBOX);
if( 0 == ToolboxHome.length() ) {
          if (os.startsWith("Windows"))
            ToolboxHome = VppToolboxHomeWin;
```

```
      else
        ToolboxHome = VppToolboxHome;
    }
}

// Toolboxにクライアントを登録:

client = app.Register();

System.out.println ("registered: " + client);

// 最初に、現プロジェクトを構築するために
// VDMProjectインターフェイスに対するハンドルを獲得:

try{
  VDMProject prj = app.GetProject();
  prj.New();

  // 必要なファイルを含めるためにプロジェクトを構築する。
  // ファイルはVDM Toolboxが始動した同じディレクトリに
  // 配置されていなければならない。それ以外の場合は
  // ファイルには絶対パスを使用すべきである


  if(app.Tool() == ToolType.SL_TOOLBOX){
    prj.AddFile(ToolboxHome + "/sort/sort.vdm");
  }
  else{
    prj.AddFile(ToolboxHome + "/sort/implsort.vpp");
    prj.AddFile(ToolboxHome + "/sort/sorter.vpp");
    prj.AddFile(ToolboxHome + "/sort/explsort.vpp");
    prj.AddFile(ToolboxHome + "/sort/mergesort.vpp");
    prj.AddFile(ToolboxHome + "/sort/sortmachine.vpp");
  }

  // ファイルの構文解析:
```

```
VDMParser parser = app.GetParser();
FileListHolder fl = new FileListHolder();
int count = prj.GetFiles(fl);
String flist[] = fl.value;

// ２つの異なる方法でファイルを構文解析する。第一は、
// ファイル一覧を検討し各ファイルを個別に構文解析する。
// （もちろん、SL_TOOLBOX に対しては構築ファイルが唯１つであるが、

// 説明にはこれで十分であろう）

System.out.println("Parsing files individually");
for(int i=0; i<flist.length; i++){
  System.out.println(flist[i]);
  System.out.println("...Parsing...");
  if(parser.Parse(flist[i]))
    System.out.println("done.");
  else
    System.out.println("error.");
}

// そしてその後すべてのファイルを一度に構文解析:

System.out.println("Parsing entire list...");
parser.ParseList(flist);
System.out.println("done.");

// 構文解析中にエラーが起きた場合には
// ここで調査:

// エラーハンドラー

VDMErrors errhandler = app.GetErrorHandler();

ErrorListHolder errs = new ErrorListHolder();
// エラー列を取り出す
```

```
 int nerr = errhandler.GetErrors(errs);
  jp.co.csk.vdm.toolbox.api.corba.ToolboxAPI.Error errlist[] =
    errs.value;
 if(nerr>0){
   // エラーの印刷:
  System.out.println("errors: ");
   for(int i=0; i<errlist.length; i++){
     System.out.println(errlist[i].fname);
     System.out.println(errlist[i].line);
     System.out.println(errlist[i].msg);
   }
 }

 // 同様に警告を問い合わせることもできる。

 // すべてのモジュールの名称と状態を一覧:
ListModules(app);

 // すべてのモジュールの型チェック:

 VDMTypeChecker tchk = app.GetTypeChecker();
 ModuleListHolder moduleholder = new ModuleListHolder();
 prj.GetModules(moduleholder);
 String modules[] = moduleholder.value;
 System.out.println("Type checking all modules...");
 if(tchk.TypeCheckList(modules))
   System.out.println("done.");
 else
   System.out.println("errors.");

 // すべてのモジュールの最新状態の一覧:
ListModules(app);


 // 最後にインタープリターの使用法を示す。
```

```
System.out.println("Interpreter tests:");

VDMInterpreter interp = app.GetInterpreter();


// 素数計算をする関数の呼び出し:
EchoPrimes(20, interp, app);

// 第２の方法として問合せを用いる方法を示す:
// ソートされるべき整数の列を構築する。このために
// VDM値を作り出すためのVDMFactoryに対するハンドルを求める:

VDMFactory fact = app.GetVDMFactory();

app.PushTag(client);
// これ以降は生成されたオブジェクトすべてにタグ付けする

VDMSequence list = fact.MkSequence(client);

VDMNumeric elem;
for(int j=0; j<20; j++){
  elem = fact.MkNumeric(client, j);
  list.ImpPrepend(elem);
}

System.out.println("The sequence to be sorted: " +
                   list.ToAscii());

// 呼び出しのための引数の一覧の構築。これは
// 右並びに全引数を含めるSequenceの構築である:
VDMSequence arg_l = fact.MkSequence(client);

arg_l.ImpAppend(list);

// ユーザーインターフェイスでインタープリターを用いる結果を
// 表示するため、Verbose をtrueに設定:
```

```
 interp.Verbose(true);
 interp.Debug(true);

 // 始めにインタープリターを初期化する
System.out.println("About to initialize the interpreter");
 interp.Initialize();

 VDMGeneric g;
 if(app.Tool() == ToolType.SL_TOOLBOX){
   g = interp.Apply(client, "MergeSort", arg_l);
 }
 else{ // PP_TOOLBOX
   // 最初は、中心となるソートオブジェクの生成:
   interp.EvalCmd("create o := new SortMachine()");

   // 次は、このオブジェクト上で GoSorting メソッドを呼び出し:
  g = interp.Apply(client, "o.GoSorting", arg_l);
 }

 System.out.println("The sorted sequence: " + g.ToAscii());

 // 最後に、返された列挙を通して列挙中の全要素の
 // 合計を計算するための繰り返し:

 VDMSequence s = VDMSequenceHelper.narrow(g);

 VDMGenericHolder eholder = new VDMGenericHolder();

 int sum=0;
 for (int ii=s.First(eholder); ii != 0; ii=s.Next(eholder)) {
   VDMNumeric num = VDMNumericHelper.narrow(eholder.value);
   sum = sum + GetNumeric( num );
 }

 System.out.println("The sum of all the elements: " +  sum);
```

```
      EchoPrimes2(50, interp, app);

      app.DestroyTag(client);

      app.Unregister(client);
      System.exit(0);
    }
    catch(APIError err) {
      System.err.println("API error"+err.getMessage ());
      System.exit(1);
    }
  }
  catch
    (jp.co.csk.vdm.toolbox.api.ToolboxClient.CouldNotResolveObjectException ex)
    {
      System.err.println(ex.getMessage());
      System.exit(1);
    }
  catch(COMM_FAILURE ex) {
    System.err.println(ex.getMessage());
    ex.printStackTrace();
    System.exit(1);
  }
};

public static void ListModules(VDMApplication app){

  try{
    // この関数はモジュールとそれらの状態を一覧にする。
    // プロジェクトハンドル

    VDMProject prj = app.GetProject();

    // モジュールリポジトリ
    VDMModuleRepos repos = app.GetModuleRepos();
```

```
    ModuleListHolder ml = new ModuleListHolder();
    prj.GetModules(ml);
    String mlist[] = ml.value;
    System.out.println("Modules:");

    for(int i=0; i<mlist.length; i++){

      // この構造はモジュール状態の保持に使用される:
     ModuleStatusHolder stateholder = new ModuleStatusHolder();
      // i 番目のモジュールの状態を得る
      repos.Status(stateholder, mlist[i]);
      ModuleStatus stat = stateholder.value;

      // 状態を印刷する。
      System.out.println(mlist[i]);
      System.out.println("SyntaxChecked: " + stat.SyntaxChecked);
      System.out.println("TypeChecked: " + stat.TypeChecked);
      System.out.println("Code generated: " + stat.CodeGenerated);
      System.out.println("PrettyPrinted: " + stat.PrettyPrinted);
    }
  }
  catch(APIError err) {
    System.err.println("API error");
    System.exit(1);
  }
}


public static void EchoPrimes(int n, VDMInterpreter interp,
                              VDMApplication app)
{
  try{
    // n以下の素数の列を生成し
    // それを stdout にそのまま送り返す。
```

```
    app.PushTag(client);

    // この Generic はインタープリターからの結果を保持することに
    // 用いる。
    VDMGeneric g;

    // 20以下の素数を計算するのにはEvalExpressionを用いる

    String expr = "[e|e in set {1,...,"+n+"} &"+
                  " exists1 x in set {2,...,e} & e mod x = 0]";
    g = interp.EvalExpression(client,expr);

    if(g.IsSequence()){
      System.out.println("All primes below " + n + ": " +
                         g.ToAscii());
    }

    VDMSequence s = VDMSequenceHelper.narrow(g);

    VDMGenericHolder eholder = new VDMGenericHolder();

    int sum=0;
    for (int ii=s.First(eholder); ii != 0; ii=s.Next(eholder)) {
      VDMNumeric num = VDMNumericHelper.narrow(eholder.value);
      sum = sum + GetNumeric( num );
    }
    System.out.println("The sum of all the primes: " + sum);
    app.DestroyTag(client); // 仕上げ...
 }
  catch(APIError err) {
    System.err.println("API error");
    System.exit(1);
  }
}

public static void EchoPrimes2(int n, VDMInterpreter interp,
```

```
                                    VDMApplication app)
{
  // n以下の素数の列を生成し
  // その列を stdout へそのまま送り返す。
  // 追加として、VDM 値全体を toolbox からクライアント側へ移行し
  // これを jp.co.csk.vdm.toolbox.VDM に宣言されている
  // "real"の Java 値に変換するために
  // どう GetCPPValue を使用できるか、をこの関数が示している

  try{
    app.PushTag(client);

    // この VDMGeneric はインタープリターからの結果の保持に
    // 使用される。
    VDMGeneric g;

    // EvalExpression を使用し 20 以下の素数を計算する

    String expr = "[e|e in set {1,...,"+n+"} &" +
                  " exists1 x in set {2,...,e} & e mod x = 0]";
    g = interp.EvalExpression(client,expr);

    if(g.IsSequence()){
      System.out.println("All primes below " + n + ": " + g.ToAscii());
    }

    VDMSequence s = VDMSequenceHelper.narrow(g);

    // real Java VDM 値に変換!

    java.util.LinkedList sj =
        new java.util.LinkedList ();

    VDMGenericHolder eholder = new VDMGenericHolder();

    // Generic g を "real" Java Sequence 値に変換する
```

```
    for (int ii=s.First(eholder); ii != 0; ii=s.Next(eholder)) {
      VDMNumeric num = VDMNumericHelper.narrow(eholder.value);
      sj.add(new Integer( GetNumeric( num ) ));
    }

    int sum=0;
    for (java.util.Iterator itr = sj.iterator();
         itr.hasNext();){
        Integer i = (Integer) itr.next ();
        sum = sum + i.intValue();
    }

    System.out.println("The sum of all the primes: " + sum);
    app.DestroyTag(client); // 仕上げ...
  }
  catch(APIError err) {
    System.err.println("API error");
    System.exit(1);
  }
}

public static int GetNumeric( VDMNumeric num )
{
  byte[] b1 = num.GetCPPValue();
  try
  {
    InputStream is = new ByteArrayInputStream( b1 );
    int type = is.read();
    int c = -1;
    int last = -1;
    String str = "";
    while( true )
    {
      c = is.read();
      if ( ( c == -1 ) || ( c == ',' ) )
```

```
        {
          last = c;
          break;
        }
        str += Character.toString( (char)c );
      }
      return Integer.parseInt( str );
    }
    catch( Exception e )
    {
      return 0;
    }
  }
}
```