

VDMTools

VDM++ Rose リンクマニュアル
ver.1.2



How to contact:

<http://fmvdm.org/>

VDM information web site(in Japanese)

<http://fmvdm.org/tools/vdmttools>

VDMTools web site(in Japanese)

inq@fmvdm.org

Mail

VDM++ Rose リンクマニュアル 1.2

— 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	1
2	Rose との連携	3
2.1	Rose との連携の構造	4
2.2	Rose-VDM++ リンクのメインウィンドウ	5
2.3	VDM++を UML に変換する	7
2.4	UML から VDM++への変換	10
2.5	VDM++モデルと UML モデルの併合	14
2.6	仕様の型チェック	19
3	Rational Rose	20
3.1	Rational Rose におけるクラスダイアグラム	20
3.2	Rational Rose における操作定義	23
4	VDM++ と UML の間のマッピング規則	27
4.1	クラス構造	27
4.2	クラス間の関連	30
A	マッピング規則のまとめ	35
B	Rose-VDM++ リンクにより生成される警告	37

1 導入

オブジェクト指向分析・設計は、ソフトウェアエンジニアリング分野で広く使われている開発手法である。*Unified Modelling Language (UML)* は、このオブジェクト指向設計を表現し共通理解を図るための標準図式言語である。UML は Rational Software Corporation における Grady Booch, Ivar Jacobson, Jim Rumbaugh の共同のもと、他の進歩的な方法論者、ソフトウェアベンダー、また多くのユーザーからの支援を得て開発されたものである。Booch, OMT, および Jacobson 表記法を併合したものであり、ビジネスプロセス、オブジェクト、コンポーネント、に対するモデル化技法を提供している。現在いくつかの商用 CASE ツールが UML をサポートしていて、そのひとつが Rose 2000 であり、これは Rational Software Corporation 製 Rose 98 の後続版である。本書では以降この 2 つを総称して Rose 98/2000 と記述する。

UML は、全体としてはオブジェクト指向設計によく適合しているが、既存のモデルに形式的意味を与えるのには適さない。一方で VDM++ は、同時実行するリアルタイム処理に対してのオブジェクト指向システム形式記述として設計されている。この言語は ISO VDM-SL をもとにしてクラスとオブジェクトの概念を取り入れた拡張を行ったもので、オブジェクト指向形式記述の発展を促進する役割を担っている。

Rose との連携 はこれら 2 つの言語を結びつける。2 つの言語間にマッピング規則を定義し実装することで、*Rose* との連携 はユーザーによる VDM++ で表現されたモデル（あるいはその一部）の UML への翻訳を可能にし、またその逆も同様とした。*Rose* との連携 はラウンドトリップ技術を支えるものであって、ユーザーは UML 上でシステムに対してオブジェクト指向的に全体的モデリングを始めたら、それを VDM++ に変換することでモデルの一部に形式的意味を与えながら進めることができる。続けて、ドキュメント化目的、あるいはそのモデルを更にオブジェクト指向的にモデル化するために、マップやマージを行って VDM++ 仕様を UML へ戻すことが可能である。この 2 つの表現間におけるマッピングの行き来は、モデルの最終的な完成まで続けられる。

Rose との連携 - Rose 98/2000 のアドインとして

Rose との連携は Rose 98/2000 に対するアドインとしてインストールされ、Rose 98/2000 のアドインマネージャーにより作動や停止を行うことができる。Rose と

の連携 のインストール方法についての説明は [1] を参照のこと。

このマニュアルの使用について

本書は、 *VDMTools* ユーザマニュアル (*VDM++*) [4] の拡張版である。

本書の前に、まず Toolbox マニュアルを読んでおかれることをお勧めする。さらに *UML* [5] と Rose 98/2000 [6] の前提知識も望まれる。このマニュアルは次のような構成となる：第 2 章で Rose との連携 の様々な機能について、第 3 章で Rose 98/2000 のいくつか重要な機能について述べる。Rose との連携 が 1 つの表現形から別のものへ翻訳する方法すべてを理解するためには、適用されるマッピング規則を理解することが重要だ。これらの規則については、第 4 章で述べるが、Rose との連携 の理論的基礎となるものである。この章を読まなくても Rose との連携 を利用することはできるが、適用された変換規則を知っておくことで多くの場合に tool の使用が容易となるはずである。変換規則は付録 A の図 23 にまとめられている。Rose との連携 の許容力については、必要な場合にいつでもユーザーシナリオや様々なスクリーンダンプを伴っての提示がなされる。さらにいくつかの Rose 98/2000 の機能についても、必要な場面で述べる。Toolbox の配布では、一緒にいくつか別のソートアルゴリズムの仕様が含まれている。これらについては Rose との連携の使用記述で用いることになる。例題は [3] に記述されている

2 Rose との連携

この章では、Rose との連携 で提供されるさまざまなサービスの紹介を行う。Rose との連携 は、UML クラスの構築や表示を行い、また UML で編集したり修正したモデルを VDM++ へマップするために CASE ツール Rose 98/2000 と相互のやり取りを行う、ツールである。ツールの使用法としては、3つのカテゴリーに分けられる：

UML から VDM++ へのマッピング (フォワードエンジニアリング): UML で定義されたクラスから VDM++ 仕様を生成するためには Rose との連携 を用いる。

VDM++ から UML へのマッピング (リバースエンジニアリング): 既存の VDM++ 仕様から UML モデルを作成する。

UML モデルと VDM++ モデルの同期: システム開発中に、システムの VDM++ モデルと UML モデルを同時に修正するという状況はよく起きる。Rose との連携では各々のモデルで変更追跡が許されていて、2つのモデルを1つに併合することで同期をとり、この併合モデルが UML と VDM++ に伝えられる。

Rose との連携の基本は、第 4 章で記述されているマッピング規則である。異なる構築要素が VDM++ と UML の間でどのようにマップされるのかを厳密に述べる。さらに第 4 章で、VDM++ や UML の部品でマッピング規則に含まれないものについて述べる。たとえば VDM++ 仕様のリバースエンジニアリングを行うとき、演算や関数の本体は UML にマップされない。演算や関数のシグニチャだけが UML に翻訳される。しかしながらこれは、関数や演算の本体が失われることを意味するものではなく — 単に UML モデルからは見ることができない、ということである。その後この UML モデルを VDM++ に翻訳すると、関数や演算の本体は保持されていることがわかる。同様に、Rose との連携 は マッピング規則に含まれない UML モデルを部分的に変更してしまうといったこともない。たとえば UML モデルは、ユースケース、配置図、状態遷移図、等を用いて拡張可能である。これらの部品は、Rose との連携によって変更されることなくそのまま残されていく。Rose との連携で可能なことと限界のすべてを理解するために、どのように VDM++ Toolbox と Rose 98/2000 の結合がなされるか、また 2つのツール間の情報の流れはどうなっているか、の知識が必要となる。第

2.1 章で、このような情報について述べる。第 2.2 章は、Rose との連携のグラフィカルユーザーインターフェイスについて述べられている。第 2.3-2.5 章は、Rose との連携 の機能性の詳細な記述を行い、VDM++ と UML 間での 3 種類の変換方法を記している。

2.1 Rose との連携の構造

Rose との連携 の構造を図 1 に示す。この図はツール間のデータの流れを表すと同時に、VDM++ Toolbox と Rose 98/2000 がどのように結合しているかを描いている。

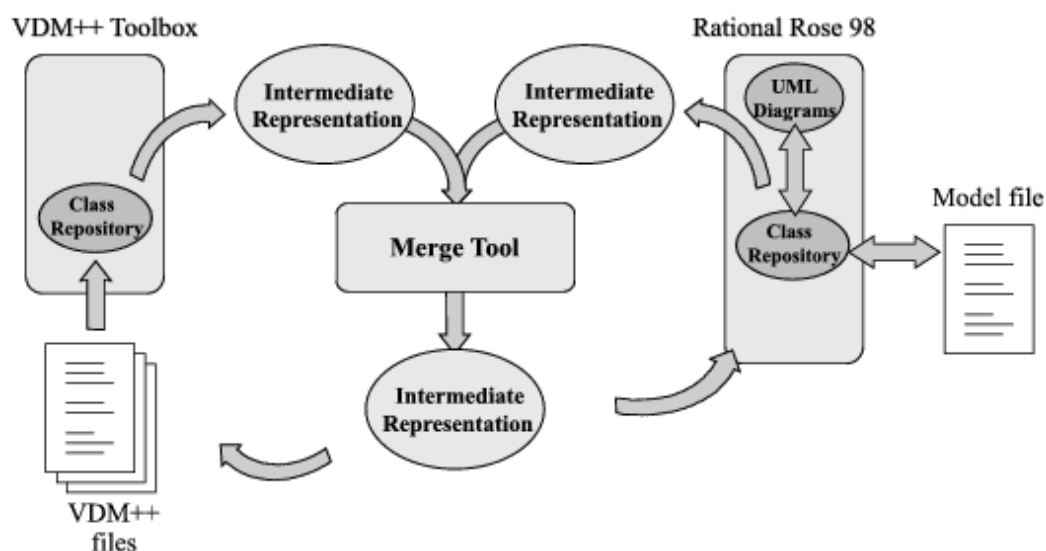



図 1: Rose との連携の構造

図の左下方よりスタートして、VDM++ ファイル (rtf 形式のもの) の集合が構文分析され VDM++ Toolbox のクラスリポジトリに加えられる様子が、示されている。Rose との連携 は VDM++ 仕様を、UML に関連する VDM++ の部品である中間表現形 (Intermediate Representation) に翻訳する。同様に UML モデルも、Rose 98/2000 のクラスリポジトリへアクセスすることで中間表現形に翻訳される。この 2 つの中間表現形は “互換性がある” ので、比較や併合を行うことができる。2 つのモデルを 1 つの共通モデルとして併合したものを、Rose 98/2000

のクラスリポジトリと VDM++ 仕様ファイル群に戻すことで、VDM++ モデルと UML モデルは同期する。この 2 つの表現形は、上記のように幾通りかの方法で併合が可能であり、結果として Rose との連携を用いた次の 3 つの方法に集約される。Rose 98/2000 の中間表現形が存在しない場合、あるいはこの表現形を無視する選択をした場合、併合は VDM++ から UML への全変換となる。同様に、VDM++ Toolbox からの結果としての中間表現形が存在しないか、あるいはこの表現形を無視する選択をした場合には、併合は UML から VDM++ への全変換となる。最後は、2 つの表現形の併合を選択することである。

2.2 Rose-VDM++ リンクのメインウィンドウ

この章では Rose との連携の GUI (グラフィカル・ユーザー・インターフェイス) について述べる。Rose との連携は VDM++ Toolbox 内でメニュー項目である ツール/Rose との連携 を選択するかあるいは  (Rose) ボタンを押すことで、起動される。図 2 がこのステップに該当する。

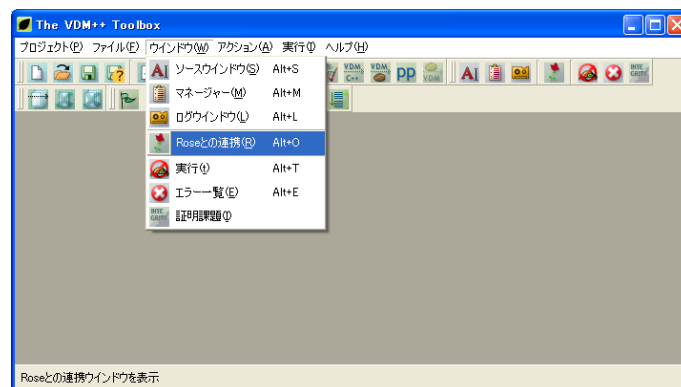


図 2: Rose との連携の起動

Rose との連携 の起動で、図 3 に示された別のウィンドウが開く。起動中は、このウィンドウから Rose との連携のすべての機能の操作を行うことができる。各々のボタンの意味については、以下の章で説明していこう。

Rose との連携が起動すると、Rose 98/2000 に対して必要とされる接続が確立される。Rose 98/2000 のインスタンスが既にマシン上で実行されていた場合は、VDM++ Toolbox はこのインスタンスに対して接続を行う。それ以外の場合は、Rose 98/2000 を自動的に始動させてそのインスタンスに対して接続がなされる。

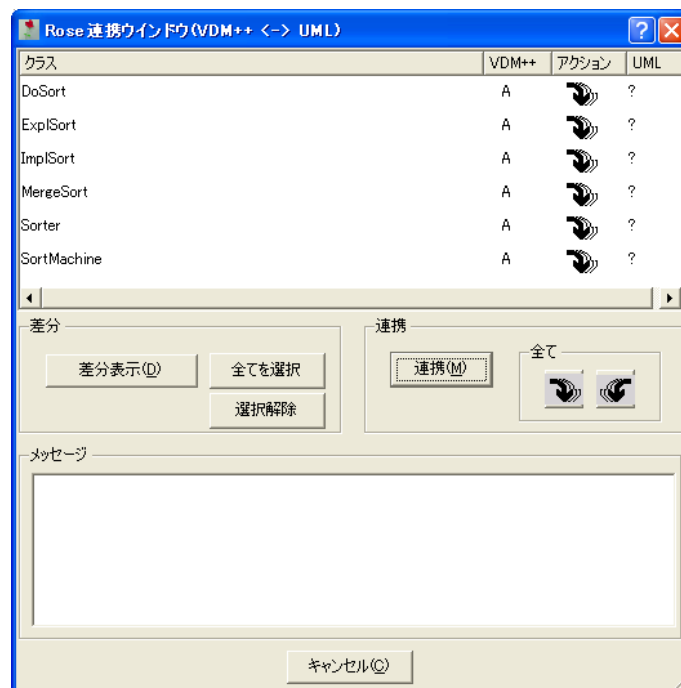


図 3: Rose との連携のメインウィンドウ

これ以降は次に挙げる簡単な規則に従って、Rose との連携 は Rose Model File の読み込みを行う：

- VDM++ Toolbox 内の現プロジェクトに名前が与えられている場合（つまり、現存のプロジェクトが開いた状態、あるいは新しいプロジェクトを1つ生成し保存した状態）、Rose との連携 はこのプロジェクトファイルと同じ名前と配置場所をもつモデルファイルを開こうと試みる。たとえばプロジェクトファイル MyProject.prj に対して、Rose との連携 はこのプロジェクトファイルと同じディレクトリにモデルファイル MyProject.mdl を開こうとする。同じ名前のモデルファイルが存在しなければ、空のモデルファイルが生成される。
- VDM++ Toolbox 中の現プロジェクトが一度も保存されていない場合でも、Rose との連携 はそのまま現在 Rose 98/2000 中に置かれたこのモデルを使用する。

Rose との連携 が開始されたとき、VDM++ モデルと UML モデルの中間表現は自動的に、各々が VDM++ Toolbox に含まれるクラスと Rose 98/2000 に含まれ

るクラスに基づき処理される。この2つの表現形の内容は、ユーザーにクラス名称の一覧として表示され、VDM++ クラスと UML クラスの各々の状態は指定の状態標識記号を用いて示される。これら標識記号の厳密な意味の詳細は、この章の後方に記されている。以下に続く章では、1つの表現形から別の表現形にどのように翻訳するかについて述べると共に、Rose との連携を用いて2つの表現形を併合する方法についても述べていく。このツールの処理能力を描き出すために、[4] のソートの例題を用いる。

2.3 VDM++ を UML に変換する

Rose との連携 の開始前に1つ以上のクラスが構文チェックされる場合、これらのクラスは 4 章で述べられるマッピング規則を用いた中間表現形に翻訳されることになる。ここで、VDM++ Toolbox の処理対象にソート例題ファイルが構成され名称 Sortpp.prj のプロジェクトファイルが生成された、という状況を仮定してみる。さらに VDM++ ファイルの構文チェックが完了し、Rose 98/2000 はクラス定義を含まないと仮定する。この後に Rose との連携 の起動を行うと、図 4 に示されるようなクラス一覧が表示される。


DoSort	A		?
ExpISort	A		?
ImplSort	A		?
MergeSort	A		?
Sorter	A		?
SortMachine	A		?

図 4: ソート例に対するクラス一覧


ここで用いられているクラス指定子は A と ? で次のような意味をもつ:


A モデルの直前の処理でこのクラスが追加されたことを示す。


? モデル内でこのクラスは認識されないことを示す。

したがって図 4 のクラス一覧は次のように解釈すべきである: 6つのクラスすべてが VDM++ モデルに追加された。一方 VDM++ で定義されたクラスはいずれも UML モデルで認識されない、この理由は簡単で Rose 98/2000 は Rose と

の連携の起動前にクラス定義を含まないと仮定してあったからである。一覧では各々のクラスが、2つの表現形の併合をどのように引起すかの構成に用いる独自の作用ボタンをもつ。Rose との連携ウィンドウが開かれるときに、各々のボタンには結合する2つのクラス指定子に基く既定の行動が割り当てられている。作用ボタンの状態の変更には単にクリックすればよい。クリックで、最大4通りの異なる状態間での切り替えがなされる：

 **VDM++ から UML:** この動作は、VDM++で定義されたクラスの定義をUMLの同じ名クラスへマップする。このクラスがUMLに既に存在していた場合はこれが更新され、そうでない場合は新しいクラスが生成される。

 **UML から VDM++:** この動作は、UMLで定義されたクラスの定義をVDM++の同じ名のクラスにマップする。この動作で、UMLで定義されたクラスの定義はVDM++の同じ名のクラスにマップされる。結果は、新しいクラスの生成かあるいは既に存在する同じ名のVDM++クラスの定義更新となる。第 2.4 章を参照のこと。

 **併合:** この動作は、VDM++で定義されたクラスの定義をUMLで定義されたクラスの定義に併合する。第 2.5 章を参照のこと。

 **除外:** これは処理対象の新しいモデルからこのクラスを除外する。結果は、このクラスがUMLモデルとVDM++モデルの両方から取り除かれる。

図 4 の例において、すべての動作ボタンは既定の動作として、VDM++からUMLへのマップとして設定される。動作ボタンのひとつをクリックすると、“VDM++からUML”と“除外”の2つの状態が切り替え可能としてのみ現れる。この理由は、他の2つの状態はRose 98/2000で定義されたクラスがないため意味がないからである。動作ボタンの状態が変化してしまったら、“既定”ボタンをクリックすることでいつでも元の既定の設定に戻すことができる。さて例題にもどり：Rose との連携にVDM++からUMLへの変換を行わせるために、“連携”と書かれたボタンをクリックする。Rose との連携はここで新しく6つのクラスをRose 98/2000のクラスリポジトリに追加する。Rose との連携により生成された新しいクラスはすべて、“Generated classes”という名のパッケージに追加される。しかしいつでも、“Generated classes”パッケージから別のパッケージに移すことは可能である。このクラスがその後Rose との連携によって更新/修正されれば、“Generated classes”パッケージに戻されることはない。図 5 は、ソート例題の

VDM++ クラスを UML に翻訳した後の、クラスリポジトリのスナップショットである。

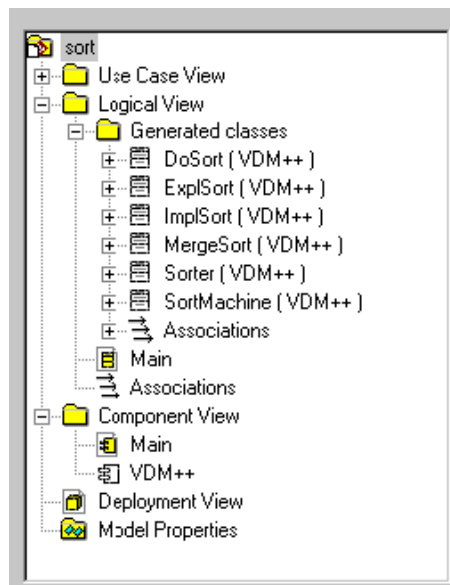


図 5: ソート例題に対する Rose 98/2000 のクラスリポジトリ

1つの例として、VDM++クラス Sorter に対して生成された UML クラスを見よう。図 6 は Sorter クラスの VDM++ 仕様と生成された UML クラスである。見ての通り、このクラスの変換は極めて直接的である。使用されるマッピング規則の詳細は、第 4 章を参照のこと。

```
class Sorter
operations
  public
  Sort: seq of int ==> seq of int
  Sort(-) ==
    is subclass responsibility
end Sorter
```

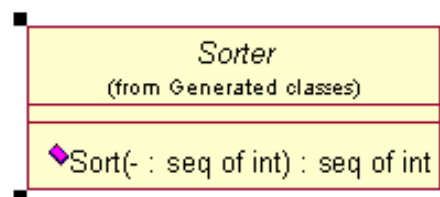


図 6: VDM++ Sorter クラス (左) と UML Sorter クラス (右)

Rose 98/2000 では、クラスリポジトリを閲覧し、属性や操作の定義を検索や修正できる。さらに、クラスリポジトリにあるクラスからクラスダイアグラムの生

成を行うことができる。第 3 章では、簡単に Rose 98/2000 の紹介を行おう。

生成されたファイル

Rose 98/2000 の UML モデルを更新する前に、Rose との連携 は現在の UML モデルのバックアップを生成している。バックアップは、_old を Rose 98/2000 の現在のモデルの名前に付け足した名を用いる。たとえば、モデルファイルである MyProject.mdl は MyProject_old.mdl にバックアップがとられる。VDM++ から UML へのマッピングが予期しない結果である場合、Rose との連携でなされた変更はこのバックアップで簡単に取り戻すことができる。

2.4 UML から VDM++ への変換

この章では、Rose 98/2000 で定義された UML モデルから VDM++ を生成するため Rose との連携 をどのように使用するかを述べる。事柄をできる限り単純化し、第 2.3 章で生成されたモデルから VDM++ を生成する。VDM++ Toolbox がモデルをまったく含まない、つまり、Rose との連携の発動に先立ち構文がチェックされたクラスはないということを、仮定する。VDM++ Toolbox のメニュー項目の プロジェクト/新規プロジェクト を選択し、それまでに VDM++ Toolbox から持ってきたクラスをすべて削除する。メニュー項目の *Tools/Rose との連携* を選択すると、図 7 で示されたクラス一覧が表示される。予想通り 6 つのクラスはどれもが VDM++ では認識されないクラスで、UML で定義されたものと分かる。

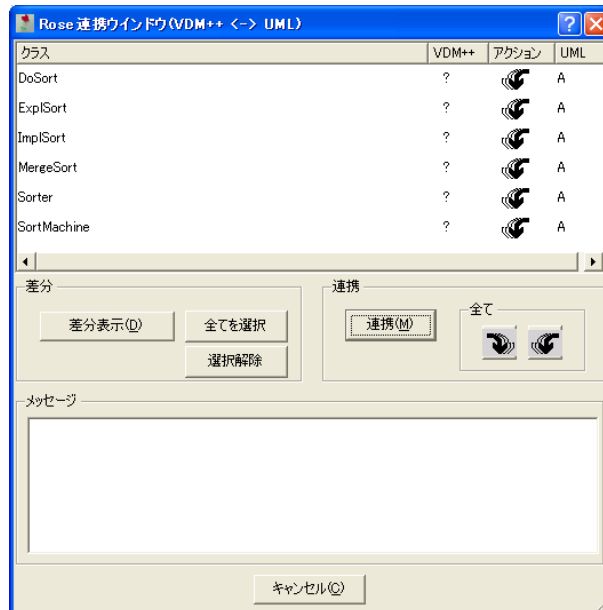


図 7: UML クラスのみに対するクラス一覧

“連携” ボタンをクリックすると、作業ディレクトリの選択ダイアログが表示される。UML で定義された 6 つのクラスに対して生成される、VDM++ クラスファイルを保存するディレクトリを選択することができる。



図 8: 作業ディレクトリの選択ダイアログ

例として、図 9 でクラス ExplSort の UML 版と、同クラスに対して生成された VDM++ 仕様、を示す。関数機能や演算は単にシグニチャとして生成されることに注意しよう - ユーザーはそれらの本体を後で指定しなければならない。

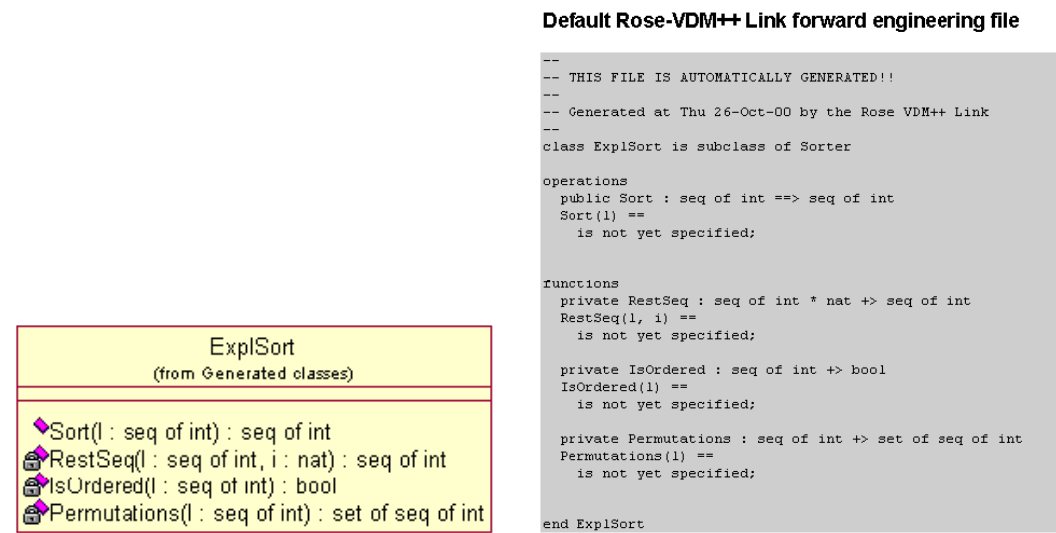


図 9: UML ExplSort クラス (左) と VDM++ ExplSort クラス (右)

生成されたファイル

UML に導入される新しいクラスは、各々そのクラスと同じ名前の別々の rtf ファイルの中に生成される。たとえばクラス MyClass は、プロジェクトファイルと同じディレクトリ中の MyClass.rtf という名前のファイルに生成される。この名前のファイルがすでに存在する場合は、Rose との連携 が警告を出し新しいファイルは 生成されない。この場合は既存ファイルを移動するか名前変更をするかしておくべきで、あるいは UML に導入されるクラスに名前の不一致を避けて他の名前を与えるべきである。VDM++ Toolbox のログウィンドウは、生成ファイルの名前すべてをディレクトリと共に表示する。UML に導入されるクラスに対する新しい rtf ファイルを生成するとき、Rose との連携 は NewClass.rtf という名前の特別なファイルをこの新しいファイルの “スケルトン” に用いる。このファイルは、VDM++ Toolbox インストールを行った場所の uml サブディレクトリに 置かれていなければならない。このスケルトンファイルの内容は変更してもよいが、次の項目を満たすことが要求されている：

- 文書は 常に VDM ブロックを、少なくとも 1 つの (空の可能性もある) VDM スタイルの段落を含む。
- ファイル名は `NewClass.rtf` である必要があり、このファイルは VDM++ Toolbox インストールにおける `uml` サブディレクトリに置かれている。

生成されたクラスの利用

生成された VDM++ クラスは、自動的に現在のプロジェクトに加えられ構文解析が行われる。これが可能なのは、Rose との連携 が常に構文的にほとんど正しいクラスを生成しているからである。しかしながら、Rose モデルの構文エラーは VDM++ モデルまで持ち越される (通常はキーワードの不適切な使用とされる)。ここでおそらく、たとえば演算操作と関数機能の本体を書くというように VDM++ 仕様を拡張することで、続行されることになる。このモデルのオブジェクト指向面 (継承や関連といった関係) で修正したい、あるいは新しいインスタンス変数、関数、その他の追加が必要となった、という場面も想像される。この場合、単に生成された `rtf` ファイルを編集すればよい。その後両モデルの一貫性を保つために、新しい VDM++ モデルも簡単に UML にリバース・エンジニアリングできる。VDM++ モデルの修正を行う間は UML モデルを変更しないという条件下で、クラスを構文チェックし (必要ならば型チェックもして) 第 2.3 章の記述のように続行される。

UML から VDM++ へ変換する間に起こされる警告

UML クラスが VDM++ に翻訳されるとき、UML 属性と演算は VDM++ 構築要素に転換される。UML モデルで使用された名前と型も、必然的に VDM++ の構文規則に従わなければならない。これが当てはまらない場合は、UML 定義は単に無視されユーザーには警告という通知がなされる。これは、生成される VDM++ 仕様を構文的に最大限正しくするためである。Rose との連携 が UML からの定義を読み取るときに生成される警告に対しては、注意が必要だ。なぜなら、警告が発せられるのは常に UML モデル中で VDM++ に翻訳されないものがあるからで、それは結果として構築された共通表現形において無視されてしまうことになる。Rose との連携 構造の結果 (第 2.1 章を参照)、マージツールで処理された表現形は Rose 98/2000 のクラスリポジトリの現モデルを置き換える。

UML モデルの読み込み中に無視された構築要素は、“連携” ボタンがクリックされたら除外されることを意味する。

詳しくは付録 B を参照のこと。

2.5 VDM++モデルと UML モデルの併合

第 2.3 章と 2.4 章で、モデルは VDM++ と UML のどちらかで表現されたものと仮定していた、つまり、図 1 の 2 つの表示のうちの 1 つは空であった。これは事柄をできる限り単純に扱うためであり、また 1 つの表現形のモデルをもう 1 つの表現形に翻訳することがどのように可能かを描くためであった。これは、既存の VDM++ 仕様を記述するのに UML ダイアグラムを生成するとき、また既存の UML モデルから VDM++ を生成するときに役に立つ。しかしもっと大きくもっと複雑なシステムをモデル化する場合、UML と VDM++ モデルが同時に展開されるものであれば、第 2.3 章と 2.4 章の仮定は保持されるものではない。この場合の要求に応えるため、Rose との連携はこの章で 2 つの異なるモデルを 1 つに併合する機能を提供する。以降は、VDM++ Toolbox に第 2.3 章のソート例が初期化構成され、さらにこの仕様は第 2.3 章に記述されているように UML に翻訳されている、と仮定するとしよう。続いて VDM++ 仕様と UML モデルの両方とも少しだけ修正がなされている。

図 10 で、VDM++ モデルと UML モデル両方にいくらか変更がなされた後の Rose との連携 のクラス一覧を示している。

DoSort	-		-
ExplSort	-		M
ImplSort	D		-
MergeSort	M		D
QuickSort	?		A
SortMachine	-		-
Sorter	-		-

図 10: UML と VDM++ の両方が変更されたときのクラス一覧

この図より、UML で新しいクラス QuickSort が追加され、ExplSort クラスは変更されたということが見て取れる。さらに、MergeSort クラスはモデルから

削除されている。VDM++ では、MergeSort クラスが変更され ImplSort クラスが削除されている。

図 10 では M、-、D という新しく 3 つのクラス指定子を紹介していて、これらは前で述べた 2 つの指示子と共に、Rose との連携で使用される 5 種類のクラス指示子を構成している。

3 つのクラス指示子は次の意味をもつ：

M はクラスが変更されたことを示す。

- はクラスに変更がないことを示す。

D はモデルが最後に実行された以降にクラスが削除されたことを示す。

同じクラスで 2 つの表現形間の違いを調べるのに、簡単なことでクラスの横のチェックボックスをクリックし、続けて“差分”ボタンをクリックしてみる。結果はログウィンドウに表示される。VDM++ と UML の表現形の違いをすべて調べるためには、全クラスを選択するための“All”ボタンを押せばよい。図 11 で全クラスの VDM++ と UML の表現の違いを処理した結果を示している。たとえば、ImplSort クラスの 2 つのバージョンで関数 RestSeq に対する 1 つの引数の名称が異なる；“i”が“j”となっている、それのみ違うことがわかる。

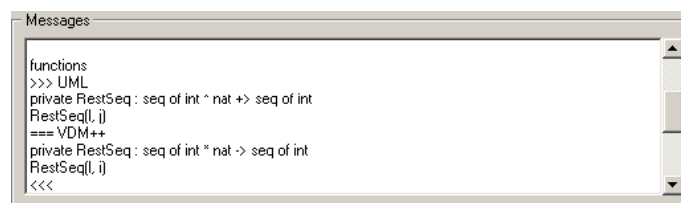


図 11: すべてのクラスに対する VDM++ と UML 表現の違い

仮にこれらの 2 つのモデルを 1 つの表現形に併合したい場合、Rose 98/2000 におけるクラスリポジトリと VDM++ 仕様を含む rtf ファイルに影響が及ぶ。2 つの表現形の併合をどのように行うのか決定するためには、求める動作をアクションボタンの状態を変えることで選択しなければならない。

クラス別に可能な動作の一覧を図 12 に示す。

クラス	可能な動作	“連携” ボタンを押した結果
DoSort	 (既定)   	<p>どちらのモデルにおいてもクラスの変更はないため、更新はされない。</p> <p>既定動作と同じ</p> <p>既定動作と同じ</p> <p>クラスは両モデルから除外される。</p>
Sorter	 (既定)   	<p>どちらのモデルにおいてもクラスの変更はないため、更新はされない。</p> <p>既定動作と同じ</p> <p>既定動作と同じ</p> <p>クラスは両モデルから除外される。</p>
ExplSort	 (既定)   	<p>UML モデルでなされた修正が VDM++ モデルにおいてもなされる。</p> <p>UML モデルでなされた修正は削除される。クラスは元の状態に戻る。</p> <p>以下に述べるように矛盾がおきる。</p> <p>クラスは両モデルから除外される。</p>
ImplSort	 (既定)  	<p>クラスは UML モデルからも削除される。</p> <p>新しい ImplSort クラスが VDM++ モデルに生成される。</p> <p>既定動作と同じ</p>
MergeSort	 (既定)  	<p>VDM++ モデルでの変更を含む新しい MergeSort クラスが UML モデルに生成される。</p> <p>クラスは VDM++ モデルからも削除される。</p> <p>クラスは VDM++ モデルからも削除される。</p>
QuickSort	 (既定) 	<p>クラスは VDM++ モデルに追加される。</p> <p>クラスは両喪出るから除外される。</p>
SortMachine	 (既定)   	<p>どちらのモデルにおいてもクラスの変更はないため、更新はなされない。</p> <p>既定動作と同じ</p> <p>既定動作と同じ</p> <p>クラスは両モデルから除外される。</p>

図 12: VDM++ モデルと UML モデルの併合時に可能な動作

矛盾する定義を含むマージクラス

UML モデルと VDM++ モデルが平行して展開することが許される場合は、矛盾の可能性が生じる。たとえば、VDM++ モデルと UML モデルの両方で同じ名前で異なるシグニチャを持つ操作を定義したとすると、Merge Tool は2つのうちのどちらの操作を結果としてのモデルに選ぶべきかわからなくなる。“連携” ボタンのクリックで2つのモデルの併合が始まるが、矛盾に出会うと併合は異常終了し、ユーザーには矛盾の起きたクラス間についての情報が届けられる。ExplSort クラスの VDM++ 表現形と UML 表現形の併合では、たとえば両モデルの RestSeq 関数の引数の名前が1つ異なっているということから、矛盾が引き起こる。この矛盾は図 11 でも明らかである。ユーザーは通常 “Diff” ボタンを使用して矛盾の箇所を特定し、単純に2つの定義のどちらを残すか決定する。すべての矛盾が解決すると、VDM++ 仕様は構文チェックされ、2つのモデルの併合を行うため Rose との連携 が再起動される。ユーザーは矛盾を避けるためモデルの1つを変更するという代わりに、矛盾を解決するために併合モードの変更を選択することもできる。ただ異なる状態間の切り替え用トグルボタンをクリックし、表現形の2つのうちの1つを抑制する動作を選択する。矛盾は、表現形が併合されたときにのみ起こるのである。

VDM++ 仕様の更新

VDM++ モデルと UML モデルが互いに矛盾しなければ、“連携” ボタンのクリックで、元の2つのモデルを併合した結果を含めるために Rose 98/2000 のクラスリポジトリが自動的に更新される。同様に VDM++ プロジェクトの rtf ファイルが自動的に更新され、UML モデルに応じてなされた変更の反映のために構文分析が行われる。新しいクラスが導入されていれば、第 2.4 章で述べている様に新しい rtf ファイルとして生成される。Rose との連携 は rtf ファイルを更新する前にそのコピーを作成する。コピーは元のファイル名に _old.rtf を付与した名前となる。つまり、ファイル ExplSort.rtf は更新前に ExplSort.rtf_old.rtf にコピーされる。以下は VDM++ 仕様ファイルの更新時に適用される規則である：

新しい要素 は、そのクラスの先頭に追加される。たとえば、新しいインスタンス変数はクラスで最初の instance variables ブロックの先頭に挿入される。instance variables ブロックが VDM++ クラス中ですでに宣言されている場合は、そのクラスの先頭に生成される。

古い要素 は、VDM++ のコメントに変換されファイルから除外される。この方法ならば、除外の必要のなかったものを簡単に元に戻すことができる。

修正された要素 は、単純に古い定義を（コメント化して）除外し新しい定義を追加するという方法で扱われる。

仕様ファイルの修正すべては、次のコメントで識別されることになる：“Rose-VDM++ リンクによる追加” あるいは “Rose-VDM++ リンクによる削除”。図

13 は、関数 RestSeq のシグニチャが UML で修正された結果、Rose との連携によって ExplSort クラスがどのように更新を受けたかを表している。ある状況下で Rose との連携が必要なファイルの更新ができなくなることがある。一部のワードプロセッサでは、今編集集中のファイルはロックすることで他のアプリケーションからのアクセスを許可しない。この結果として Rose との連携はロックされたファイルを更新することができない。更新すべきファイルが他のアプリケーションによりロックされていれば、Rose との連携はそれらのファイル名称を一覧にし、ユーザーは（他のアプリケーションがこれらを使用中でないことを確認することで）ロックを解除するかまたは併合処理をキャンセルするか、どちらか選べる。Rose との連携で修正されるすべてのファイルは、VDM++ Toolbox のモデルが Rose 98/2000 のモデルと同一になるように、それ以降は自動的に構文解析される。

```
class ExplSort is subclass of Sorter
...
functions
-- Removed by the Rose-VDM++ Link:
--   private RestSeq : seq of int * nat +> seq of int
--   RestSeq(l, i) ==
-- Added by the Rose-VDM++ Link:
  private RestSeq : seq of int * nat +> seq of int
  RestSeq(l, j) ==
    [l(j) | j in set {inds l \ {i}}];
...
end ExplSort
```

図 13: 更新された ExplSort.rtf ファイル

2.6 仕様の型チェック

Rose との連携 を用いるために、VDM++ 仕様は唯一構文チェックのされる必要があるが、同様に型チェックも推奨されている。インスタンス変数で定義されたクラス間関係の情報は、仕様が型チェックされるまで利用できない。したがって VDM++ 仕様がただ構文チェックのみされていた場合、このような関係で定義され、関連生成ができないという意味で Rose との連携 は完全とは言えないかもしれない。

3 Rational Rose

ここまでで Rose との連携の特徴のいくつかを論じてきた。このツールは Rose 98/2000 との密接な結び付きに依存するため、このマニュアルの適用範囲として Rose 98/2000 へ簡単な導入を行う。最初に、Rose との連携によって生成されたクラスやユーザーが生成したクラスから、クラスダイアグラムを作る方法を述べる。第 3.2 章は Rose 98/2000 で定義の修正、削除、追加を行う方法が述べられている。Rose との連携 はいわゆる Rose 98/2000 のアドイン言語であり、 [1] で述べられているようにインストールできる。このアドインは Rose 98/2000 のアドインマネージャーを通した起動や終了が可能である。Rose 98/2000 は Rose との連携 の起動 (既定で起動) で、 VDM++、VDM++ の基本データ型と UML から VDM++ へのマッピングで使用される特殊なステレオタイプ、を 認識する。

3.1 Rational Rose におけるクラスダイアグラム

クラスダイアグラムの生成

VDM++ クラス定義を UML に翻訳するのは、Rose 98/2000 のクラスリポジトリにおけるクラス定義を単に作成または更新するということで - 自動的にクラスダイアグラムの生成は行わない。しかし、Rose 98/2000 ではクラスダイアグラムの生成は簡単である。Rose 98/2000 でクラスダイアグラムを生成するには、次のように進める (以下の動作は Rose 98/2000 内から行う):

1. 空のクラスダイアグラムを作成するか、既に存在するものを開く。
2. クラスダイアグラムにクラスを追加する
 - ドラッグアンドドロップ: クラスリポジトリ中のクラスを 1 つクリックし、それを追加したいクラスダイアグラム中にドロップする。
 - *Query/Add Classes...* を選択し、追加したいクラスを含むパッケージ (通常は "生成クラス" パッケージ) を選ぶ。"選択クラス" リストボックスに追加するクラスを移動し、"OK" をクリックする。選択済みクラスはすべて自動的に作動中のダイアグラムに追加され、自動的にダイアグラムに配置される。

図 14 で、このアプローチにより生成されたクラスを示す。

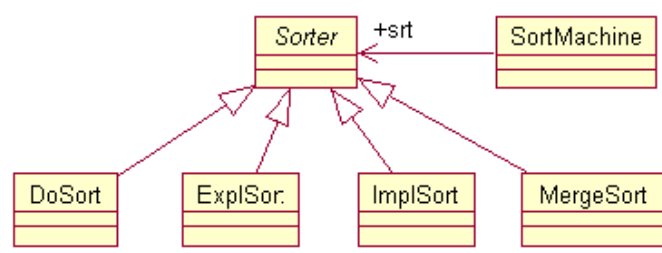


図 14: ソート例についての Rose 98/2000 のクラスダイアグラム

ダイアグラムの修正

Rose 98/2000 は生成されたダイアグラムの配置変更も行う。マウスを用い簡単な操作で、クラスや関連を他の任意のクラスに移そう。Rose 98/2000 のクラスダイアグラムを修正すると以下の影響がある:

- 配置替えて、つまりクラスや関連の移動でそのダイアグラムに修正が行われるが、クラスリポジトリへの影響はない。さらに後で、Rose との連携 がリポジトリを更新する場合も、ダイアグラムの配置は変えない。クラスが リポジトリから削除されたときだけは、あるいは他のクラスに対して関連 が変化したときは、ダイアグラムの表示が変化するであろうが、レイアウトの原型は保たれる。
- クラス間に関係（継承か関連）を追加することで、クラスリポジトリのクラス定義は変化する、つまり、モデルに新しい関係が追加される。新しい関係が追加されたダイアグラムは当然すぐに形を変える。それに対し、他のダイアグラムは新しい関係を示すことはないが、変化を反映する更新はなされているはずである。次の章ではクラスダイアグラムの更新方法を述べる。
- クラスのインスタンス変数、演算、その他の変更は、クラスを含む他のすべてのダイアグラムですぐに表示される。

関係の選別

Rose 98/2000 は、選択中のダイアグラムである種の関係の型を残すという機能を提供している。この機能を用いて、クラス間の継承関係のみが示されるようなダ

イアグラムを簡単に構築することができる。Rose 98/2000 でどのような種類の関係をダイアグラムに表示するかを指定するために、*Query/Filter relationships...* を選択する。図 15 では、関係選択のためのダイアログボックスを示している。

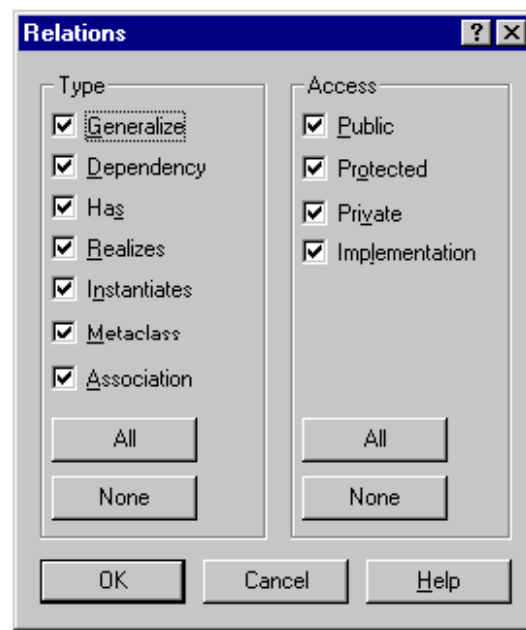


図 15: 関係の指定

このダイアログボックスは、ダイアグラムの関係の更新にも使用できる。単純に全種類の関係を選択し (“All” ボタンを使用)、“OK” をクリックしよう。すると現在選択されたクラスダイアグラムのクラス間で定義された全関係が、再表示される。Rose との連携により追加されて既存のダイアグラムに表示された新しい関連を確定する、ただ 1 つの方法である。

ローカルビュー

しばしば単一のクラスダイアグラムで、すべてのクラスとそれらの相互関係を示すのにあまりに複雑になりすぎて、結果として理解するのが困難になる場合がある。この理由から Rose 98/2000 では、ローカルビューの生成機能を提供する。このため、始めに初期状態としてローカルビューを作成する対象の 1 つ以上のクラスから構成される新しいクラスダイアグラムを作成することが必要で、これに対してローカルビューを作成したいと考える。後はこの新しいダイアグラムでク

ラスを選択し、*Query/Expand Selected Elements...* メニュー項目を選ぶ。ダイアグラムウィンドウが飛び出し、生成するローカルビューで見たいレベル番号と関係を決定するよう促される。“OK” ボタンをクリックすれば、生成されるローカルビューを見せるクラスダイアグラムの作成が行われる。

この処理を用いて、たとえばクラスとそのすべての直接のスーパークラスとサブクラスを表示する新しいクラスダイアグラムの生成が、簡単に行える。

3.2 Rational Rose における操作定義

Rose 98/2000 においては常に、任意の定義の修正、削除、追加が許される。この章では、Rose モデルの変更を行う例をいくつか挙げる。

リポジトリ内の定義の修正

Rose 98/2000 における属性の定義 (VDM++においてはインスタンス変数と値) および操作 (VDM++においては操作と関数) は、簡単に修正できる。修正したいクラスをダブルクリックするという簡単な操作で、ダイアログボックスが現れ、クラスのすべての部位に対する定義を検証し変更する機能が提供される。“クラス仕様” ウィンドウは任意クラスの仕様の閲覧に用いられる。図 16 は、SortMachine クラスの仕様ウィンドウを表示している。

特定の実体型の生成

ここで、Rose 98/2000 では新しい実体がどのように生成されるかを述べよう。UML の実体は VDM++ 構築要素にマップできるが、これ以上のことは第 4 章を読むことをお勧めする。Rose 98/2000 の VDM++ アドインが、新しい実体を生成する際の助けをする。以下では、クラス、属性、演算、関連、の生成について述べていこう。

- クラス

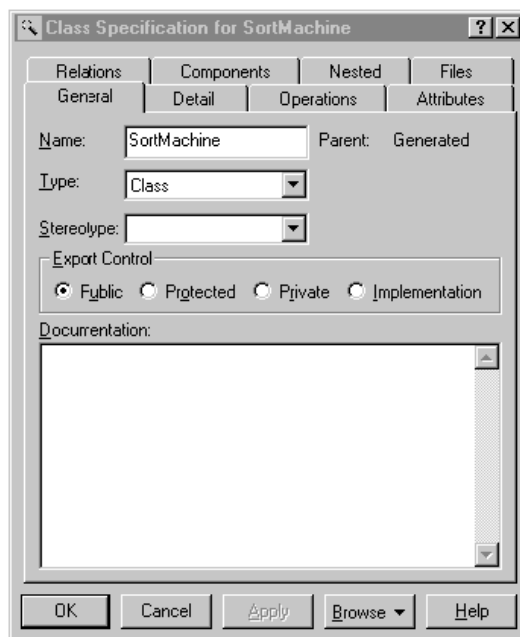



図 16: クラス SortMachine に対する UML クラス仕様

Rose 98/2000 の  ボタンを選択することで、新しいクラスが生成される。基本的な VDM++ 型や Rose との連携 で定義済みのステレオタイプを使用するためには、VDM++ コンポーネントとして新しいクラスが割り当てられなければならない。このために、次の手順に従う：

- － Rose 98/2000 のブラウザで “Component View” パッケージを右クリックし新しいコンポーネントを作成してから、*New/Component* を選択する。この新しいコンポーネントをダブルクリックし、その仕様ウィンドウ (“General” タブ上) の言語に VDM++ を選ぶ。
- － この新しいコンポーネントの仕様ウィンドウ内で “Realizes” タブを選択して、割り当てるべきクラスを選択する。それら選択クラスは右クリックし “Assign” を選ぶことで、VDM++ コンポーネントに割り当てられる。

もう 1 つの方法としては、マウスでクラスをドラッグしコンポーネントの中に入れることで、簡単にこれらのクラスのコンポーネント割り当てを行うことができる。

- 属性

第 4 章で述べたように、UML の属性は VDM++ のインスタンス変数と値を表現するために用いられる。クラスを右クリックしメニュー項目 *New Attribute* を選択すれば、そのクラスに簡単に 新しい属性を追加できる。第 4 章で述べているが、3 種の VDM++ 構造を区別するためにステレオタイプを使う。各々の属性に対しては、ステレオタイプ (“<<” “>>” で指定) を割り当てることができる。ステレオタイプを指定しなかった場合、属性は既定でインスタンス変数と見なされる。VDM++ アドインが、属性に対するステレオタイプを定義する際の手助けを行う。図 17 で、新しく追加された属性の “Attribute Specification” ウィンドウを示す。このクラスは VDM++ コンポーネントに割り当てられるため、3 つの定義済みのステレオタイプの中から、<<instance variable>>または <<value>>を選ぶことができる。

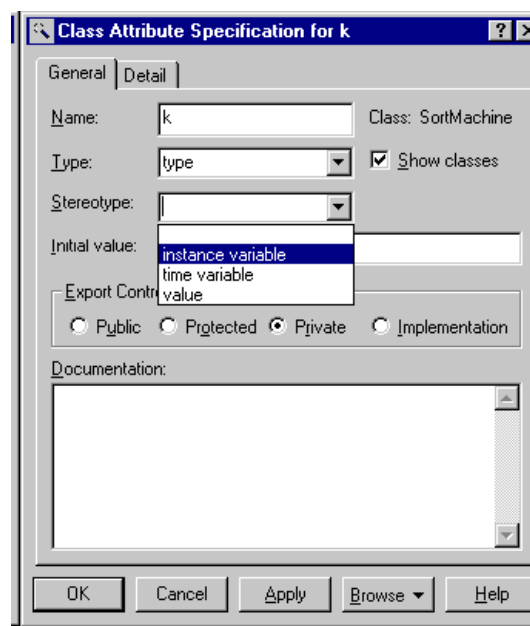


図 17: 属性に対する定義済みステレオタイプの選択

- 操作

クラスを右クリックしメニュー項目を選択することで、簡単に新しい操作をクラスに追加することができる。属性で用いたと同様、VDM++ の関数と操作を区別するためにステレオタイプを用いる。これらのステレオタイプ

は属性と同様にして作成される。ステレオタイプが省略された場合は、既定ですべてが操作であると解釈される。

- 関連

2つのクラス間に関連を定義するには、簡単なことでまず“Association Tool”をクリックしてから、1つのクラスをクリックしマウスを他のクラスにドラッグする。ダブルクリックすると“Association Specification”ウィンドウが開かれ、更に関連の詳細を指定できる。ここでは関連に対する、ロール名、多重度、制約、を追加することができる。

ロール名: マッピング規則で述べたが、関連のロール名は、関連の翻訳対象となるインスタンス変数の名前として用いられる。このため関連に少なくとも1つのロール名を与えることが、生成されたVDM++クラスが1つのインスタンス変数として表されるために、極めて重要である。ロール名が与えられていない関連は、単に無視されてしまう。

多重度: ロールに対する多重度が指定されていない場合、既定で1が仮定される。

制約: “Association Specification”ウィンドウにおいては、ある関連のロールに対して複数の制約を追加することが可能である。例えば{ordered}といった制約をロールに追加すると、これがVDM++に翻訳されるとき、オブジェクト参照では *set* でなく配列 (sequence) が生成される。

限定子: 関連に簡単に限定子を追加できる。限定子を保持するためには、クラスの傍の関連を単純に右クリックし、そこで *New Key/Qualifier* を選択する。VDM++ 内で限定された関連をモデル化したマップ中では、定義域として用いるために限定子の名前に VDM++ 型を指定する (第 4 章参照)。修正するロールの傍の関連を右クリックすることで、関連に関する2つのロールの詳細をいろいろ修正することができることを覚えておこう。

4 VDM++ と UML の間のマッピング規則

この章では、VDM++ と UML 間の関係を示す。これは、Rose との連携で 1 つの表現形から他の形に変換を行うときの適用規則を示すということになる。以下のマッピング規則を完全に理解するためには、明らかに、VDM++ と UML について詳細な知識が必要とされる。ここでは 2 つの表記法についての紹介は行わないで、必要な場合は単に様々な構築要素について述べるに留める。完全な提示を行うために [2] と [5] を参照する。以下の章で提示する規則は、付録 A の図 23 にまとめている。VDM++ 仕様のいくつかをそれに対する UML 変換と共に表示することで、規則提示とする。これと関連して、定義されたマッピングが *injective* であることを述べておくことが重要で、結局 VDM++ から UML へのマッピング定義は、同時に逆マッピングの定義となる。更には、以下の規則は VDM++ で記述できるすべての構築要素を網羅するものではなく、UML のすべての特徴も網羅してはいない、ことを述べておくこと大切である。マッピング規則は単にマッピングを 1 対 1 にするために定義されているので、2 つの表現形のうちの 1 つでしか記述できない構築要素はマッピングの外に取り残されていることを意味する。Rose との連携では、以下のマッピング規則で網羅されていない構築要素は単に無視される。

4.1 クラス構造

この章では、2 つの表現形の間で内部のクラス構造がどのようにマップされるのかを明らかにする。

インスタンス変数と値

VDM++ において、あるオブジェクトの内部状態はインスタンス変数と値定義によって叙述される。値とインスタンス変数の主な違いは、値は読み取り専用であるということだ。UML においては内部状態は属性部に記述されるが、ここでインスタンス変数と値をそれらに対応する型と値と共に簡単な一覧にしてみる。インスタンス変数と値を区別するために、`<<instance variable>>` と `<<value>>` という名のステレオタイプを用いる。UML で宣言される属性に対する構文は、次の通り:


```
name : type = initial-value
名称 : 型 = 初期値-値
```

これは VDM++でのインスタンス変数と値の定義の構文とほぼ一致する。したがって値とインスタンス変数は、UML に直接マップされ得る。図 18 では、インスタンス変数と値のマッピングを描く。

操作と関数

操作と関数は、UML クラスの操作部中にマップされる。ここでは、ステレオタイプ <<operation>>と <<function>>に区別される。UML で操作見出しの定義を行うための構文は、次の通り:

```
名称 (引数 : 型, ...) : 戻り型
```

これは、VDM++で陰定義関数 / 操作のための構文とほぼ同じである。明示的にあるいは暗黙で定義された関数見出しは、操作見出しと同様に、この構文に変形されなければならない。マッピングの操作と関数の例題は、図 18 を参照のこと。操作で定義された事前条件と事後条件は、UML マッパーに保持される。これは、構文的に正しい VDM 式として記述される必要があるからである。UML は操作の結果識別子の定義方法を提供しないため、特殊識別子 RESULT が、事後条件の中で操作結果の記述に使用される。陽または陰関数を UML の同じ構文にマッピングするため、UML から VDM++へのマッピングで陰定義関数と陽定義関数の区別は難しくなる。このため、以下の規則が UML から VDM++への関数のマッピングに適用される:

- ある関数がすでに VDM++で定義されている場合、VDM++の中で定義されているのと同じ種類に (暗黙的にあるいは明示的に) マップされる。
- ある関数が VDM++で認識されていなければ (UML レベルで定義された関数であれば)、明示的にマップされる。


```
class Queue
```

```
types
```

```
  public Item = token;
```

```
instance variables
```

```
  q : seq of Item := [];
```

```
  inv len q <= max;
```

```
values
```

```
  max : nat = 256;
```

```
operations
```

```
  public Enqueue(i:Item) ==
```

```
    q := q ^ [i];
```

```
  public Dequeue : () ==> Item
```

```
  Dequeue() ==
```

```
    is not yet specified;
```

```
functions
```

```
  public Merge (q1 : seq of Item,  
                q2 : seq of Item) q : seq of Item
```

```
  pre IsSorted(q1) and IsSorted(q2)
```

```
  post IsSorted(q);
```

```
  IsSorted : seq of Item +> bool
```

```
  IsSorted(q) ==
```

```
    forall i,j in set inds q & i < j => LessThan(q(i), q(j));
```

```
  LessThan : Item * Item +> bool
```

```
  LessThan (i,j) ==
```

```
    is not yet specified
```

```
end Queue
```

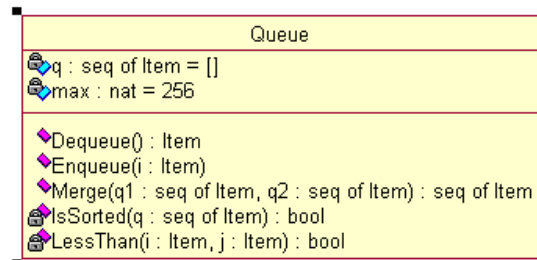


図 18: VDM++ と UML 間でクラスのマッピング

4.2 クラス間の関連

VDM++において、オブジェクト（クラスの実体）は、他のクラスのオブジェクトあるいはクラス自身のオブジェクトと関係をもつ可能性がある。このようなクライアント側の関係は、オブジェクト参照型を介して可能となる。UML ではこういった関係を関連と呼び、*client* クラスから参照されるクラスへ向けた矢印によって表示される。このように矢印は Rose との連携の方向 を表し、オブジェクトは実際に他から参照することができる。図 19 では、VDM++ と UML の間でこのように単純な関連付けがマップされる様子を示す。オブジェクト参照を表すインスタンス変数は UML クラスで属性とみなされることはない、ということには注意しよう。

```
class A
instance variables
  a: A;
  b: B;
  c: C;
end A

class B
end B

class C
instance variables
  a: A;
end C
```

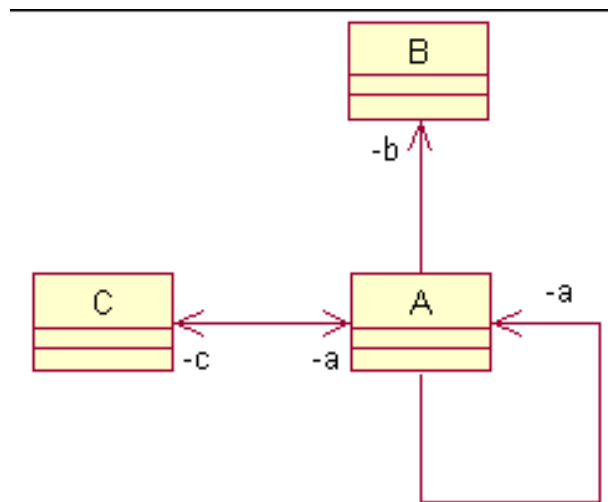


図 19: VDM++ と UML 間でのオブジェクト参照 / 関連のマッピング

UML において関連の *far end* はロールと呼ばれる。通常ロールには名前が与えられるが、少なくとも実社会モデルにおいては、関連の役割を記載する傾向がある。ロールは 2 項関連の場合には特に役立つ。ここで、関連を表すインスタンス変数の名前から簡単にロール名を構築してみよう。これは図 19 においても示されている。

関連に多重度を追加

ここまでは単純な“1対1”の関連付けを考えてきた、つまり、1つのクラスの実体がきっちり1つの他の実体に接続する関係である。seq of A や set of A といった構築要素を用いて、1つの実体をいくつかの他の実体と関係付けることができる。UML においては、関連の最後に数字 / 記号を追加し多重度を示すことで、多重度が表現される。

ここで、関連の多重度に関係する別の VDM++ 構築要素がどのように UML にマップされるかを示そう。

set of objref: この構築要素は“1対多”の関連に翻訳されるが、UML においては、関連の1端に“1”を置き多端に範囲である“0..*”を置くことでモデル化される。

seq of objref: これはまた1対多の関連であり、配列型を用いることで参照に順番を与えているため、UML では関連の多端に対して ordered の制約を追加し表示する。

seq1 of objref: オブジェクト参照の空でない配列は、関連の多側の範囲“1..*”で表示される。

[objref]: オプションのオブジェクト参照は、関連のロールに範囲“0..1”を追加することで識別される。

これらの構築要素は図 20 にまとめられている。

map と inmap を用いたオブジェクト参照

VDM++ では、オブジェクトを map 型と inmap 型を用いて関係づけることができる。VDM++ は map type to objref というように構築し、ここで type は任意の VDM++ 型、objref は単一あるいは多数のオブジェクト参照で、結果として 限定子つき関連となる。

限定子つき関連の多様性もまた、関連の他の型と同様に与えられる得るものである。UML では限定子自体に名前を割り当てることが可能で、この場合は写像の定義域の型となる。他の関連と同様、この関連のロール名はマップを表示するインスタンス変数の名称である。例題として図 21 を参照のこと。

```
class D
instance variables
  es: set of E;
  fs: seq1 of F;
end D
```

```
class E
instance variables
  opt_f: [F];
end E
```

```
class F
instance variables
  d: D;
end F
```

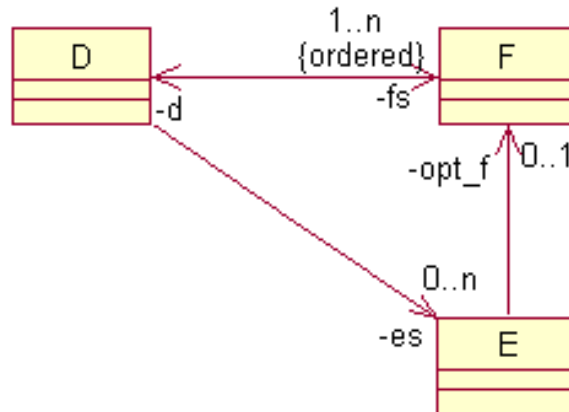


図 20: VDM++ と UML 間の多重オブジェクト参照のマッピング

```
class G
instance variables
  qual_h: map nat to H;
end G
```

```
class H
instance variables
  qual_j: map real to J;
end H
```

```
class J
instance variables
  qual_h: map nat to set of H;
end J
```

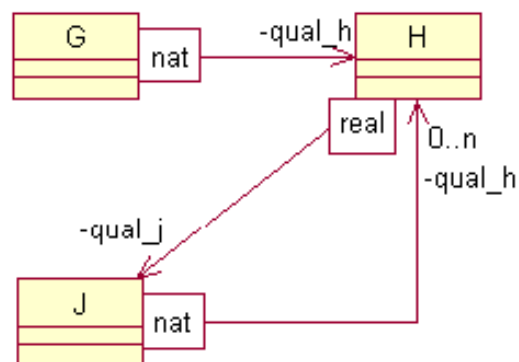


図 21: マップを用いて定義されたオブジェクト参照からの限定子つき関連

継承関係

VDM++ から UML へまたはその逆の継承の翻訳は、わかりやすい。図 22 で例題を示す。SubA クラスと SubB クラス は Super クラスから継承する。UML に

```
class Super
operations
  methodA() ==
    is subclass responsibility;
  methodB() ==
    ...;
end Super

class SubA is subclass of Super
end SubA

class SubB is subclass of Super
end SubB
```

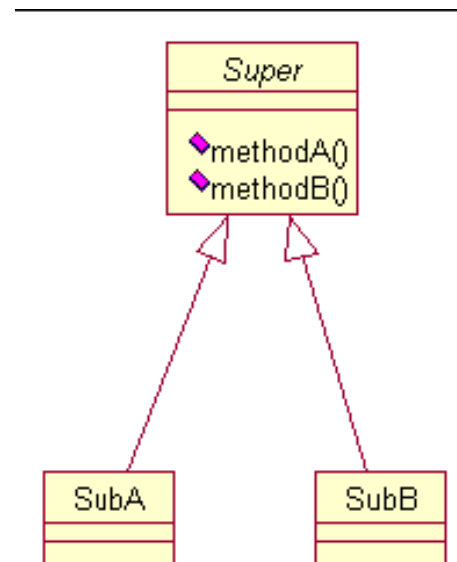


図 22: VDM++ と UML 間の継承のマッピング

において、継承関係は汎化として表示される。

委任

VDM++ では操作の仕様は、`is subclass responsibility` 節を用いてサブクラスに委任することができる。UML ではこのような操作は 抽象メソッド と呼び、抽象的な操作を含む節は 抽象クラス と呼び、具体的なクラスと対比する。少なくともその操作の 1 つを委任した (少なくとも 1 つの操作を `is subclass responsibility` を用いて指定した) 場合にクラスは抽象的と見なされ、それ以外は具体的と見なされる。UML では抽象クラスと操作は、名前をイタリックフォントで書くことで識別される。

参考文献

- [1] CSK. *VDM++ Installation Guide*. CSK.
- [2] CSK. *The VDM++ Language*. CSK.
- [3] CSK. *VDM++ Sorting Algorithms*. CSK.
- [4] CSK. *VDM++ Toolbox User Manual*. CSK.
- [5] GRADY BOOCH, IVAR JACOBSON AND JIM RUMBAUGH. The Unified Modelling Language, version 1.1. Tech. rep., Rational Software Corporation, September 1997. Available at: <http://www.rational.com/>.
- [6] RATIONAL. *Rose 98 User Manual*, 1998.

A マッピング規則のまとめ

図 23 のテーブルは Rose との連携で適用されるマッピング規則をまとめている:

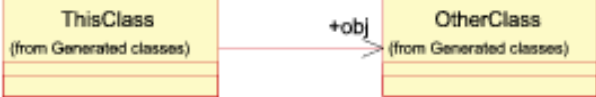

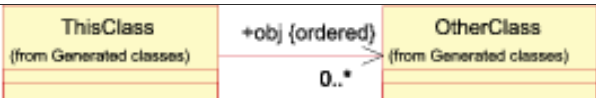
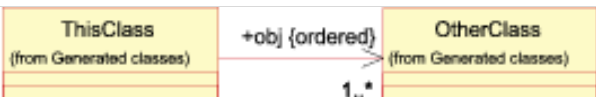
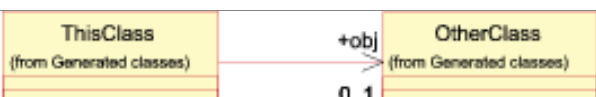
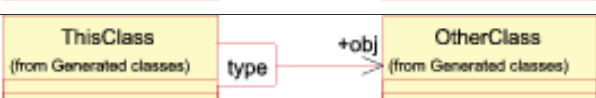
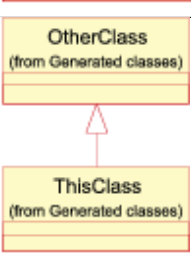
VDM++	UML
instance variable	ステレオタイプ <<instance variable>>の属性
value	ステレオタイプ <<value>>の属性
operation	ステレオタイプ <<operation>>の操作
function	ステレオタイプ <<function>>の操作。UMLでのみ定義されていて VDM++ に明白な関数としてマップされた関数。それ以外、関数は暗黙であるいは VDM++ で定義済みとして明白にマップされる。
obj: OtherClass	
obj: set of OtherClass	
obj: seq of OtherClass	
obj: seq1 of OtherClass	
obj: [OtherClass]	
obj: map type to OtherClass	
class ThisClass is subclass of OtherClass	

図 23: Rose との連携で適用されるマッピング規則

B Rose-VDM++ リンクにより生成される警告

図 1 を見直してみる。2つのモデルを併合する前に、VDM++ と UML モデルの両方が内部表現に翻訳されている。

この翻訳はいくつか検査を含み、結果いくつか別の警告がなされる可能性がある。

UML から VDM++への翻訳中に生成される警告

UML クラスが VDM++に翻訳される場合、UML 属性と操作は VDM++ 構築要素に転換される。UML モデルで使われる名称と型は、必然的に VDM++の構文上の規則に従わなければならない。従わないものがある場合、UML 定義は単に無視され、ユーザーは警告による注意を受ける。これは生成される VDM++仕様の構文的正当性を最大限にするために行われる。

1つの例題として、図 24 に示される UML クラスを見よう。

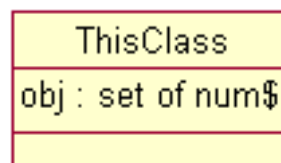


図 24: VDM++の構文規則に従わないUML クラス (\$ 記号をここで用いることはできない)

UML モデルを取り込むときに生成される警告を図 25 で示す。

警告の原因となる UML 定義を取り除かずに UML モデルから VDM++へマッピングを行う場合、結果は以下のような VDM++ クラス定義となる:

```
class ThisClass
end ThisClass
```

見ての通り、生成されるクラス定義は単なる空となる。

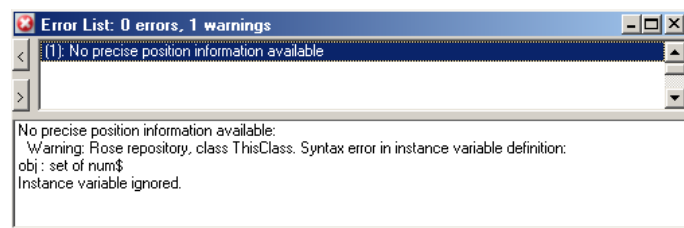


図 25: インスタンス変数の定義が VDM++ の構文規則に従わない場合、警告がなされる