



九州大学  
KYUSHU UNIVERSITY

# VDMTools

---

VDMTools ユーザマニュアル  
(VDM-SL)

VDM を活用するために



**How to contact:**

<a href="http://fmvdm.org/">http://fmvdm.org/</a>	VDM information web site(in Japanese)
<a href="http://fmvdm.org/tools/vdmtools">http://fmvdm.org/tools/vdmtools</a>	VDMTools web site(in Japanese)
<a href="mailto:inq@fmvdm.org">inq@fmvdm.org</a>	Mail

*VDMTools ユーザマニュアル (VDM-SL) 2.0*

— Revised for VDMTools v9.0.6

© COPYRIGHT 2018 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

<b>1</b>	<b>はじめに</b>	<b>1</b>
<b>2</b>	<b>VDMTools の概略</b>	<b>4</b>
<b>3</b>	<b>VDMTools ガイドツアー</b>	<b>6</b>
3.1	VDMTools で取り扱う仕様の作成	6
3.2	GUI で VDM-SL を始める	7
3.3	オンラインヘルプ	7
3.4	メニュー、ツールバー、サブウィンドウ	8
3.5	プロジェクトを作成する	10
3.6	VDM 仕様の構文チェック	11
3.6.1	仕様の解析	11
3.6.2	構文エラーの修正	12
3.7	VDM 仕様の型チェック	15
3.8	仕様の検証	18
3.8.1	インタープリタを使用した式の評価	19
3.8.2	ブレイクポイントの設定	21
3.8.3	動的型チェック	24
3.8.4	証明課題のチェック	26
3.9	体系的テスト	31
3.10	清書機能	32
3.11	コード生成	34
3.12	動的リンク機能	34
3.13	VDMTools API	34
3.14	VDMTools の終了	34
<b>4</b>	<b>VDMTools リファレンスマニュアル</b>	<b>35</b>
4.1	GUI 全般	35
4.1.1	プロジェクト・ハンドリング	36
4.1.2	仕様の操作	39
4.1.3	ログウィンドウ、エラーリストウィンドウ、ソースウィンドウ	40
4.1.4	ファイルの編集	41
4.1.5	インタープリタを使う	42
4.1.6	オンラインヘルプ	42

---

4.2	コマンドラインインターフェース全般	42
4.2.1	ファイルの初期化	45
4.3	構文チェック機能	46
4.3.1	GUI	46
4.3.2	構文エラーのフォーマット	47
4.3.3	コマンドラインインターフェース	47
4.3.4	Emacs インターフェース	48
4.4	型チェック機能	50
4.4.1	GUI	51
4.4.2	エラーおよびワーニングのフォーマット	51
4.4.3	コマンドラインインターフェース	53
4.4.4	Emacs インターフェース	54
4.5	インタープリタとデバッガ	56
4.5.1	GUI	56
4.5.2	スタンダードライブラリ	63
4.5.3	コマンドラインインターフェース	65
4.5.4	Emacs インターフェース	66
4.6	証明課題生成機能	71
4.7	清書機能	73
4.7.1	GUI	74
4.7.2	コマンドラインインターフェース	74
4.7.3	Emacs インターフェース	76
4.8	VDM-SL から C++コード生成	77
4.8.1	GUI	77
4.8.2	コマンドラインインターフェース	77
4.8.3	Emacs インターフェース	78
4.9	VDM モデルの体系的テスト	80
4.9.1	テストカバレッジファイルの準備	81
4.9.2	テストカバレッジファイルの更新	81
4.9.3	テストカバレッジの統計データ作成	82
4.9.4	L <sup>A</sup> T <sub>E</sub> X を使ったテストカバレッジ例	83
	用語集	90
A	VDM 技術の情報源	92

<b>B</b>	<b>VDM-SL と <math>\text{\LaTeX}</math> の結合</b>	<b>95</b>
B.1	仕様ファイルのフォーマット . . . . .	95
B.2	$\text{\LaTeX}$ 文書のセットアップ . . . . .	95
<b>C</b>	<b>VDMTools 環境の設定</b>	<b>99</b>
C.1	一般 . . . . .	99
C.2	インターフェースオプション . . . . .	100
<b>D</b>	<b>Emacs インターフェース</b>	<b>102</b>
<b>E</b>	<b>Sort 例題向けテストスクリプト</b>	<b>103</b>
E.1	Windows/DOS プラットフォーム . . . . .	103
E.2	UNIX プラットフォーム . . . . .	104
<b>F</b>	<b>Microsoft Word についてのトラブルシューティング問題</b>	<b>106</b>
	<b>索引</b>	<b>107</b>



## 1 はじめに

**VDMTools** は、コンピュータシステムの精巧なモデルを開発・分析するツールである。システム開発の早期に導入すれば、これらのモデルはシステム仕様として、あるいはユーザの要求の網羅性や整合性のチェックを助けるものとして役立つ。モデルは、ISO VDM-SL 標準言語 [5]、あるいは、オブジェクト指向形式仕様言語 VDM++ [8] [3] によって表現される。本マニュアルでは、実装に先立ち、VDM-SL で表現されたモデルの自動チェックと検証を行うためのツールである VDM-SL ツールボックスについて記述する。その範囲は、従来からの構文と型のチェックツールから、必要に応じてモデルを実行し、実行中には自動的に整合性チェックを行う強力なインタープリタに及ぶ。実行機能は、分析・設計の早期からテスト技術の利用を可能にするとともに、確立されているソフトウェアエンジニアリングの実践に即した全体的なテストの実行を可能にする。そのうえ、このインタープリタでは、ブレイクポイントの設定、文のステップ実行、表現の評価、コールスタックの調査、スコープにおける変数の値のチェックなど、モデルのインタラクティブなデバッグが可能である。

本ドキュメントには VDM-SL ツールボックス（この文書では Toolbox と呼ぶ）の紹介とリファレンスマニュアルを記載する。VDM-SL 言語には別に言語マニュアルがある。[10]。このマニュアルでは、仕様という言葉は目的を問わず本言語で構成されたすべてのモデルを指すものとして使う

## VDM 入力フォーマット

Toolbox は MS Word または L<sup>A</sup>T<sub>E</sub>X の文書に埋め込まれた VDM-SL の仕様をサポートしているので、仕様を抜き出したファイルを別途作ることなく、仕様の分析を行うことができる。システムのモデルとその文書を統合して扱う方法として、これらの書式を使用することを推奨する。モデルの記述と文書を統合して扱うことで、最新版の仕様と文書化されている仕様の不整合を避けることができる。もちろん、Word や L<sup>A</sup>T<sub>E</sub>X を使用しなくてもよく、好みのテキストエディタを使って、シンプルなテキストファイルの仕様を書くことも出来る。

仕様は日本語を使って記述することができ、Toolbox を使って解析することができる。Toolbox は複数の文字コードを入力文書としてサポートしている。付録 C.1 で Toolbox への文字コードの設定方法を説明している。

MS Word を使用して VDM-SL の仕様を作成する場合は、仕様を記述した文書を *Rich Text Format* (RTF) フォーマットで保存しなくてはならない。Toolbox の配布パッケージには、このフォーマットでのサンプルファイルが含まれる。このマニュアルでは、Toolbox の配布パッケージに含まれるそれらのファイルを使った例を参照している。例題のファイルで、“`.rtf`” が拡張子がついていれば、リッチテキストフォーマット (RTF) のファイルであることがわかる。

このマニュアルでは、Toolbox の機能の紹介を Word の RTF を前提として行う。L<sup>A</sup>T<sub>E</sub>X で仕様を作成するのであれば、付録 B に書かれているフォーマットとスタイルを使って L<sup>A</sup>T<sub>E</sub>X のコマンドに VDM 仕様を埋め込む方法を参照してほしい。Toolbox にはこのフォーマットでもサンプルファイルが入っているが、拡張子は “`.rtf`” ではなく “`.vdm`” である。そのため、例が `sort.rtf` というファイルを参照していた場合、代わりに `sort.vdm` というファイルを使わなくてはならない。ディレクトリ構造の参照はこのマニュアルを通じて以下のような形式で示される。

`examples/sort.vdm` (スラッシュ区切り) Windows では `examples\sort.vdm` (英語ではバックスラッシュ区切り). と同じである。

シンプルなテキストファイルとして仕様を記述した場合、説明文を仕様に組み込む唯一の方法は、言語マニュアルに記載されている VDM-SL コメント構文を用いることだ。このフォーマットについては通常拡張子 “`.vdm`” でファイルが用意されている。

## このマニュアルの使い方

このマニュアルは 3 つの部分に分かれる。セクション 2 と 3 は Toolbox のさまざまなツールの概略と GUI を利用した Toolbox のチュートリアルを提供する。マニュアルのこの部分を実際に試してみる前に、Toolbox がインストールされていなくてはならない (付録 C 参照)。Toolbox のインストールについてはこの文書に記述される [9]。セクション 3 を読み進むにつれ、さまざまなツールや制御コマンドを利用できることがわかるだろう。

第 2 のパート (セクション 4) は、Toolbox のシステムのすべての機能をカバーするリファレンスガイドである。3 つの利用可能なインターフェースすべて (コマンドラインインターフェース、Emacs インターフェース、GUI) について、それぞれの機能を記述してある。



このマニュアルの第3のパートは一連のトピックスについての付録で構成されている。付録 **A** は VDM の情報源について記載されているが、これにはインターネットのサイト、プロジェクトの記述、技術論文や参考文献が含まれている。付録 **B** では L<sup>A</sup>T<sub>E</sub>X の文書でどのようにテキストと仕様をマージするかを説明している。

付録 **C** では環境を Toolbox 向けにどう設定するかが記述されている。付録 **D** では Emacs インターフェースについて。付録 **E** には Sort 仕様（このマニュアルで実行可能な例として使われている）のシステムテストに使うテストスクリプトがいくつか含まれている。そして付録 **F** では Toolbox を使っていると見られる Microsoft Word においての一般的な問題について、考えうる解決策をいくつか提供している。

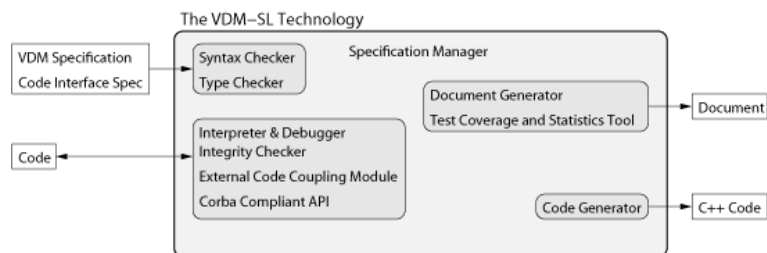


図 1: VDMTools の概略

## 2 VDMTools の概略

VDM-SL 仕様は、システムの特性を厳密に記述することを目的としたドキュメントである。この仕様は、セクション 1 で記述されている入力フォーマットで書かれた複数のファイルに分かれていることもある。図 1 は Toolbox の機能の概要とその付随的な機能を示したものである。ツールについては下記に記述する。

**マネージャー** 仕様に定義されているモジュールの状態を常に保持する。仕様は、複数のファイルで構成されていることもあり、それらすべての状態を保持する。

**構文チェック機能** VDM-SL 言語の定義に照らして、VDM-SL の仕様の構文が正しいかどうかチェックする。構文が受け入れられれば、Toolbox 内の他ツールの利用が可能となる。

**型チェック機能** 強力な型推論メカニズムを持ち、値や演算子の誤用を特定する。またランタイムエラーの起こりうる箇所も示す。

**インタープリタとデバッガ** インタープリタは、VDM-SL で実行可能な構成要素すべてを実行する。その範囲は、集合の内包式や列の列挙式など単純な構成子から、より複雑な構成要素（例外処理、ラムダ表現、ルーズな表現やパターンマッチング）にも及ぶ。仕様を実行することの利点の一つは、テスト技術がこれらの検証に役立つということだ。開発プロセスにおいて、仕様の小（大）部分が設計者の理解と自信を深めるのに有効である。その上、実行可能な仕様は実行プロトタイプを形成する。

ソースレベルのデバッガは、実行可能な仕様を用いて作業をする上で、必要不可欠な機能である。VDM-SL デバッガは、ブレイクポイントの設定、

ステップ実行、スコープ内で定義された変数の値チェック、コールスタックのチェックなど、通常のプログラミング言語向けのデバッガと同様の機能をサポートする。

**証明課題生成機能** 証明課題生成機能は、VDM-SL ツールボックスの型チェックの機能を拡張するものである。仕様全体をスキャンして、内的矛盾点や整合性を侵害しうる潜在的なソースをチェックする。データ型の不変条件、事前条件、事後条件、列の境界や写像のドメインの違反チェックが含まれる。証明課題はVDM-SL の式で表され、true と評価されなければならない。もし false と評価されていたら、それは仕様の相当する箇所に潜在的な問題があることを示している。

**テスト機能** テスト機能ではテストスイートと呼ばれる予め用意されたテストセットを使って、仕様の実行が可能である。テストカバレッジ情報は、テストスイートの実行中に自動的に記録され、あとから、仕様のどの部分が頻繁に評価されているか、どの分が全くカバーされていないかなどを、表示する。テストカバレッジ情報は、仕様が Word または  $\text{\LaTeX}$  で記述されている場合には、ソースファイル文書に直接表示することができる。

**自動コード生成** Toolbox は、VDM-SL の仕様から C++ のコードを自動生成する機能をサポートしており、この機能により仕様と実装の間に一貫性を持たせることができる。コード生成機能はVDM-SL の構文の 95% から実行可能なコードを生成する。仕様の実行不可能な箇所向けに、ユーザによる定義コードを含めることを可能にする機能も併せ持つ。いったん仕様がテストされれば、コード生成機能を迅速な実装を自動的に得る手段として利用できる。C++ のコード生成機能の使い方は本ドキュメント [12] に記載されている。

**動的リンク機能** 外部のコードを仕様の実行に加えることを可能にする。この機能により、伝統的な方法で開発されたコンポーネントで構成される形式モデルを結合したり、モデルにグラフィカルなフロントエンドを提供するといったことが可能になる。

**Corba 対応 API** Toolbox は Corba 対応 API を提供しており、これにより実行中の Toolbox に他のプログラムがアクセスできる。これは型チェック機能やインタープリタ、デバッガなどの Toolbox のコンポーネント外からの制御を可能にするものである。API を使えば、グラフィカルなフロントエンドやレガシーコードなど、どんなコードからでも Toolbox へのアクセスが可能になる。

### 3 VDMTools ガイドツアー

このセクションでは、Toolbox の「ガイドツアー」を提供する。VDM によるシステムモデリングを新たに学ぼうとするのであれば、“ソフトウェア開発のモデル化技法” [2], J. フィッツジェラルド、P.G. ラルセン著、または、“VDM++によるオブジェクト指向システムの高品質設計と検証” [4] を最初に読むことをお勧めする。これらのチュートリアルブックには、Toolbox を使って探索可能な、VDM 仕様で書かれたさまざまな例題が掲載されている。[2] では ISO 標準の VDM-SL を使って記述されており、一方、[3] では VDM++ と呼ばれる、オブジェクト指向拡張された言語を使って記述されている。これらの一般的な概念を知っているが、標準的な記法に詳しくない人には、VDM-SL の言語リファレンスマニュアルである “*The VDM-SL Specification Language*” [10] を一読することをお勧めする。

#### 3.1 VDMTools で取り扱う仕様の作成

Toolbox を使用するためには、VDM-SL の仕様を書いておく必要がある。このセクションでは、シンプルなソートのサンプルを作成することを通じて、MS Word のリッチテキストフォーマットを使用した方法を記述する。L<sup>A</sup>T<sub>E</sub>X 文書を使いたい場合は、付録 B<sup>1</sup>を参照のこと。このセクションでは、MS Word を使用すると想定していることを覚えておいてほしい。

MS Word を起動して、Toolbox から `vdmhome/examples/sort/sort.rtf` ファイル<sup>2</sup>を開く。このファイルを一通り読めば、

このドキュメントが説明文と VDM-SL 形式のモデルであることがわかるはずだ。形式の部分は VDM スタイルですべて書かれている。VDM スタイルの文書から直接元の文書に戻すのはおそらく無理だろう。通常のプリンタは VDM のキーワードを太字のフォントで印刷する。このスタイルの外観を修正することはできるが、スタイルの名前は変えられない。なぜなら Toolbox が VDM スタイルで書かれた文書の部分だけを解析するからである。

Toolbox 内で使用されるスタイルの定義は Toolbox の中にある `VDM.dot` ファイル (Word 形式) に記述されている。このファイルはテンプレートディレクトリ (通

<sup>1</sup> プレーンテキストのみの VDM-SL を使用することも可能

<sup>2</sup> `vdmhome` は Toolbox のトップディレクトリ。Windows の場合、通常は、`C:\¥Program Files ¥The VDM-SL Toolbox v?.?` となる

常は C:\Program Files\Microsoft Office\Templates ) にコピーすることで、新しくドキュメントを作成した場合にテンプレートを選択すると（普通に使用していれば、このほかにもさまざまな方法でテンプレートディレクトリにスタイル定義がコピーされる）これらのスタイルの定義が含まれることになる。

sort.rtf ファイルの最後を見てみよう。VDM\_TC\_TABLE 形式の空の行があるのがわかるだろう。この使い方は後でテストカバレッジ情報の記録・表示方法についての話をするときに詳しく述べる。この VDM\_COV および VDM\_NCOV 形式はテストカバレッジの情報と関連付けるときにも使われる。これらの形式についても後ほど述べる。

さらに MS Word を使用して Toolbox への入力物を作成する経験を積む場合は、「ガイドツアー」を終えた後に Toolbox 内のほかのサンプルファイルを読むことをお勧めする。

## 3.2 GUI で VDM-SL を始める

Toolbox は通常 GUI を使う。GUI を使い始める前に、VDM のソースファイルがワーキングディレクトリにコピーされていなければならない。Toolbox は異なる種類のソートアルゴリズムの仕様を含んでいるが、これはテクニカルリポート [11] に書かれているとおりである。このガイドツアーでは、このソート仕様をサンプルとして使うため、vdmhome/examples/sort ディレクトリをコピーしてそこへディレクトリを移動してもらいたい。これで次からのツアーをあなたの環境で Toolbox 内のツールを直接試すことが出来るようになったはずだ。

Toolbox は Windows のスタートメニューから選択するか、Unix 環境であれば vdmgde コマンドで起動する。図 2 は Toolbox の起動画面である。このウィンドウを Toolbox のメインウィンドウと呼ぶ

## 3.3 オンラインヘルプ

Toolbox のオンラインヘルプと一般的なインターフェースはヘルプツールバーやヘルプメニューからアクセスできる。最近では以下に示す限られたものだけが利用可能である。

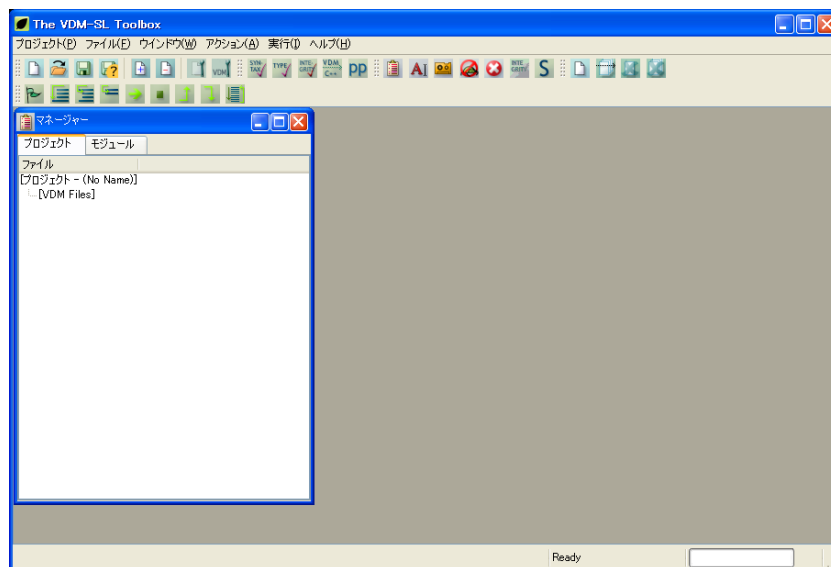


図 2: スタートアップ画面

ツールについて (?): Toolbox のバージョン番号を表示する.

Qt について (Q): Qt (Toolbox のインターフェースが利用している、C++のマルチプラットフォーム GUI ツールキット) へのリファレンス情報を表示する

### 3.4 メニュー、ツールバー、サブウィンドウ

メインウィンドウの上部は6つのプルダウンメニューが一行になって構成されている。

**プロジェクト:** プロジェクトメニューは VDM-SL の仕様を構成するファイル名の集合で構成されている。このメニューからはプロジェクトを開く／保存する、プロジェクトの設定（ファイルの追加／削除）をする、新規プロジェクトの作成などができる。Toolbox を終了させたり、Toolbox のさまざまなツールのオプションを設定する機能もここにある。（例えば型チェックのレベルを設定など）

**ファイル:** ここからは仕様を訂正するためファイルエディタが起動できる。またエラーが報告されたとき、Toolbox によって自動的に表示されるソースファイルの表示を終了させることが出来る。

**ウインドウ:** メインウインドウの画面に表示されているウインドウをコントロールする。メニュー項目を使って、対応するウインドウの表示／非表示を行うことができる。

**実行:** インタープリタのコントロール機能を提供する（セクション 3.8.1 参照）。構文チェックや型チェック、証明課題の生成、コード生成、清書など、仕様に適用されるさまざまなアクションを提供する。

このメニューの下には、メニューと同じ機能を提供するツールバーが配置される。

メインウインドウの下部の画面は、現在のプロジェクトの状態に関する情報や Toolbox 内のツールへのインターフェースを提供するさまざまなサブウインドウを表示するのに使用される。利用できるウインドウは以下のとおり:

**マネージャー** 現在のプロジェクトの最新状態を表示する。以下の 2 つのビューから構成される。

**プロジェクトビュー** プロジェクトの内容をツリー形式で表示する。プロジェクトの構成ファイルとそれぞれで宣言されているモジュール（ファイルの構文チェックが成功したもののみ）が含まれる。

**モジュールビュー** プロジェクトに含まれる VDM-SL モジュールの個々の状態を表示する。

**ソースウインドウ** 仕様を表示する。エラーが発生した場合は、エラー一覧でエラーを選択した場合は、その位置を表意する。

**ログウインドウ** Toolbox からのメッセージを表示する

**実行ウインドウ** インタープリタとのインターフェース。

**エラー一覧** Toolbox によって発見されたエラーリポート。

**証明課題ウインドウ** 仕様から生成された証明課題を表示するウインドウ。

**識別子検索ウインドウ** 仕様から識別子を検索する。定義部分のみや部分一致による検索ができる。

Toolbox の起動時には、マネージャーのみが開いている。



### 3.5 プロジェクトを作成する


まず、どのファイルを分析にかけるかを Toolbox に設定する必要がある。このため、プロジェクトメニューからファイルを追加を選択するか (プロジェクト) ツールバーから  (ファイルを追加) ボタンを押す<sup>3</sup>。すると、図 3 に示すようなダイアログボックスが表示される。



図 3: プロジェクトにファイルを追加する

sort-init.rtf ファイルをダブルクリックする (またはファイルを選択のうえ “開く” ボタンを押下してプロジェクトへファイル追加)。このファイルがプロジェクトに追加されているはずだ。Ctrl キー<sup>4</sup>を押しながらマウスの左ボタンをひとつずつ順番に押すことで一度に複数ファイルを選択することができ、Shift キーを押しながらファイルリストの最初と最後を選択する (順番はどちらでもよい) ことで一覧のファイルを選択することもできる。sort-init.rtf にはこのガイドツアーで見せるの目的で、エラーがいくつか入っていることに注意してほしい。

sort-init.rtf ファイルは、下記図 4 に示す、Toolbox のメインウィンドウにある、マネージャー のプロジェクトビューに表示されているはずだ。

<sup>3</sup> このガイドツアーでは、ツールバーのボタンを使った操作を中心に話を進めている。メニュー項目を使っても同じ操作を行うことができる

<sup>4</sup> この操作を行うためのキーは OS に依存する。Windows や Linux では Ctrl キー、Mac OS X では Command キーとなる。



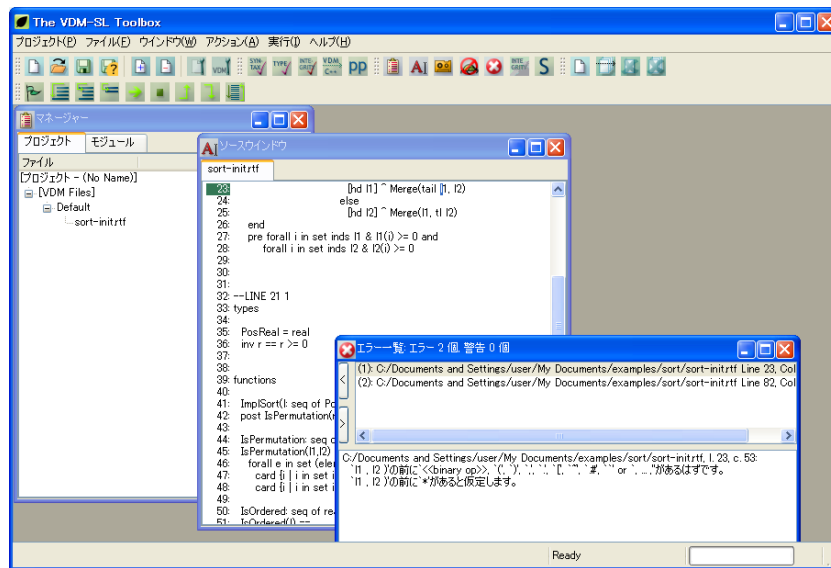



図 4: ファイル追加後のメインウィンドウ

### 3.6 VDM 仕様の構文チェック

プロジェクトを作成したら、すべてのモジュールが、VDM-SL の構文ルールに準じているかチェックする必要がある。構文チェック機能は、作成した仕様の構文が正しいかどうかチェックする。プロジェクトにファイルを追加したり、プロジェクトに登録されているファイルが変更されたりすると、ツールは自動的に構文チェックを行う。ツールの設定で、自動的な構文チェックを行わないようにすることもできるが、その場合は、ファイルを変更したら、ツールの他の機能を使用する前に、再度構文チェックをしなくてはならないことを忘れないでもらいたい。

#### 3.6.1 仕様の解析

マネージャーのプロジェクトビューで `sort-init.rtf` ファイルをクリックし、(アクション) ツールバーの  (構文チェック) を押すと、構文チェック機能が起動する (「Default」フォルダのレベルを選択し、構文チェックの操作を適用することで、同じ操作を行うことができる。つまり、そのフォルダ内の各ファイルに操作が適用される)。ログウィンドウに、“Parsing “sort-init.rtf” ...” のメッセージが表示されていることに、注目してほしい。(ファイルは複数選択することもで

き、その場合は選択されたファイルのすべての構文がチェックされる。) ログウインドウには、エラー情報が出力されている場合があるので、問題解決の手がかりになることがある。構文エラーが発見された場合はエラー一覧のウインドウが自動的に起動される。また、ソースウインドウも表示される。ソートのサンプルには、説明のために、二つ構文エラーがはいっている。

### 3.6.2 構文エラーの修正

図 5 にエラー一覧を示す。画面の上部に、エラーまたは警告の生じた箇所のリスト（ファイル名、行番号、カラム番号）が表示され、下部には選択中のエラーの詳細情報が表示される。最初はリストの先頭のエラーが自動的に選択される。

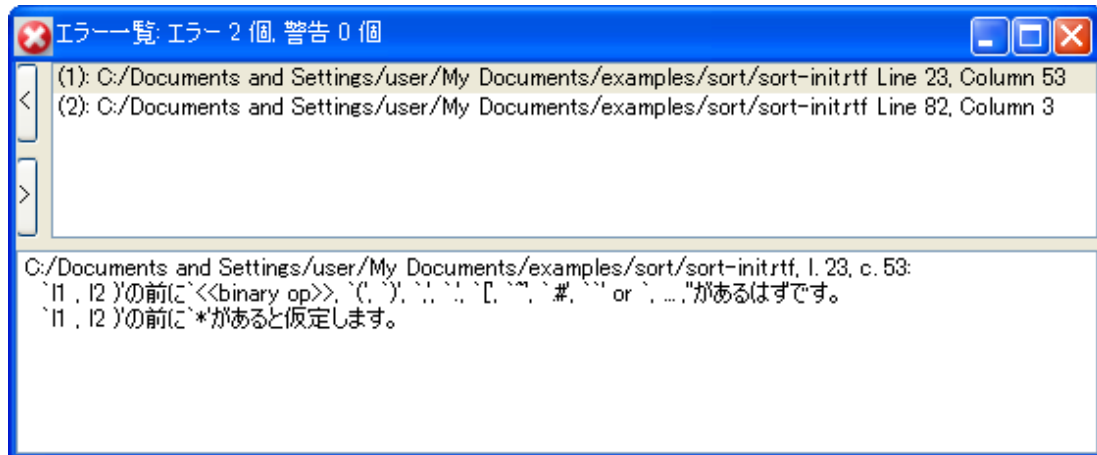


図 5: エラー一覧

ソースウインドウには、エラー一覧内で選択中のエラーに対応する、仕様の部分が表示される。実際の箇所は、ウインドウのカーソルでマークされて示される。最初の構文エラーに対しては、ソースウインドウは図 6 のように表示される。

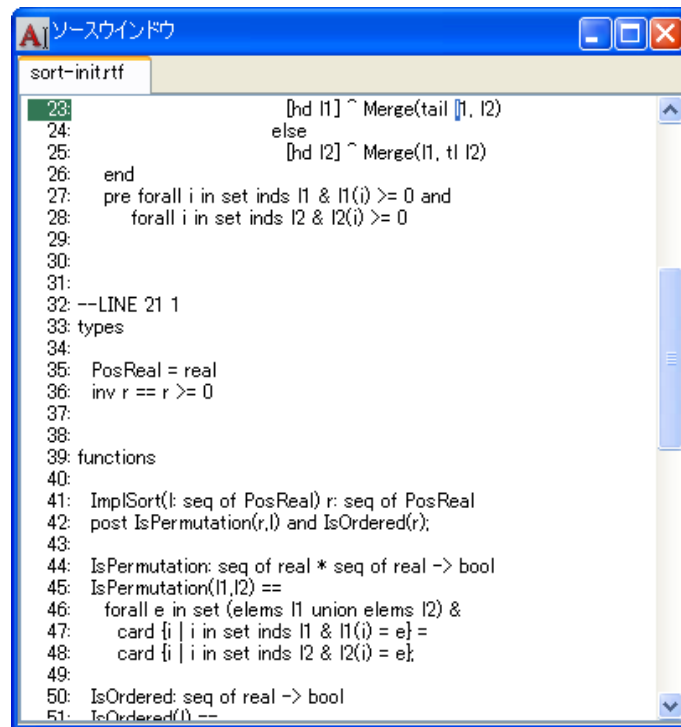


図 6: 最初のエラーがソースウインドウに表示された所

最初のエラーメッセージは下記のとおり：

C:/Documents and Settings/user/My Documents/examples/sort/sort-init.rtf, l. 23, c. 53:


‘l1 , l2 )’ の前に ‘<<binary op>>, ‘(’, ‘)’, ‘,’ , ‘.’, ‘[’, ‘- ’, ‘.#’, ‘’’ or ‘, ... ,’’ があるはずですが。  
 ‘l1 , l2 )’ の前に ‘\*’ があると仮定します。


この形式のメッセージは、エラーの発見されたポイントで、予測されるものが見つからなかった場合に表示される。構文チェック機能は、エラーを報告し、起こった箇所において、修正および構文チェックの続行を行うための仮説を示す。

この例では、エラーメッセージによれば、構文チェックを続けるためには、‘tail’ と ‘l1’ の間に記号 ‘\*’ が必要だと予測されている (‘tail’ は識別子の名前とみなされている)。しかしこの想定は間違っている。言語マニュアル [10] の VDM-SL の記述から、このエラーは、実際には、列から先頭を除いたものを返す ‘tl’ 演算子

が間違っって ‘tail’ と書かれているために起こったものとわかる。そのため、このエラーは、ファイルエディタを使い ‘tail’ を ‘tl’ に書き換えれば修正となる。

(構文チェック機能が仮説を示しても、元ファイルは変更されないことに注意してほしい。元ファイルの修正は、ユーザの手作業で行なわねばならない。)

構文エラーの修正は、Toolbox 上から好みのエディターを直接起動することで出来るようになる。(付録 C 参照) メインウィンドウで sort-init.rtf ファイルを選択し、(ファイル) ツールバーの外部エディタ ボタン (  ) を押す。

エラーリストの左側に表示されている  ボタンを押すか、エラー一覧画面の画面上半分に表示される、エラー箇所の概要を直接選択すると、見たいエラーリポートを読むことが出来る。以下のように説明がされている。

C:/Documents and Settings/user/My Documents/examples/sort/sort-init.rtf, l. 82, c. 3:

```
‘InsertSorted : PosReal *’ の前に ‘<<binary op>>, ‘(’, ‘.’, ‘;’,
‘operations’, ‘state’, ‘traces’, ‘functions’, ‘mease’, ‘.#’,
‘post’, ‘pre’, ‘types’ or ‘values’ ’ があるはずです。
‘InsertSorted : PosReal *’ の前に ‘;’ があると仮定します。
```

これは、‘InsertSorted : PosReal \*’ の前で構文エラーが起こっており、修復するためにはセミコロン ‘;’ が必要なのではないかと仮定している。この場合、仮定は正解で、[\[7\]](#) にある VDM-SL の構文の記述からもわかるように、2つの関数定義はデリミタ ‘;’ で区切られていなくてはならない。ゆえに、このエラーは、ファイルエディタに戻って ‘;’ を関数 DoSort の定義の後ろに追加することで修正される。

構文エラーを修正してファイルを保存したら、正しく修正できていることを確認するために、再度構文チェック機能を走らせなくてはならない<sup>5</sup>。今回、仕様は構文的に正しいはずであり、マネージャー 内にあるモジュールビュー のウィンドウを選ぶと、仕様を表すモジュール (この例ではモジュール構造を何も使用していないため、DefaultMod と呼ばれるモジュール) の状態が確認できる。

さらに、構文的に正しいことを示す記号  が構文の欄についているはずだ。(図 7 参照) その他の欄がブランクなのは、仕様の型チェックやコード生成、清書など

<sup>5</sup>標準の設定では、外部エディタで仕様を修正し保存した後、Toolbox のウィンドウに戻ると、自動的に構文チェックが行われる

がまだ1度も行われていないことを意味している。

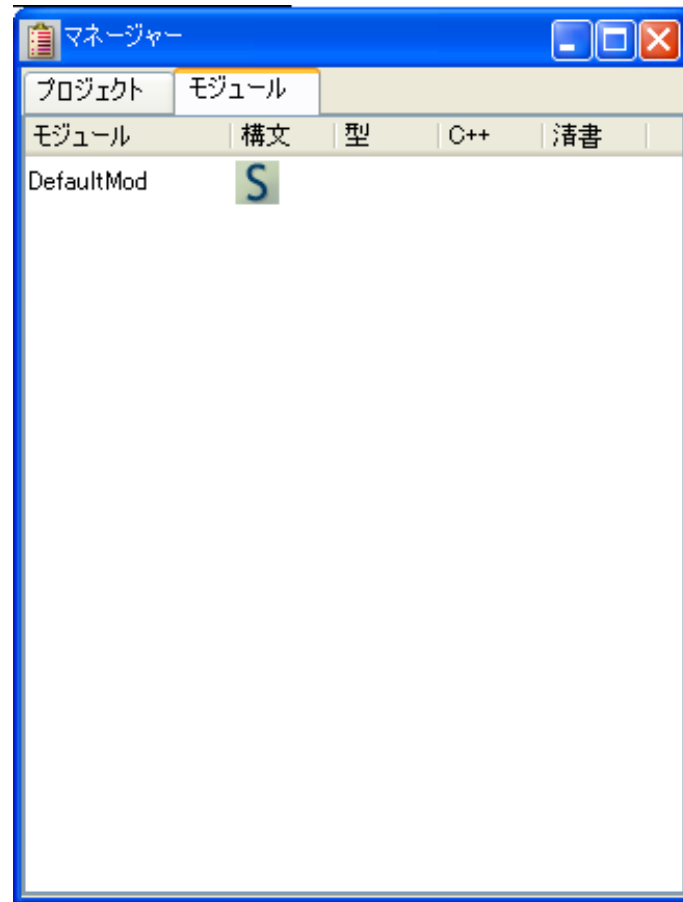





図 7: モジュールビュー

構文チェックが成功したので、ファイルをモジュールビュー から直接選択して、次の処理に進むことができる。

### 3.7 VDM 仕様の型チェック

仕様が構文チェックをパスしたら、型チェックを行うことが出来る。型チェックは (アクション) ツールバーの  (型チェック) ボタンを押すと起動する。

モジュールビュー で モジュール DefaultMod を選択し、型チェックを行う。型チェックが終わると、Toolbox は、状態表示を更新し、記号  と  (赤いライ

ンのついた **T**) を使って、各々のモジュールに関して型チェックが成功したか失敗したかを示す。

この例では、説明のために、Sort の仕様の型チェックが失敗するようになっている。エラーが4つ<sup>6</sup>と警告が1つ発生しする。これらは、構文エラーと同様、エラー一覧 に表示される。

図 8 に表示されている最初のエラーは、関数 Merge が、実際は2つの real 型の列を、引数にして呼ばれている（ソースウインドウ の表示については図 9 を参照のこと）ことを示す。しかし、想定されるパラメータは（Merge のシグネチャから見ると）int（整数値）の列と bool（ブール値）の列となっている。int は real のサブタイプであるため、列内の数字がすべて整数値であった場合には、第1引数の型は正しくなりうる。しかし、第2引数は、ブール値の列と想定されているので、問題が生じる。このエラーは、Merge 関数のシグネチャにおいて、bool を int に変更することで修正される。

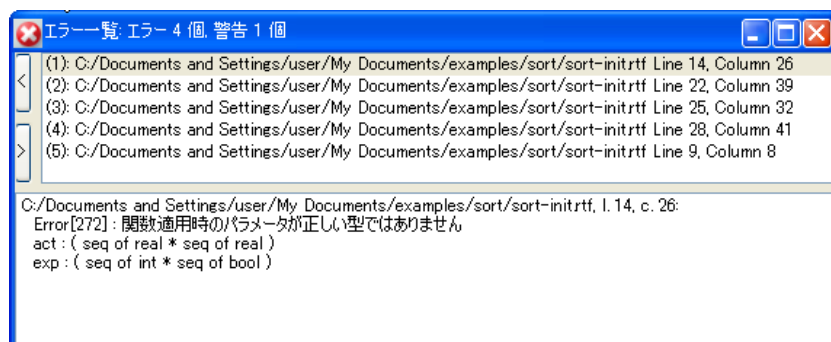


図 8: 型チェック後最初のエラー表示

図 10 に示される2番目のエラーは、数値型でない右辺に '<=' 演算子を適用しようとしたために起きたものである。もっと詳しく言えば、期待される引数の型（エラーメッセージ中では exp: と表示）が real 型（実数型は最も一般的な数値の型である）であるのに対して、実際の引数の型（エラーメッセージ中では act: と表示）は bool 型であるため型の不整合が生じている。エラーの原因を特定しようとする場合、このような情報は有用な場合がある。

このエラーは、Merge 関数のシグネチャに、seq of int であるはずのものが seq of bool と書かれていることが原因である。3番目のエラーも '<=' 演算子の左辺

<sup>6</sup>型エラーのフォーマットについては、このマニュアルのリファレンス部分に詳細が記述されている。セクション 4.4 参照

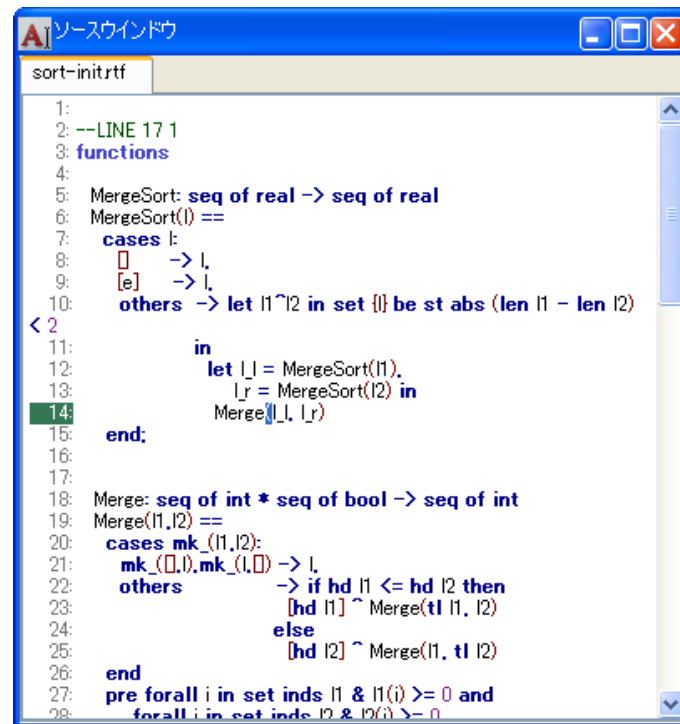


図 9: 型エラー時のソースウインドウ

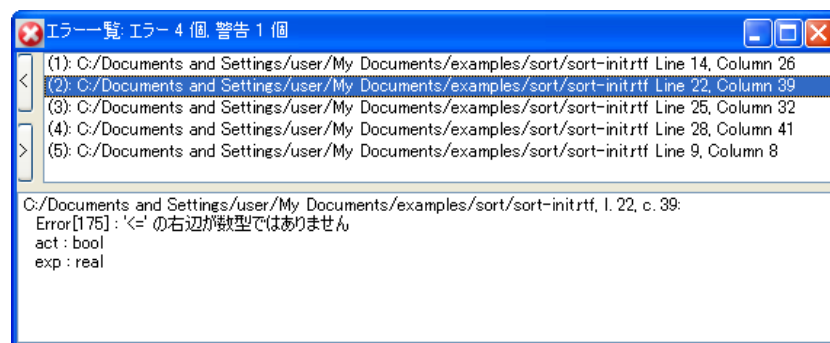


図 10: 2 番目の型チェックエラー表示

が右辺となっているだけで、同様である。そのため、これらのエラーは、最初のエラーに関連して出たものである。最初のエラーを修正して、再度、仕様の構文チェック、型チェックを行う。

メインウインドウの状態についての情報が、この処理中どのように更新されたか注目してほしい。まず元ファイルが編集されると、モジュールビューでそのファ



イルが構文的に正しいことを示す記号 **S** が、Toolbox 上に現在あるバージョンとファイルシステムにあるバージョンの不整合があることを示す **S** に変わる<sup>7</sup>。このファイルは、処理を行う前に再度構文チェックが行われていなくてはならない。次に、構文チェックの後、再度型チェックを走らせると、両方の処理が正しく終了したことを示す記号 **S** と **T** がそれぞれの状態表示する場所に表示される。

型チェック処理は、警告を返してきたとしても成功であることに注意。これは警告は、通常、実際のエラーではなく、仕様の中の冗長性を示している。例えば、あるブール式がいつも false であると予想されたり、特定のパラメータやローカル変数が関数や操作中で一度も使用されないなどである。もちろん、このような冗長性は、実際のエラー（警告を出している式や文にタイプミスがあるような）に起因するかもしれない。そのため、警告をチェックすることは、そのようなことがないかどうかを確認するためにも有用である。例では、MergeSort 関数内の cases 式の 2 番目のパターンにあるローカル変数 'e' が、一度も使用されていないという警告が表示されている。これは、現実には、エラーではなく、仕様は正しく意味をなしている。ただし、警告を除去したいのであれば、'e' を '-'（“don't-care” pattern）に置き換えればよい。

仕様が型チェックをパスしても、正しいことを保障するものではなく、まだエラーが潜んでいるかもしれない（なんらかのプログラミング言語のコンパイラの構文チェック、型チェックをパスしたとしても、0 除算によるランタイムエラーなどがありうるように）。このような、モデル中のランタイムエラーの元となりうる潜在的なところを特定する一助とするために、型チェック機能には、仕様中でランタイムエラーを起こしうる潜在的なところをすべてエラーとして報告するオプションがある。このオプションについての情報は、マニュアルのセクション 4.4 を参照のこと。

### 3.8 仕様の検証

仕様はある目的のために作成される：通常は、提案されたコンピュータシステムの設計された振る舞いの理解を深めるためや、安全性などの特性を考慮して設計されているかチェックするため、次の詳細設計や実装の基礎に役立てるためなどだ。その目的は何であれ、単に仕様の構文や型が正しいだけでは不十分で、たとえば抽象的なレベルであっても、モデル化されたシステムの振る舞いを忠実に表現

<sup>7</sup>自動構文チェックによりこの記号をみることはないかもしれない




していなくてはならない。

検証は形式仕様が、モデル化されたシステムの非形式に表現された要求を正確に反映しているかどうかについて自信を深めるプロセスである。形式仕様言語の仕様があれば、広範囲の検証技術が使える:仕様は詳細に調べることができ、テストもできる。仕様が設計された特徴を表現しているかについて極めて厳格な試験を実施することも可能だ。Toolbox はアニメーションとデバッガ（仕様の一部に値を入力して実行）とインタープリタを使ったテストまで、あるいは証明課題の生成まで検証作業をサポートしている。このセクションでは仕様のチェックとその品質向上のために使われるインタープリタ、デバッガや証明課題生成機能をどう使うかについて記述する。

### 3.8.1 インタープリタを使用した式の評価

インタープリタを使って、式や文の評価とデバッグを行うことができる。これらはかなり複雑で、Toolbox に読み込まれている仕様で定義されている関数や操作の実行、変数の使用を含んでいる。デバッガを使ってブレイクポイントの設定、評価作業のステップ実行、変数の値を見るなどができる。


図 11 に示す実行ウインドウが、(ウインドウ) ツールバーの  (実行) ボタンを押すことによって開く。実行ツールバーが、まだ開いていなければ同時に開く。

実行ウインドウの上部には2つの画面、それぞれ応答 と入力画面がある: 入力画面からはインタープリタに直接コマンドを入れることが出来、その結果が応答画面に表示される。VDM-SL の式を評価するためには、入力画面からコマンドラインで直接タイプする。

入力画面で下記のようにタイプしてみよう:

```
print { a | a in set {1,...,10} & a mod 2 = 0 }
```

Return キーを押す。答えとして、偶数の集合が表示される。今検証した式は、集合内包と呼ばれる構成子である。後に ([7]) で説明する。

 (初期化) ボタンを押すことで、インタープリタの初期化が行われ、sort-init.rtf ファイルから読み込まれている VDM-SL の構成要素を参照することが可能になる。初期化中に、定数は評価され、状態定義の値が初期化される。初期化後は、

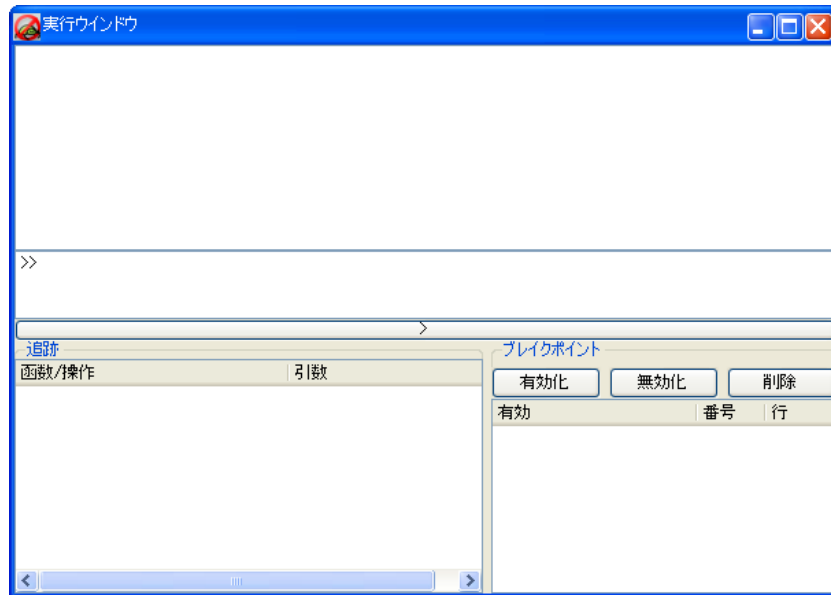


図 11: 実行ウィンドウ

関数、操作、インスタンス変数、値、型など仕様内で定義されているものなら何でも参照できる。

現在どの関数が呼び出し可能か調べるには、入力画面のプロンプトで `functions` とタイプして Return を押す。このコマンドは利用できる関数の一覧を表示する。リストからは、例えば不変条件関数を呼び出せることがわかるだろう。これは `PosReal` 型のような不変条件を含む型がそれらに付随しているからである。ダイアログは図 12 で示す。

これで `print` コマンドが使えるようになった。例えば、2つの順番に並んだ整数列を引数に取る `Merge` 関数を呼び出す（例：`print Merge([1, 3], [2, 4])` とタイプする）と、インタプリタは、`Merge` 関数を評価した結果 `[1, 2, 3, 4]`；（図 12 参照）を表示する。

VDM-SL を構成するものの中には、いくつか実行不可能なものも含まれており、ゆえにインタプリタを使用しての評価ができないものがある。例えば、陰関数 `ImplSort` をコールしようとする、インタプリタは、評価中に実行不可能な構成要素に出くわしたとエラーを返す。図 12 参照のこと。



図 12: 式の評価

### 3.8.2 ブレイクポイントの設定

ブレイクポイントは、インタプリタで関数などの実行をするときに実行を一時中断する。

`break MergeSort` とタイプすることで、関数 `MergeSort` にブレイクポイントを設定することができる。

(現在のモジュールにブレイクポイントを設定するときは、単純に関数または操作名を参照すればよい。他モジュールにブレイクポイントを設定する場合は、関数または操作名を定義済みのモジュール名で修飾しなくてはならない。) コマンドを実行すると、実行ウインドウの右下のブレイクポイント画面に、ブレイクポイントに割り当てられた番号 (この場合、最初にブレイクポイントを設定しているので 1)、ブレイクポイントが有効であることを示す印 ☒ とともに、ブレイクポイント

の設定場所が表示される。今度は、以前使ったコマンドを使う代わりに、debug コマンドを使って評価作業を行うことが出来る。この2つのコマンドの違いは、print がブレイクポイントを無視する一方、debug コマンドはブレイクポイントでインタープリタを停止させることができることである。

以下のようにタイプして MergeSort 関数を呼び出してみよう。

```
debug MergeSort([ 3, 56, 34-12, 0 ])
```


インタープリタは MergeSort 関数に入ったところのブレイクポイントでストップする。同時に、MergeSort 関数を含む仕様のソースファイルがソースウインドウに表示され、現在評価中のポイント（ここではブレイクポイントの場所。例では MergeSort 関数の最初のところ）にカーソルが当たる。加えて、追跡画面（実行ウインドウの左下）にはコールスタックが表示される。

ここで、MergeSort print コマンドを使う（入力画面で print 1 とタイプする）か、実行ウインドウの左下部分にある追跡 画面の関数名の近くに表示されている‘...’部分をマウスの左ボタンをクリックするかすると、関数のパラメータの値を詳細に見ることが出来る。値の表示されているパラメータ上でマウスの左ボタンをクリックすると、また‘...’表示に戻る



ソースファイルの箇所を直接選択することでもブレイクポイントを設定することができる。加えて、ブレイクポイントは関数・操作の最初である必要はなく、その内部であればどこでも設定できる。

ソースファイルが RTF 形式のファイルでない場合は、ソースウインドウのファイル中の設定したい位置で右ボタンをクリックし、メニューからブレイクポイント設定を選択することで、ブレイクポイントを設定できる。ソースファイルに RTF フォーマットを使っている場合は、Word でファイルの適切な位置にカーソルをあて Control-Alt-Space キー でブレイクポイントが設定できる。

デバッグ中はいつでもブレイクポイントが設定できる。それでは、上記の方法で、ソースファイルを使って、Merge 関数内にブレイクポイントを設定してみよう。

インタープリタに戻って (実行) ツールバーの  (実行再開) ボタンを押してみよう。これで次のブレイクポイントまで実行される。実際には MergeSort の再帰的な呼び出しによって、インタープリタは同じブレイクポイントで何度か止まるが、そのたびに Merge 関数の中で実行がとまるまで実行再開ボタンを何度も押す。

実行が進むにつれ、さまざまな関数が呼ばれ追跡画面にログが増えていく。そしてこのコールスタックを実行のステップをトレースするのに使うことができる。

 (一段上の関数の呼び出し位置を表示) ボタンを何度か押して関数のトレースの前後関係がどう変化しているか確認できる。 (一段下の関数の呼び出し位置を表示) ボタンは追跡を元に戻すときに使う。


 (1ステップ実行) ボタンを押すことでステップ実行をすることもできる。ボタンを何度か押して、ソースウインドウ中のカーソルが、評価箇所の変化をマークする様子を確認してほしい。これで、関数のパラメータだけでなく、スコープ中にあるローカル変数を含むすべての変数にアクセス可能になった。さらに、例えば print コマンドを使うことなどで、値の中身を見ることができるようになった。

図 13 にデバッグの例を示す。

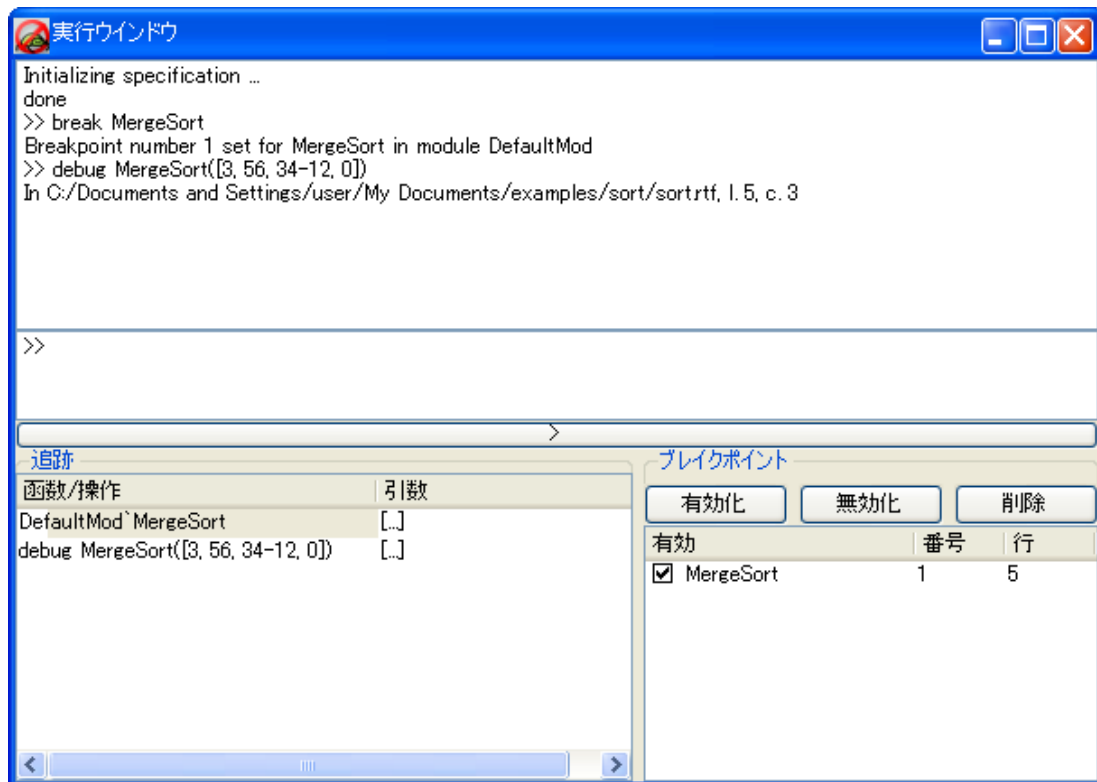


図 13: 仕様のデバッグ

ブレークポイントは、デバッグ実行中であってもなくても、いつでも削除することができる。実行ウインドウの入力画面で


`delete 1` (例 ブレイクポイント 1 番を削除)

とタイプしてみてほしい (これで 1 番のブレイクポイントが削除される)。実行ウィンドウのブレイクポイント画面で、ブレイクポイントを選択して画面上部にある削除ボタンを押しても結果は同じである。

ブレイクポイント 画面上部の他 2 つのボタンは、ブレイクポイントの有効・無効を設定する。MergeSort 関数の中にブレイクポイントを再度設定し、それをブレイクポイント画面で選択して無効化ボタンを押してみよう。記号 ☒ がブレイクポイントが無効になっていることを示す ☐ に変わっていることに注目。有効化 ボタンを押すことでブレイクポイントは再度有効になり、記号は ☒ に戻るはずである。

### 3.8.3 動的型チェック

型チェック機能が仕様のエラーを報告しなくても、型情報の簡単な静的解析 (関数や操作のシグネチャに宣言されている型のみに基づいた解析) では、通常、すべての型エラーを見つけることは不可能なので、型エラーが存在する可能性がある。例えば、ある関数の引数が 1 個の整数 (int 型) と定義され、実数 (real 型) に対して評価する式が適用されていたとしても、int 型が real 型のサブタイプであるため、関数の実行時に、整数値を持つ実数型の引数で関数が呼ばれた場合、その適用は正しい。

仕様レベルでこの手の型エラーを発見するために、評価実行中に、インタープリタが動的型チェックを実行するよう設定することができる。このオプションはプロジェクトオプション ウィンドウの **実行** タブで設定できるが、プロジェクトオプションツールバーに表示されている  (プロジェクトオプション) ボタンを押すことで当該画面が表示される。下記図 14 にこれを示す。

「動的に型チェックする」オプションを有効にすると、インタープリタは評価実行中、実際の型をチェックする。

今回の例では、プロジェクトオプションウィンドウの **実行** タブで動的型チェックを有効に設定し、適用、OK ボタンを押すと、式の評価

```
debug MergeSort([ 3.1415, -56, 34-12, 0 ])
```

のところでインタープリタは動的型チェックエラーを報告する。(ブレイクポイ

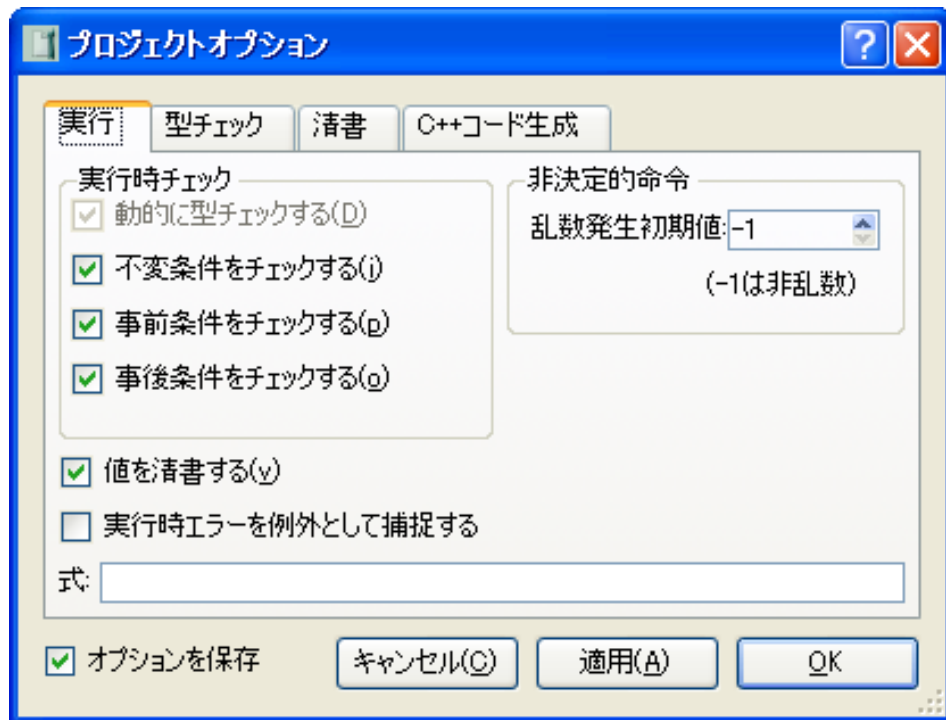


図 14: インタープリタオプションの設定

ントを有効に設定したままになっていた場合は、一度ステップ実行をする必要がある) これは、関数 Merge のシグネチャが、引数に int 型をとると宣言してあるのに、実際の引数は 3.1415 という real 型 (int 型でない) の数字を含んでいるからである。この動的型エラーは、仕様のエラーの可能性を明らかにしている。MergeSort 関数はパラメータとして real 型の列も許容すべきなのに、integer の列のみを許容する Merge 関数を呼び出しているためだ。

似たような方法で、インタープリタが動的に型の不変条件や関数の事前条件、事後条件、予想される操作 (例 true と評価される) などのチェックをするように設定することが可能である。これらのオプションも図 14. にあるようなプロジェクトオプション ウィンドウの実行タブで設定することができる。

例としてプロジェクトオプション ウィンドウの実行 タブに戻って、事前条件をチェックするのオプションチェックを有効にしてみよう。これで以前の箇所を再度評価してみると、今度はリストから数字 3.1415 が省略されてしまっている。今度はインタープリタが図 15 に示すように事前条件違反を報告している。これは Merge 関数の事前条件が、入力値の全てが負でないことを要求しているからだ。



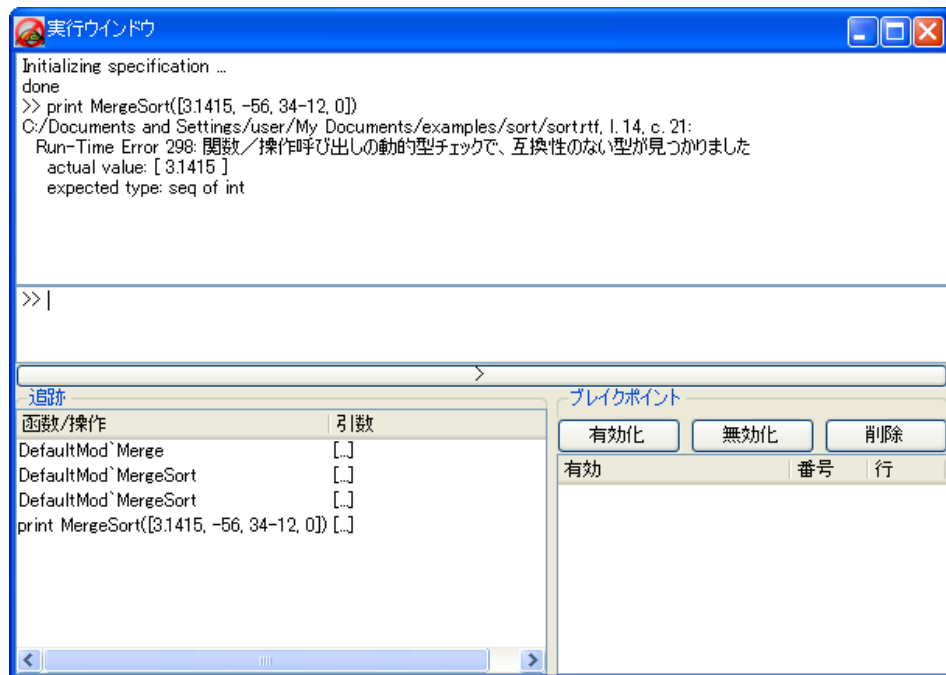


図 15: 動的型チェックエラー

### 3.8.4 証明課題のチェック


上記の型チェック、不変条件、事前条件、事後条件などの動的チェックは基本的にテストの一形式である。いくつかの特別な入力値に対してランタイムエラーが起こるかどうかチェックする。証明課題生成機能はランタイムエラーの可能性を調査するもっと一般的な方法を提供するが、これは数学よりもプログラミングに通じている人からすれば、あまり直感的に理解できないかもしれない。

証明課題生成機能は仕様の潜在的にランタイムエラーが起こりうる箇所を探して分析し、ランタイムエラーが起こりえない条件を表す一連の証明課題を生成する。これらの証明課題は、動的チェックよりもより一般的に使われるが、それは適切な変数<sup>8</sup>がとりうるすべての値の定量化を含む VDM-SL の記述として表現されるからである。これは、もし証明課題が True と実行された場合、変数の値に何が入っていようがそれと関連するランタイムエラーは存在しないことになる（動的チェックの場合、もちろん選ばれた変数の特定の値についてランタイムエラーが起こらないことが確認できるにすぎない）。もちろん、証明課題が false を示す

<sup>8</sup> 場合によっては、すべてのコンテキストが明確に示されず、変数のスコープが仕様の精査によって定義される。



こともあり、その場合仕様の相当する箇所に潜在的な問題があることを指摘している。

実際に証明課題生成機能がどう動くかを見るには、DefaultMod モジュールを選択し(アクション) ツールバーの  (証明課題生成) ボタンを押すとそのモジュールの整合性テスターが起動する。証明課題ウインドウが開いて表示され、証明課題が生成される。図 16 にこれを示す。

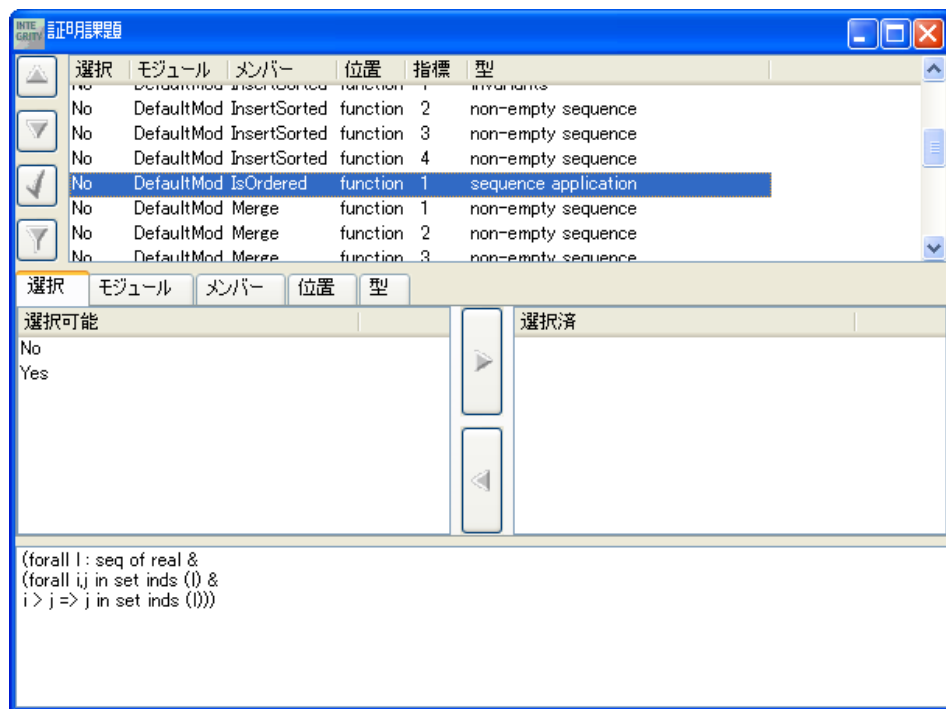




図 16: 証明課題ウインドウ

証明課題ウインドウの画面上部に証明課題のリストがそれらの状態(選択済 欄)、仕様の場所(モジュール、メンバー、位置 欄)と型(型欄)の情報と一緒に表示されている。指標 欄の数字は単純に同じ箇所の違う証明課題を区別するものである。この例でも見られるように、小さい仕様であっても、たくさんの証明課題を生成することがある—実際、30 ある証明課題のすべてがチェックされている—そのため、大きな仕様ではこれらをフィルターできるため、有効である。証明課題ウインドウ の中ほどの2つの画面で、さまざまなフィルタリング方法が利用できる最後に、特定の証明課題がウインドウのトップ画面で選択されると、それに相当する VDM-SL の記述がウインドウの下画面に表示され、同時にソースウインドウ のカーソルが仕様の関連する箇所を示す。それぞれの証明課題は True

かそうでないかを決定しようとするため、詳細に調べられる。

ウインドウの左側の画面にあるメンバー属性を選択し、同じ画面で ExplSort, IsOrdered, Permutations, RestSeq の 4 関数を選択して  (選択した項目を追加) ボタンを押し、フィルタにこれらを追加する。ウインドウ上部の左側にある  (項目の絞込み) ボタンを押すことで、証明課題の一覧がこれらの関数に関連するものだけにフィルタリングされる。

isOrdered 関数と関連する最初の証明課題 (例 インデックス番号 1 番) を選択してみよう。これは下記のような形式になっている：


```
(forall l : seq of real &
  (forall i,j in set inds (l) &
    i > j =>
      i in set inds (l)))
```

そしてソースウインドウのカーソルの位置から、 $l(i)$  で表されるシーケンスアプリケーションの状態と関連していることがわかり、

```
forall i,j in set inds l & i > j => l(i) >= l(j)
```

は正しい定義である (例：  $i$  の値は常に列  $l$  のインデックス)。

このような典型的な例では、実際証明課題が正しいことを見て取るのは簡単だが

- 2 番目の記述は、直接的には  $i$  も  $j$  も  $l$  のインデックスであることを示しており、 $i$  が  $j$  より大きいかどうかに関わらず (3 行目の記述) 不適切であるとされている。それゆえ、この課題はチェックしなくてはならない。証明課題ウインドウのトップ画面左の  (項目の選択/非選択) ボタンを押すことによってチェックができる。

列 アプリケーションに関連する他の 3 つの証明課題を見てみよう。これらが正しいことを確認するのも簡単だ。ひとつは isOrdered が列アプリケーション  $l(j)$  よりも  $l(i)$  と関連しているというところで上記で論じた例外に酷似しているため、同様のことが適用できる。Permutations に関連するものは、すぐに正しいことがわかるがこれは 2 番目のものが  $(i \text{ in set inds } (l))$  で要求される結果を出していることからである。3 番目の RestSeq の場合は、 $j$  は  $l$  のインデックスに属

していないといけないので、インデックスのひとつ  $j$  が  $i$  と一緒に書かれているのを消さなくてはならないと示している。これら 3 つの証明課題は同様の方法で選択し、マーク・チェックすることができる。

これらのようなケースでは、証明課題は機械的チェッカーを使って実際には自動的に確認をする。しかしより複雑なケースにおいては、いつも自動確認が使えるわけではなく、実際の推論が自動化されるようなものであったとしても、推理の過程で人が舵取りをする必要がある。

そのように複雑な課題の例が `ExplSort` function 関数にある。これは基本的には、暗黙の `let` 文の述部を満たす少なくともひとつの値  $r$  がなくてはならない（さもないと仕様は意味を成さない

）が、これに関する記述が仕様にないのである。この証明課題が正しいことがわかるのはそう簡単な事ではない。なぜなら `Permutations` の定義を使っているユーザ定義の関数 `Permutations`, `isOrdered`, `RestSeq` が出てくるからである。加えて、`Permutations` は帰納的に定義されている。しかしながら、提供された関数 `Permutations`、`isOrdered` が正しく定義されているため、証明課題が正しいことも簡単に見て取れる。-明らかにどんな数の順番が与えられてもソート可能なので、われわれがすべきことは `Permutations` 関数で返された順列が入力値としてとりうるすべての順列をカバーしているかということと、`isOrdered` 関数が順番に引数の数字を正しく定義しているかである。

ソースウインドウの `isOrdered` 関数の定義を見てみよう。これが相対的に正しいことは簡単にわかるはずだ。その定義の記述は直接的には、列における 2 つのポジションが与えられているが、後の番号の方は先の方の番号のより小さくなりえないとなっている—これははっきりとこの要素が（昇）順になっているはずであることを意味している。

`Permutations` 関数を見てみよう。case 式の最初の部分は扱いやすい—空の数列の順列がひとつしかない可能性があり、要素がひとつしかない数列は、はっきり言えば数列そのものである。others 部分については、まず `RestSeq` 関数を見る必要がある。これは単に与えられた数列から与えられた箇所の要素を取り除けばよいだけだ。`Permutations` 関数の others 部分では、順列の最初の要素として元の数列から任意の要素を選択することと元の数列の残りの要素のすべてのとりうる順列を結合することによって順列を構成している。それゆえ、これですべてのとりうる順列が与えられるため証明課題は満たされる。

残り 2 つの証明課題を見てみると、`RestSeq` 関数に関係するものだが、ひとつは

事前条件の型の一種、これは事前条件が満たされてさえいれば、関数の明確な結果が事後条件を満たすことを要求しているがーが有効であることがわかる。この関数は数列からひとつの要素を取り除くため、数列の Length が一つ減って数列の要素は変わらないか（その数列で1回より多く要素の削除が行われた場合）少なくなる。しかしながら、不変条件の型の証明課題は、すべての自然数は0と異なるとしており、これはもちろん正しくない。

ソースウィンドウの RestSeq の仕様を見てみると、証明課題は関数の事前条件から生成されていることがわかる：

```
i in set inds l
```

実際、列のインデックスは正の自然数の集合（例. set of nat1nat1 型）であるため、別名をつけるが i が nat1 型でない時点で事前条件は自動的に正しくないことになる。これはこの関数のシグニチャで nat を nat1 に修正するべきだということの意味している。こうすれば、新しい証明課題は

```
(forall l : seq of real, i : nat1 &  
i <> 0)
```

となり、これはもちろん正しい。

最後になるが、最後の証明課題、これは関数 ExplSort の不変条件課題であるが、見てみよう。以下のような形式となっている

```
(forall l : seq of PosReal &  
(forall xx_10 in set elems (let r in set Permutations(l) be st  
isOrdered(r) in r ) &  
DefaultMod'inv_PosReal(xx_10)))
```

見たところとても複雑だが、実際は正の real 値をもってこの関数を呼び出すと、すべての結果の数が PosReal の型の不変条件を満たすことが記述されている。i.e. もまた正の real 型の数である。これは明らかに正しいので、この課題はチェック済みマークがつく。

他関数の証明課題は同じようなやり方で扱うことができる。

### 3.9 体系的テスト

検証のサポートという点からすると、Toolbox はテストカバレッジの測定結果を含む VDM-SL の仕様のテスト向けツールを提供する。テストカバレッジの測定結果は与えられたテストスイートが仕様をどのくらいカバーできているか見るための助けとなる。これは記述や表現がテストスイートの実行中評価された特別なテストカバレッジファイルで情報を集めることによってなされる。

テストカバレッジレポートの作成には3つのステップがある。

1. テストカバレッジファイルを準備する。このファイルは仕様の構造についての情報を含んでいる。
2. インタープリタに仕様の構成要素の呼び出しを実行させることで仕様をテストする。このプロセスはテストカバレッジファイルの情報を更新する。
3. テストカバレッジレポートを清書する。清書機能は仕様とテストカバレッジファイルを取り、うまく活字に組まれたテストカバレッジ情報を含む仕様を作り出す。以下についてはセクション 3.10 でまた記述する。

このプロセスは図 17 に記述されている。まず `tcov reset` をテストカバレッジファイルをリセットするために発行するため、与えられた仕様のテスト情報には何も載っていない。それから仕様と異なる物を評価するため `print` コマンドを使う。それから、`tcov write` で先ほどの `tcov reset` を発行してから生成されたテストカバレッジ情報すべてを `vdm.tc` ファイルに保存する。最後に、コマンド `rtinfo` がテストカバレッジファイルの情報を要約したテーブルを表示する。これが仕様のさまざまな関数や操作のリストをひとつに構成し、それぞれテスト中に何回その関数／操作が呼び出されているかと、仕様の1度以上テストされた箇所のパーセンテージの注釈がつく。

VDM-SL Toolbox (`vdmdc`) のコマンドラインバージョンもテストカバレッジ情報の収集をサポートするため同様の機能を有していることに注意してほしい。

`tcov write` コマンドを使って `vdm.tc` ファイルにテストカバレッジ情報を書き込む前に、実際のテストでは自然にもっとたくさんのテストを増やしていくものだ。本当に実際のプロジェクトでは、一般的にこのプロセス全体を自動化するために小さなスクリプトファイルを書くなどして全体的なテスト環境を構築したする。これもまた予測される結果に対する実際の個々のテスト結果と比較できる（通常

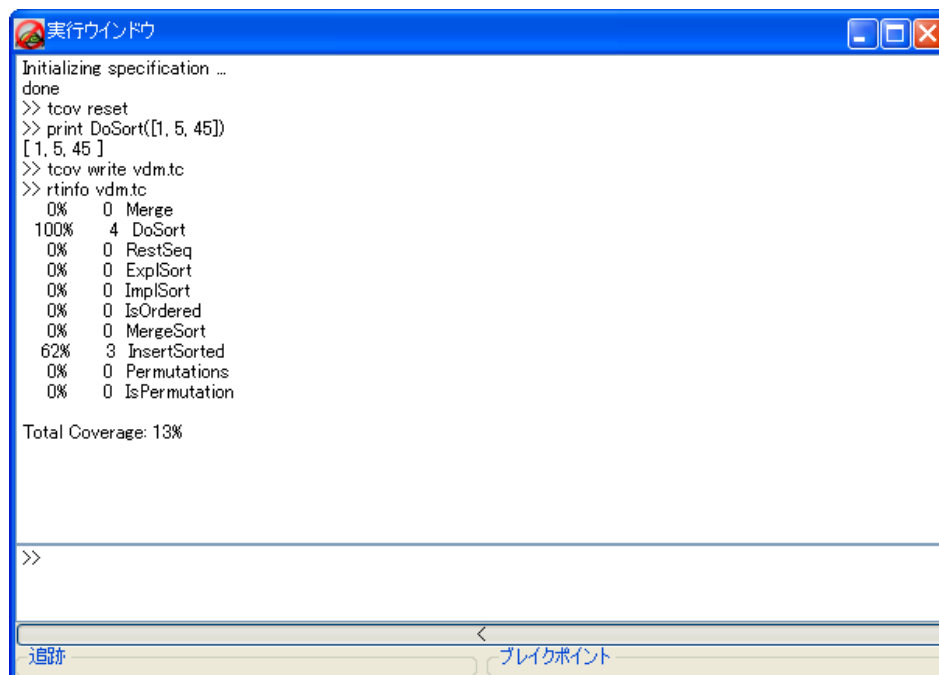


図 17: テストカバレッジ情報の収集

-O オプションが使うのに必要である)。付録 E にこのような Windows と Unix のスクリプトファイルの例が含まれている。

### 3.10 清書機能

清書機能は仕様を入力フォーマットから清書版に変更する。この清書版は大体ドキュメント化の目的で使われる。

清書機能が動いているところを見るには、まずプロジェクトオプション 画面の清書タブをクリックし、インデックスを生成するためにオプションをひとつ有効にし (RTF フォーマットが使われていれば2つのうちどちらを使っても問題はない)、テストカバレッジの色オプションも有効にする。たった今 Toolbox のワーキングディレクトリ に生成した vdm.tc ファイルをコピーしておく必要もある。インタープリタの入力 画面から pwd を入力することで確定できる。

マネージャーのプロジェクトビューで the sort-init.rtf ファイルを選択し、(アクション) ツールバーの **pp** (清書) ボタンを押す。ログウィンドウ にファイル



`sort-init.rtf.rtf` が出来ているのがわかるはずだ。`sort-init.rtf.rtf` ファイルで Word を起動してみる。VDM-SL のキーワードはすべて太字になっていることに注意してほしい。仕様のその他の部分は、Word の `VDM_COV` と `VDM_NCOV` 形式を使って書かれており、それぞれカバーされた部分とカバーされていない部分に関連している。これらの形式の定義は変更することができ、ドキュメントにカラープリンタを使うのであれば `VDM_NCOV` 形式の定義を変更する必要がある（例カバーされていない部分はグレーを使う）

`sort-init.rtf.rtf` ファイルの最後に行ってみよう。`VDM_TC_TABLE` 形式で書かれたテキストがどのようにテストカバレッジを示す統計資料をあらわすテーブルに置き換わっているかに注目。3つの欄が関数/操作名、テストカバレッジファイル内で呼ばれた回数、そのカバレッジのパーセンテージの3つである。テーブルはこのようになる<sup>9</sup>


name	#calls	coverage
DoSort	4	100%
ExplSort	0	0%
InsertSorted	3	62%
IsOrdered	0	0%
IsPermutation	0	0%
Merge	0	0%
MergeSort	0	0%
Permutations	0	0%
RestSeq	0	0%
<b>total</b>		15%

最後に、ファイルの最後について、Word の挿入 プルダウンメニューから Index and Tables ... を選択する。`sort-init.rtf.rtf` の定義の概要の見出しのためのレイアウトを希望するものに決めて Ok ボタンを押すと DM の定義のインデックスが自動的に生成される。

双方向の清書機構を  $\text{\LaTeX}$  に使用する際とはまったく違うが、このマニュアルのリファレンスセクション（セクション 4.9 参照）で説明する。

<sup>9</sup>前のセクションのインタープリタの応答 画面内で直接見られた情報の一部にとってもよく似ていることに注意。

### 3.11 コード生成

VDM-SL から C++ へのコード生成のライセンス を持っていれば、 (C++生成) ボタンを押して自動的に仕様から C++ のコードを生成することが出来る。C++ コード生成についての詳細は、[12] を参照のこと。

### 3.12 動的リンク機能

特別なライセンスを必要とする追加機能がもうひとつあり、動的リンク機能と呼ばれている。この機能は、一部 VDM-SL に特有で、一部 C++ のコードに現存するシステムを結合させたものを解釈することを可能にする。Sort の例ではこの機能の例証はないが、[6] には示されている。動的リンク機能についての詳細な情報は [6] を参照のこと。

### 3.13 VDMTools API

**VDMTools** のすべての機能は、Corba API を経由して外部プログラムにエクスポートすることができる。API の使い方についての詳細は [13] を参照のこと。

### 3.14 VDMTools の終了

Toolbox を終了させたいときは、メインウィンドウのプロジェクト メニューから **終了** を選ぶ。プロジェクトを保存せずに終了しようとする、ダイアログが現れてプロジェクトを保存するかどうか聞いてくる。

これで Toolbox の「ガイドツアー」は終了である。ツールの提供する機能がよりよく理解出来ていることと思う。自身の VDM-SL のモデルで Toolbox を使い始められるようになっているはずだ。このマニュアルの残りの部分では、特定の部分の特徴について詳細なリファレンスガイドとなっている。



## 4 VDMTools リファレンスマニュアル

このセクションは Toolbox 内のツールそれぞれをカバーする数々のサブセクションで構成されている。それぞれのツールは GUI、Emacs、コマンドラインの各インターフェースで使うことができる。以下それぞれについて記述する。

### 4.1 GUI 全般

Toolbox の GUI はウインドウズのプログラムから選択するか、Unix 環境で `vdmgde` を入力することで起動する。図 18 に示す GUI のメインウインドウが開く。

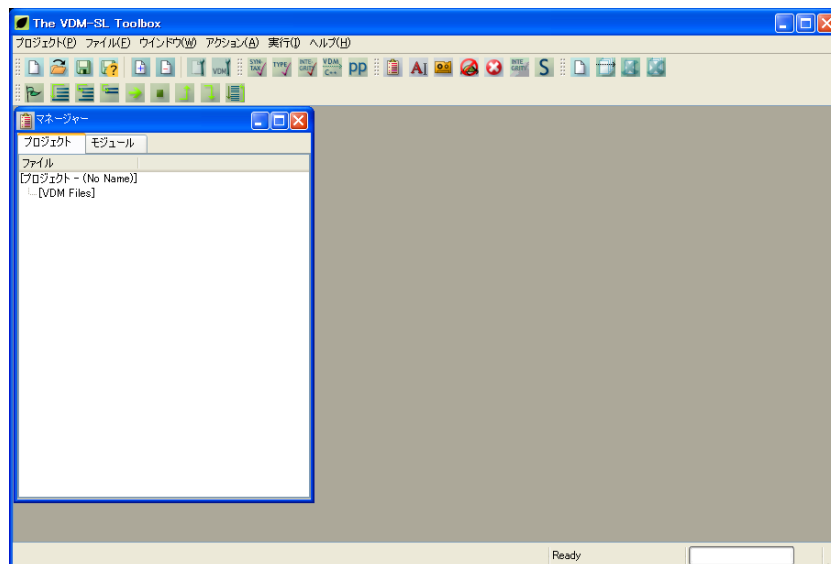


図 18: GUI スタートアップ画面

ウインドウのトップは6つのプルダウンメニューで構成されており、その下にはメニューと同様のアクションを提供するボタン<sup>10</sup>から成る6つのツールバー<sup>11</sup>がある。ウインドウの下部分は現在のプロジェクトの状態についての情報や Toolbox 内ツールのインターフェースを提供するさまざまなサブウインドウを表示するの

---


<sup>10</sup> ツールボックスの終了はプロジェクトメニューからしかできない

<sup>11</sup> Toolbox を起動したときはツールバーは3つしか開いておらず、他の3つは上部にアイコン化されて表示されている。





に使われる。以下のサブセクションでは、メニュー、ツールバー、サブウィンドウそれぞれについて機能別に記述する。

#### 4.1.1 プロジェクト・ハンドリング

プロジェクトは VDM-SL 形式の仕様のファイルを集めたもので構成される。プロジェクトは保存されディスクから読み込むことが出来るが、これは Toolbox に個々のファイルを毎度毎度使いたいときに保存する設定をする必要がないことを意味する：ただ適切なプロジェクトファイルを開けばよいだけなのだ。プロジェクトは GUI でのみ利用できる。

マネージャーは、ウィンドウ メニューから適切なものを選択するかまたはウィンドウ ツールバーの  ボタンを押下することで起動/終了するが、現在のプロジェクトの状態を表示するだけでなく、操作しようとするプロジェクトファイルのサブセットに対し適用しようとするさまざまな Toolbox の操作を選ぶ場所でもある。プロジェクトビュー とモジュールビューの 2 つで構成される。

プロジェクトビュー はプロジェクトのファイル構成（構文チェック済みのファイルのみ）と各々のファイルで宣言されているモジュール の内容をツリー構造で表示する。図 19 にそれを示す。

VDM-SL のファイルが構文チェックを無事パスすると、それらのファイルで定義されているモジュール の名前がモジュールビュー にリスト表示される。このビューはプロジェクトでのモジュール それぞれの状態を記号 **S**, **T**, **C**, **P** で各々適切な欄にこれらは、当該モジュール が正常に構文チェック済み (syntax checked)、型チェック済み (type checked)、C++コードを生成済み (translated to C++)、清書済み (pretty printed) なのを示すが、似たような印で各記号に赤線が入ったもの (, , , ) は個々の処理が失敗したことを示す。空欄だった場合は、まだ個々の処理が実行されていないことを示す。プロジェクト中のファイルのひとつがこのシステム上で修正されると、構文 欄に現在の Toolbox でファイルのバージョンとファイルシステムでのバージョンに不整合が起こっていることを示す記号が表示される。そのファイルは他の処理に進む前に再度構文チェックを行うべきである。

プロジェクトを開く/保存する、プロジェクトへのファイルの追加と削除、新規プロジェクトの作成などを含むプロジェクト操作のためのさまざまな処理がプロジェクトメニューとそれに相当するプロジェクトツールバーから利用できる。こ

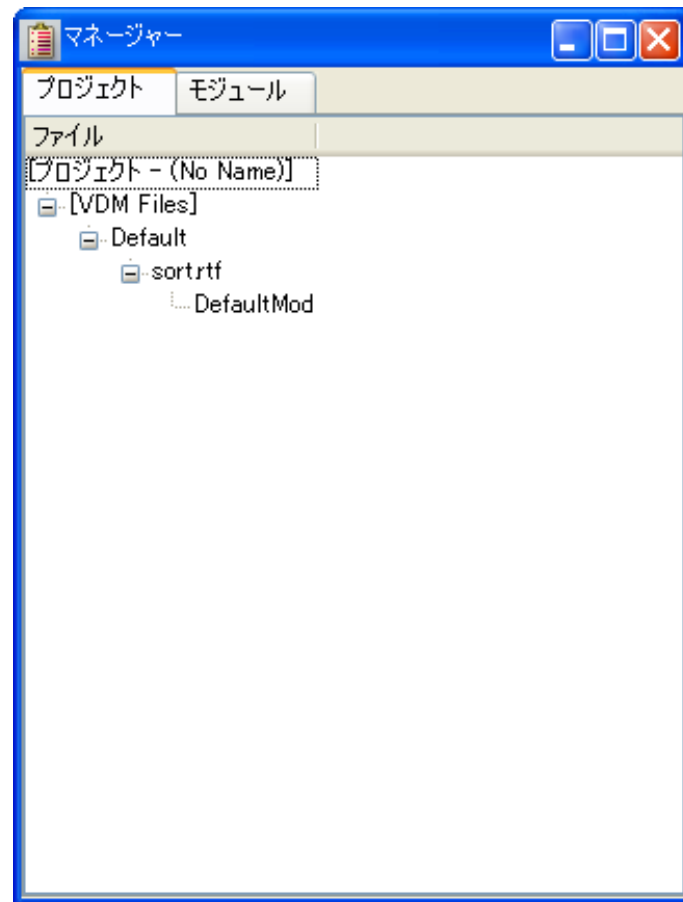


図 19: プロジェクトビュー

れを図 20 に示す。

同じメニュー/ツールバーを使って、Toolbox の環境に関するオプション設定、Toolbox 中のさまざまなツールのオプション設定、Toolbox の終了（メニューからのみ可能）をすることができる。以下で利用できる処理について詳しく述べる。

**新規プロジェクト (📁):** 新しいプロジェクト上で作業を始めたいときに選ぶ項目

**プロジェクトを開く (📂):** すでに存在するプロジェクトを開きたいときに選ぶ項目。ファイルブラウザが表示され希望するプロジェクトファイルを選択することができる。これがロードされると Toolbox はプロジェクト中のすべてのファイルに自動的に構文チェックをかける。

**保存 (💾):** プロジェクトの設定を変えてそれを保存したいときに使用する項目

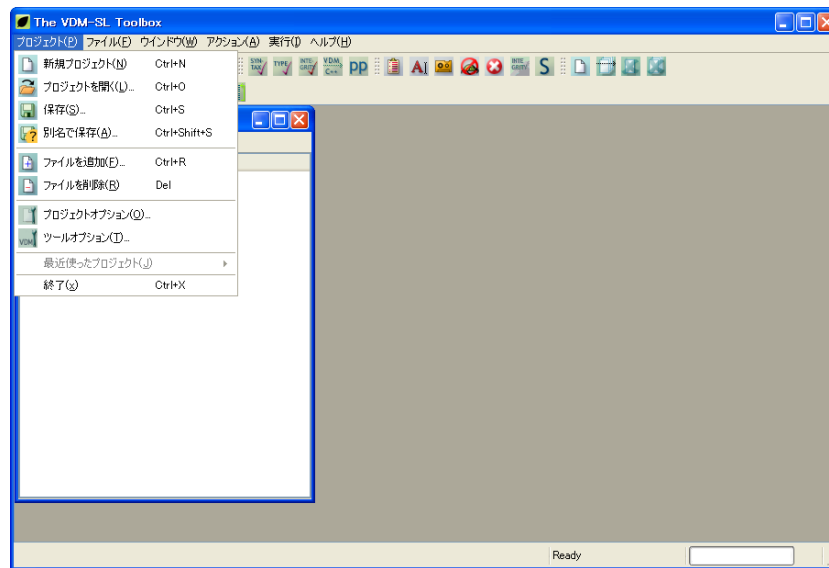


図 20: プロジェクトメニューとプロジェクトツールバー

別名で保存 (📁?): 現在の構成を別な名前で保存したい場合に使う。ファイルブラウザが表示され新しいプロジェクトを保存する場所を好きに設定できる。また名前も好きなものに出来る。

ファイルを追加 (📁+): ツールボックス上の現在のプロジェクトにファイルを追加するときに使う。図 3 に示したようなウィンドウが表示され、追加したいファイルを選択することができる。

ファイルを削除 (📁-): プロジェクトからファイルを削除したいときに使う。削除の確認ダイアログが表示される。

プロジェクトオプション (🔧): .

- 実行 (セクション 4.5 参照);
- 型チェック (セクション 4.4 参照);
- 清書 (セクション 4.7 参照);
- C++コード生成 (セクション 4.8 参照)

ツールオプション (🔧): . ツールオプション ウィンドウを開く。Toolbox のインターフェースオプションの設定ができる。これらのオプションについては付録 C を参照のこと。

最近使ったプロジェクト: PCで最近使ったプロジェクトのリストを開く。

終了: Toolboxを終了する。プロジェクトを保存していない場合は、Toolboxが保存するかどうかを聞いてくる。ツールバーからは使えないことに注意。

#### 4.1.2 仕様の操作

Toolboxは仕様に適用させうる広範囲な機能を提供している: 構文チェック、型チェック、証明課題の生成、C++のコード生成、清書など。これらはアクションメニューまたはそれに相当するアクションツールバーから起動することが出来る。図 21 にこれを示す。

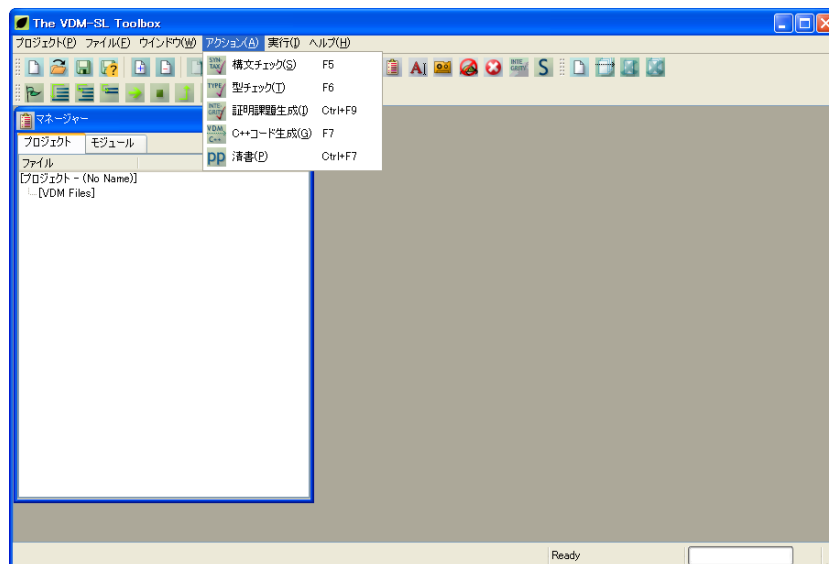




図 21: アクションメニューとツールバー

それぞれのアクションはマネージャで現在選択中のファイル・モジュール 各々に適用される。アクションはある程度まで相互依存しているため、そのうちのいくらかは選択されたモジュールが求められ適用する機能を可能にする状態の時にのみ実行される。例えば、型チェック機能と清書の機能はモジュールが構文チェック機能をパスしてから適用できる。


さまざまなアクションが下記に示すセクションで詳細に記述される。

構文チェック (): セクション 4.3 参照


型チェック (): セクション 4.4 参照



証明課題生成 (): セクション 4.6 参照


C++コード生成 (): セクション 4.8 参照

清書 (): セクション 4.7 参照

### 4.1.3 ログウインドウ、エラーリストウインドウ、ソースウインドウ

ログウインドウ は Toolbox からのメッセージを表示するが、これには上で記述した動きを適用したときの成功・失敗の報告メッセージを含む。すでに開いていない限り、新しいメッセージを表示するときに自動的に開く。代わりにウインドウ ツールバーで  ボタンを押すかウインドウ メニューで相当する項目を選ぶことで手動でウインドウを開いたり閉じたりすることも出来る。

エラー一覧 はアクション実行中に Toolbox によって発見されたエラーを報告する。図 22 に示すとおり 2 つの画面から構成される。上のほうの画面はエラーやワーニングの起こった箇所（ファイル名、行数、欄番号）のリストを示し、一方下の画面では選択中のエラーの詳細な説明が表示される。構文チェック中や型チェック中に生じるさまざまなエラーの形式は、セクション 4.3.2 と 4.4.2 にそれぞれ記述されている。最初は、自動的にリストの先頭のエラーが選択されている。エラーリストの左にある  or  ボタンを押すことでエラーリスト内の次・前へ移動できる。代わりにエラー一覧の上の画面にあるエラー通知を示す印を直接選択しても任意のエラーへ動かすことができる。

エラー一覧 はすでに開いていない限り新しいエラーが見つかるると自動的に開く。代わりに ウインドウ ツールバーで  ボタンを押すかウインドウ メニューで相当する項目を選ぶことで手動でウインドウを開いたり閉じたりすることも出来る。

ソースウインドウ もまたすでに開いていない限り新しいエラーが見つかるると自動的に開く。現在選択中のエラーが発見された元の仕様の一部を表示し、実際のエラーの位置はウインドウズのカーソルでマークされる。図 22 に記述されたエラー一覧 に相当するソースウインドウ を図 23 に示す。

多くのソースファイルがソースウインドウに表示されているが、内容が表示されているのはそのうち 1 つだけである。違うソースファイルを見なければ、ソース

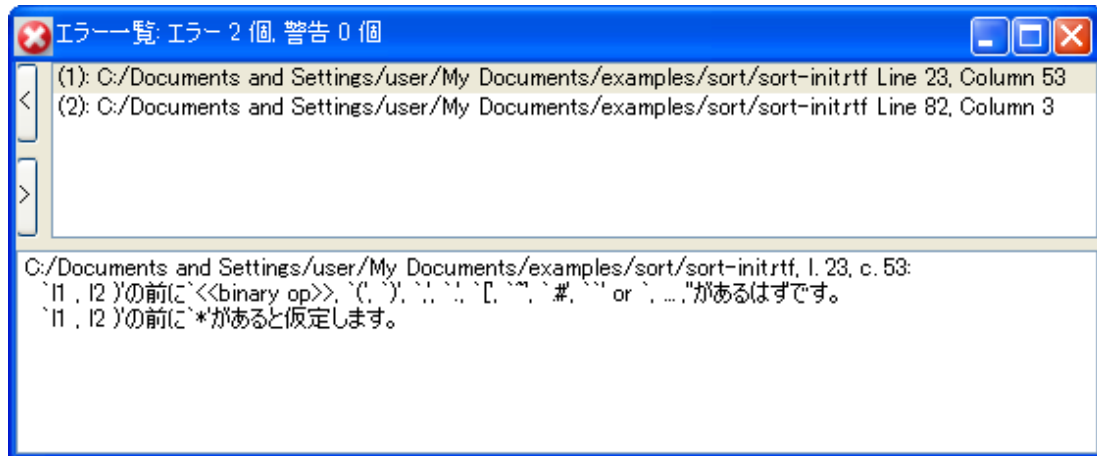








図 22: エラー一覧

図 23: ソースウインドウ

ウインドウの上部にあるファイルに相当するタブを選択することで表示が変わる。新しいソースファイルを足すには、マネージャーで手動でファイル名（またはファイルに含まれるモジュールのひとつ）をダブルクリックする。ソースファイルはファイル ツールバーの （ソースウインドウから選択されたファイルを閉じる）ボタンまたは （ソースウインドウのすべてのファイルを閉じる）ボタンを押すと画面上から消える。 ボタンは現在表示中のファイルを閉じ、 ボタンはすべてのファイルを閉じる。

ソースウインドウ はウインドウ ツールバーの  ボタンを押すかウインドウ メニューから同様の項目を選択することで開いたり閉じたり出来る。


#### 4.1.4 ファイルの編集

Toolbox を終了させずにエラーを修正するために、好みのエディタ (付録 C 参照) を作業中のプロジェクトのファイルから直接起動することができる：単純に編集したいファイルを Manager で選択し、プロジェクトツールバーの外部エディタボタン () を押す。この方法で外部エディタを起動するときに複数のファイルが

選択した場合、ひとつの外部エディタで選択したファイルそれぞれを表示する。

Toolbox はエディタで保存した変更を自動的に登録する。しかし編集されたファイルのバージョンは自動的に更新されることはないので、ソースファイルを編集したら他のツールを使う前に必ず構文チェック機能を再度走らせれば、Toolbox にも変更が反映される。


#### 4.1.5 インタープリタを使う

インタープリタを使って式と文のデバッグ、評価ができる。実行ウインドウはインタープリタへのインターフェースを提供するが、(ウインドウ) ツールバーの  (実行) ボタンを押すと開く。また実行 メニューおよびツールバーはインタープリタでできるあらゆる処理を提供する。詳細はセクション 4.5 で記述する。

#### 4.1.6 オンラインヘルプ

Toolbox のオンラインヘルプと一般的なインターフェースはヘルプ ツールバーやヘルプ メニューからアクセスできる。最近では以下に示す限られたものだけが利用可能である。

ツールについて (): Toolbox のバージョン番号を表示する

Qt について (): Qt (Toolbox のインターフェースが利用している、C++のマルチプラットフォーム GUI ツールキット) へのリファレンス情報を表示する

## 4.2 コマンドラインインターフェース全般

コマンドラインインターフェースはプロンプトから以下のように入力 することで起動する。<sup>12</sup>:

```
vdmde [-o scriptfile] [-q] [specfiles]
```

---

<sup>12</sup>実行可能な `vdmde` コマンドがサーチパスに必ずあるか、フルパス指定をしなくてはならない



コマンドラインから `vdmde` がファイル名 (VDM-SL の仕様を含んでいなければならない) で起動されると、ツールはコマンドモードに入り指定されたファイルの構文チェックを始める。`-o scriptfile` をつけると、指定された仕様を読み込んだ後に、スクリプトファイルに記述されたスクリプトを実行する。`-q` オプションをつけると、すべての処理が終了した時点で、`vdmde` は終了する。

ユーザーは Toolbox が提供するたくさんのコマンドをプロンプトからタイプすることで、仕様の操作、実行、デバッグができる。下記で与えられたコマンドは `vdmde` によってサポートされている。丸カッコ内の省略形はコマンドの短縮形である。

多くのコマンドが、仕様の初期化前には使用できない。(init コマンドについては、セクション 4.5 を参照) これらのコマンドには (\*) マークがついている。

種々の構成要素の名前を表示するのに、多くのコマンドが使われる。**modules**, **functions**, **operations**, **states**, **types**, **values** である。**info** または **help** を使って Toolbox のコマンドのヘルプが得られる。頻繁に使われるコマンドのシーケンスはスクリプトファイルに集められ、**script** コマンドを使って実行することができる。一般的な OS のシステムコールは **system** コマンドで発行することができる。**dir** コマンドはツールボックスの検索パスにディレクトリを足すときに使う。**pwd** コマンドは現在のワーキングディレクトリを表示する。最後に **quit** または **cquit** コマンドでコマンドライン入力の Toolbox を終了することが出来る。下記でこれらコマンドについて記述する。

## modules

定義済みのモジュール名とその状態情報を表示

## \*functions

現在のモジュールで定義される関数名を表示する。事前条件、事後条件、関数の不変条件は仕様がそれを含む場合自動的に作成される

## \*operations

現在のモジュールで定義されたすべての操作名を表示する

## \*states

定義済みのグローバルステート名を表示する

## \*types

現在のモジュールで定義済みの型名を表示する

**\*values**

現在のモジュールで定義される値の名前を表示する

**help [command]**

このセクションで使われているような、すべての利用可能なコマンドを説明するオンラインヘルプと同じスタイル。引数なしだと利用可能なコマンドすべてのリストを表示する。そうでない場合は引数で与えられたコマンドの説明となる。

**info [command]**

help と同じ。

**script file**

file からスクリプトを読み込み、実行する。スクリプトは VDM-SL コマンドの羅列である。これらはコマンドラインインターフェースであればこのセクションや他のセクションで記述されたどのコマンドも使用できる。スクリプトの実行が終わると、コントロールは Toolbox に戻る

**system (sys) command**

シェルコマンドを実行する

**dir [path ...]**

アクティブなディレクトリのリストにディレクトリを追加する。これらのディレクトリは仕様のファイルの場所を探すとき自動的にサーチされる。このコマンドを引数なしで実行するとアクティブなディレクトリのリストが画面に表示される。ディレクトリは表示された順にサーチされる。

**pwd**

現在のワーキングディレクトリを表示する。例えば、プロジェクトファイルがあれば作業中のプロジェクトファイルのある場所である。すべての場合、vdm.tc ファイルのある場所であり、コード生成で生成されたファイルが書き込みをするところとなる。

**encode [encoding]**

仕様が書かれている文書の文字コードを設定する。

**cquit**

確認の質問なしでデバッガを終了する。バッチジョブでデバッガを使うときに利用するとよい

**quit (q)**

cquit と同じ

**4.2.1 ファイルの初期化**

コマンドラインインターフェースでは、「ファイルの初期化」をすることができる。これらのコマンドは Toolbox をコマンドラインで起動すると自動的に実行される。

初期化ファイルは `.vdmde` ファイルで指定され、引数としてファイルを指定するためには Toolbox が起動するディレクトリか仕様のファイルのあるディレクトリと同じディレクトリになくてはならない。


### 4.3 構文チェック機能

構文チェック機能は作成した仕様が言語定義であたえられている構文に沿うものであるかどうかチェックする。このシステムのほかのツールは仕様が構文的に正しい前提で動くため、仕様は Toolbox のほかのツールを適用する前に構文チェックを行い、構文エラーのない状態にしておく必要がある。元ファイルを修正した場合は、他のツールが修正に気づく前に構文チェックを再度しておかなければならないことに注意すること。

構文チェック機能は GUI、コマンドライン、Emacs のいずれのインターフェースを使っても使用することができる。

構文チェック機能の狙いは仕様の構文エラーを出来るだけ多く同時にレポートすることである。このため、構文チェック機能は最新のリカバリー機構を使用しているが、これにより構文エラーを見過ごしてしまう前に捕捉し復旧することや、すぐ次に生じる仕様の構文エラーを報告することができる。仕様のある記号を無視したり足りない記号を想定したりすることでこれを可能にする。エラーメッセージは、仕様のエラーが起こった箇所で何が期待されていたかということやチェッカーが実行し続けるためには何が無視されるべきで何が想定されるのかということについての情報もあたえてくれる。最初は、何が想定/無視されるのかということに集中することによってエラーメッセージを理解するのが最も簡単である。なぜならこの構文チェック機能による推測は実際のエラーに近いものであることが多いからだ。

#### 4.3.1 GUI

構文チェック機能を GUI で起動するには、チェックしたいファイルまたはモジュール（複数選択可能）<sup>13</sup> をマネージャー のプロジェクトビュー またはモジュールビュー で選択して、(アクション) ツールバーの  (構文チェック) ボタンを押す。ログウィンドウ が（開いていなければ）自動的に開き選択したファイルやクラスそれぞれのチェックの進行状況についての情報を順番に表示する。構文エラーが発見されると、エラー一覧 とソースウィンドウ が自動的に起動する。

<sup>13</sup> モジュールビュー でモジュール を選択すると、構文チェック機能は実際には選択したモジュールの含まれる ファイル一式に適用される。Toolbox はどのファイルが編集されたかということしか知らない。これはもし特定のファイルが複数のモジュール 定義を含んでいて、そのうちのいくつかだけを選択していた場合には、暗黙のうちに同じファイルの他モジュール が処理に含まれる。

#### 4.3.2 構文エラーのフォーマット

仕様の構文エラーが見つかったら、構文チェック機能はエラー一覧に以下のような情報を表示する。

1. 構文エラーの箇所にどんな記号が（足りないことが）想定される (expected) か
2. 構文エラーから復旧するにはどうすればよいか。記号を挿入するか、記号を無視するか、違う記号に置き換えるかなど。仕様のファイル自体はこの処理によって何も変わらない（構文チェック機能内でのみ実行される変化であり、これがさまざまな構文エラーを捕捉することを可能にしている）

記号は3つの形式の混合で表示される:

- シングルクォート内に表示 e.g. ‘functions’.
- メタシンボルの表示 e.g. <end of file>, 「ファイルの最後」の意味
- 似たようなトークン群をシングルトークンとして表示 e.g. <<type>>, 構文上のユニット type (定義は [7]) 予想される記号のリストを短くするためにこれがなされる

#### 4.3.3 コマンドラインインターフェース

コマンドラインから構文チェック機能を起動するコマンドの構文は:

```
vdmde -p [-w] [-R testcoverage] specfile(s) ...
```

-p オプションをけると、vdmde コマンドはそれぞれ1つ以上のモジュール、またはフラットな仕様の一部を含むたくさんのファイルをチェックする。エラーは stderr で報告される。

その他の追加オプションは、

-w VDM-SL の RTF ファイルの一部を ASCII に書き出す。ASCII のファイル名は RTF のファイル名に拡張子 `.txt` がつく。例) `sort.rtf` は `sort.rtf.txt` となる

このオプションはテスト環境で仕様の解析時間を減らすためによく使われる。RTF ファイルの文書の部分が大きいと、ファイル全体を解析しなければならないためとても遅くなる。例えば、図はファイルの文書部分をととても大きくしてしまう傾向がある。

-R VDM-SL 仕様のテスト中違う構成要素がどれだけ実行されたかを記録するのに使われるテストカバレッジファイル生成する。(ファイル名は引数 `testcoverage` で指定できる) 現在のバージョンでは、このテストカバレッジファイルは `vdm.tc` という名前ではなくてはならない (清書機能が動くため)。例についてはセクション 4.9 を参照。

-W code 仕様文書の文字コードを `code` に設定する

#### 4.3.4 Emacs インターフェース

Emacs インターフェースでは、すべてのコマンドをプロンプトから入力する。構文チェックは `read` コマンドで行われ、構文エラーを詳しく見るには `first`, `last`, `next`, `previous` コマンドが使われる。

##### `read (r) file(s)`

`file(s)` から仕様の構文チェックを行う。 `file(s)` はモジュール定義または関数、値、操作、型、場合によっては状態の定義を含まなくてはならない。

それぞれのファイルの内容は全体として扱われる。これはもし構文エラーが起こっても、そのファイルの VDM-SL での構成要素は何も含まれないということを意味する。またこれはもしそのファイルが複数のモジュールを含んでいた場合も含む (モジュールには何も含まれない)。ファイルが構文チェックをパスし、構文チェック済みのファイルですでに定義されたモジュールが再定義されたならば、ワーニングが発生する。

##### `first (f)`

構文チェック機能、型チェック機能、コード生成、清書機能からなど最初に記録されたエラーまたはワーニングメッセージを表示する。

**last**

構文チェック機能、型チェック機能、コード生成、清書機能などから最後に記録されたエラーまたはワーニングメッセージを表示する。

**next (n)**

構文チェック機能、型チェック機能、コード生成、清書機能などからソースファイルウィンドウの次の位置に記録されたエラーまたはワーニングメッセージを表示する

**previous (pr)**

構文チェック機能、型チェック機能、コード生成、清書機能などからソースファイルウィンドウの前の位置に記録されたエラーまたはワーニングメッセージを表示する

## 4.4 型チェック機能

型チェック機能は記述されているものが仕様のその位置に想定される型であるかどうかを評価する。しかし、型の正しさはそれが想定するようにいつもははっきりしたものであるわけではない。例えば、ある関数が引数に `int` 型の値をとっているが記述としては `real` 型が適用されているとすると、`int` 型は `real` 型のサブタイプなので、提供されたその関数は実行時たまたま実際には `int` 型の引数をとって呼ばれ、アプリケーションが正しいのかもしれない。また `real` 型は `int` の一部ではないため、アプリケーションは正しくないのかもしれない。このようなアプリケーションはおそらくよくまとめられているとは言えても、**明確**によくまとめられているとは言えないのである。

実際、型チェック機能はこれら2つの異なるレベルどちらでも型チェックを実行することができる。端的に言えばこれら2つの違いは「おそらく適格な仕様」が型としては正しいがそうであることがきちんと保障されていない一方で「明確に適格な仕様」は型として正しいことが保障されているのことにある。そのため、全節で論じた関数のアプリケーションは「おそらく適格な (possible well-formedness (“pos”)) 型の」チェックは通っても「明確に適格な (definite well-formedness (“def”)) 型の」チェックは通らないことになる：“def” 型チェックはランタイムエラーの潜在的な原因となるものを特定するからである。

“def” 型チェックは潜在的にランタイムエラーが起こりうる箇所をすべて特定する。これらは事前条件のある関数（事前条件はアプリケーションのその関数が呼ばれる前に満たされていなければならない）を持つアプリケーションやVDMへ直接ビルトされる一部の演算子（例. 除算演算子は2番目の引数が0だとランタイムエラーを起こす）を含むアプリケーションを含み、同様に定義にサブタイプを使ったことから来る潜在的な不整合をも含む。


一般的に、“def” 型のチェックは“pos” 型のチェックよりエラーメッセージが多くなる。そのため仕様をチェックするときはまず型がおそらく正しくない箇所を扱うため“pos” タイプのチェックを行い、それからランタイムエラーの原因となる潜在的な箇所を特定する目的で“def” 型チェックを行うことをお勧めする。多くのケースで、これら例えば、表記が“if ... then ... else ...” 式の内にあるせいで、ランタイムエラーの条件が発生するのを阻害する状態になっている箇所などを考慮外にすることができるだろう。その他のケースとしては、“def” 型のチェックは仕様の修正による防備を導入したいために状態を特定することができる。

型チェック機能はGUIからでもToolboxのコマンドラインからでも、Emacsイ




ンターフェースからでもアクセス可能である。

#### 4.4.1 GUI

GUIで型チェック機能を起動するには、マネージャー のプロジェクトビュー またはモジュールビュー でチェックしたいファイルまたはクラスを選択し、(アクション) ツールバーの  (型チェック) ボタンを押す。ログウィンドウ が自動的に開き、選択されたファイルそれぞれについてチェックの進行状況についての情報を順番に表示する。型エラー が発見されるとエラー一覧とソースウィンドウ が自動的に起動される。

#### オプション設定

“pos” 型チェック または “def” 型チェック のどちらの適格性チェックをするかはプロジェクトオプション ウィンドウの型チェックタブで(プロジェクト) ツールバー上の  (プロジェクトオプション) ボタンを押すとできる。これを図 24 に示す。“pos” 型の “def” 型どちらもいつでも利用可能である。デフォルトは “pos” 型の適格性チェックが有効になっている。

以下2つのオプションも提供されている。

拡張型チェック: 有効になった場合、“‘ conc ’ の結果が空列になるかも知れません” などの追加ワーニングが型チェックの際にたくさん出る。

デフォルト: 無効。

ワーニング/エラーメッセージ分離: 有効になった場合、エラー一覧に表示する際に型チェック機能の出すエラーメッセージとワーニングを分ける。エラーメッセージはワーニングの前に表示される。

デフォルト: 有効。

#### 4.4.2 エラーおよびワーニングのフォーマット

型チェック機能が生成するワーニングのすべては、潜在的な問題は何かという説明のテキスト記述である。未定義の識別子なども同様に単純なテキスト形式であ

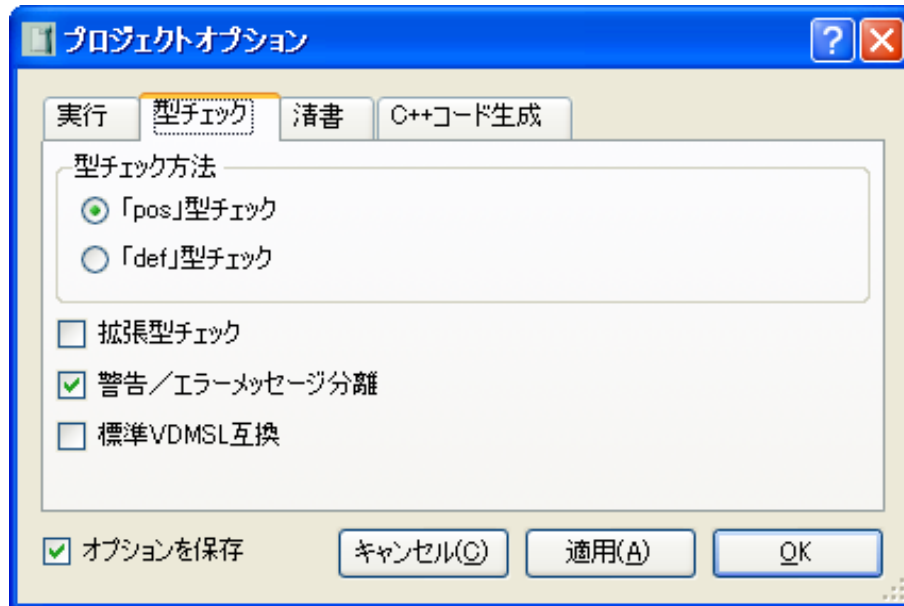


図 24: 型チェック機能のオプション設定

る。しかし、型エラーの大半は3行で構成され、1行目では問題についてのテキストの説明、2行目は型チェック機能が推測する実際の型（`act:`というキーワードで特定される）、3行目は型チェック機能が期待する型（`exp:`というキーワードで特定される）である。これらの型の記述についての構文はほぼ通常のVDM-SLの型の構文と同様である。（下記は例外）

- `seq of A` は `seq1 of A | []` と表示される。[] は空シーケンスの型。
- `map A to B` は `map A to B | {|->}` と表示される。{|->} は空マップの型。
- `set of A` は `set of A | {}` と表示される。{} は空セットの型。
- `[A]` は `A | nil` と表示される。
- # はどんな型の代わりにもなる。型チェック機能はエラーの状況では他に何も思い当たらない場合は、この型を推測に当てはめる

型エラーの例はセクション 3.7 に記述されている。

## “def” 型チェックで考えられるエラー

式がいつも正しい型であると保証することができない箇所はどこでも、‘def’ チェックを実行することでエラーレポートを作り出すことができる。‘def’ タイプの適格性チェックをすることで出てきたワーニングやエラーのうちいくつかを理解するために、‘DEFINITELY’（明確に）という言葉のエラーメッセージに暗に挿入してみるとよい。例えば、メッセージ

```
Error : Pattern in Let-Be-expression cannot match
```

が‘def’ チェックで帰ってきたとすると、これを以下のように読んでみる。

```
Error : Pattern in Let-Be-expression cannot DEFINITELY match
```

すなわち、パターンにマッチしない値をとりうる。型チェック機能がエラーを報告するときは、その位置に推測される型と予想される型を表示する。これは何がいけないのかを見つけるには有効である。

### 4.4.3 コマンドラインインターフェース

```
vdmde -t [-df] specfile(s) ...
```

-t オプションを使うと vdmde コマンドは specfile(s) の型チェックを行う。まず、仕様が解析される。それから構文エラーが見つからなければ、仕様の型チェック（デフォルトは‘pos’ タイプの適格性チェック）がなされたことになる。型のエラーは stderr に報告される。

その他の型チェック機能の追加オプションは下記のとおり：

- d ‘def’ タイプの適格性チェックを実行する。‘pos’ と ‘def’ タイプの適格性チェックの違いについては、言語マニュアルに記載されている。( [7] )。端的に言う と ‘def’ タイプの適格性チェックは型に関する立証の義務を返す
- f 拡張された型チェックを実行する。‘pos’ ‘def’ どちらの適格性チェックであっても “Result of ‘conc’ can be an empty sequence” のようないくらか多くのワーニングとエラーメッセージが出る。

-W code 仕様文書の文字コードを code に設定する

#### 4.4.4 Emacs インターフェース

Emacs インターフェースではすべてのコマンドをプロンプトから入力する。型チェックは **typecheck** コマンドで実行され、ワーニングと型エラーを詳細に見るには構文エラーと同様 **first**, **last**, **next**, **previous** コマンドを使う。エラーの箇所は specification ウィンドウで示される。拡張された型チェックのオプションは **set** コマンドで有効にでき、**unset** コマンドで無効にすることができる。詳しくは下記に利用可能なコマンドを記述する。

##### **typecheck (tc)** [module] option

与えられたモジュール (モジュールが提供されていない場合は現在のモジュールがチェックされる) の静的型チェックを行う。(カレントディレクトリの全モジュールを型チェックするには、“\*” 記号を用いる) option は pos または def でありこれは仕様が pos タイプまたは def タイプのどちらで適格性をチェックするかを表す。

型エラーが起こって報告されると、specification ウィンドウに情報が表示される。

##### **first (f)**

構文チェック機能、型チェック機能、コード生成、清書機能からなど最初に記録されたエラーまたはワーニングメッセージを表示する。

##### **last**

構文チェック機能、型チェック機能、コード生成、清書機能などから最後に記録されたエラーまたはワーニングメッセージを表示する。

##### **next (n)**

構文チェック機能、型チェック機能、コード生成、清書機能などからソースファイルウィンドウの次の位置に記録されたエラーまたはワーニングメッセージを表示する

##### **previous (pr)**

構文チェック機能、型チェック機能、コード生成、清書機能などからソースファイルウィンドウの前の位置に記録されたエラーまたはワーニングメッセージを表示する

### **set full**

Toolbox の内部オプションをすべて有効にする。パラメータなしで実行されると現在の設定を表示する。

`full` は拡張された型チェックを有効にする。このオプションは `pos` タイプ・`def` タイプどちらの適格性チェックにも有効である。デフォルトは無効。

### **unset full**


拡張された型チェックを無効にする

## 4.5 インタープリタとデバッガ

インタープリタとデバッガは VDM-SL の仕様の実行を可能にする。かならずしもインタープリタを使う前にすべてのモジュールの型チェックをする必要はない(が仕様の型が正しくないとよりランタイムエラーが起こりやすくなる)。インタープリタ・デバッガは GUI、コマンドライン、Emacs いずれのインターフェースを使っても利用可能である。

VDM-SL の構成要素で実行できないものは、陰関数や陰操作、型束縛、モデリングに VDM-SL の 3 値論理を課す制約に従った式である。

### 4.5.1 GUI


実行ウィンドウ は(ウィンドウ) ツールバーの  (実行) ボタンを押すことで開いたり閉じたり出来る。ウィンドウ メニューから同様の機能を選択することでも起動が可能である。


このツールには画面が 2 つある。それぞれ応答画面と入力 画面である：入力 画面からはインタープリタ に直接コマンドを入力することが出来、その結果を応答画面で見ることが出来る。VDM-SL の式を評価するには、入力 画面からコマンドラインで直接入力する。


ツールの下 2 つの画面は追跡 画面とブレイクポイント 画面である。追跡画面は実際の引数を伴った関数のコールスタックを表示する。引数は一般的にデフォルトでは省略され、ただ‘...’と表示されるだけである。‘...’表示の上でマウスの左ボタンをクリックすると詳細を見ることが出来る。再度左ボタンをクリックするとまた‘...’表示に戻る。


ブレイクポイント 画面は現在のブレイクポイントの位置と状態を表示するが、それぞれ関数名の左側に有効化 (☒) と表示) または無効化 (☐) と表示) となる。画面上部のボタンはそれぞれ現在リスト中で選択しているブレイクポイントの有効化、無効化、削除 にあたる。


実行メニューとツールバーはインタープリタでできるさまざまな操作を提供している。


処理系を初期化 (


ステップ実行 (


関数内をステップ実行 (


1 ステップ実行 (

実行再開 (

関数の実行を終了 (

一段上の関数の呼び出し位置を表示 (

一段下の関数の呼び出し位置を表示 (

実行中断 (

入力画面で利用できるコマンド

上記に記述された操作に加え、入力画面でこれらをタイプすることでインタプリタへコマンドを直接入力することができる。これらは下記に示される。しかし、これらのコマンドのうち多くがインタプリタの初期化 (Init) ボタンを押下することで可能) 後でなければ実行できない。これらのコマンドには (\*) マークをつけてある。

**print** または **debug** コマンドを使って式の評価をすることができる。2つのコマンドの違いは、**debug** コマンドを使うとブレイクポイントで停止するのに対し、**print** コマンドでは停止しないことにある。

ブレイクポイントは **break** コマンドを使用するかソースウインドウにて希望する箇所を右クリックし、メニューからブレイクポイント設定を選択すると設定できる<sup>14</sup>。

ブレイクポイントに来ると、ステップ実行 (F7), 1ステップ実行 (F8), 一段上の関数の呼び出し位置を表示 (F9), 実行再開 (F5), 関数の実行を終了 (Ctrl+F5) などの操作が可能になる。ブレイクポイントは **delete** コマンドで削除できる。

“現在の” モジュール (実行中のもの) は **curmod** コマンドを使って詳細に見ることができる。**push** コマンドを使うとモジュールスタックの先頭に新たなモジュールを push することができる。同様に **pop** コマンドを使うとモジュールスタックの先頭にあるモジュールを pop することができる。現在のモジュールスタックは **stack** command. で詳細に見ることができる。

これらのコマンドに加えて、詳細は下記で説明されるが、セクション 4.2 には入力画面で利用できるたくさんのコマンドが載っている。

上矢印キーと下矢印キーは、前に実行したコマンドの履歴をスクロールして見るのに使える。この履歴リストで Enter キーを押すとそのコマンドを実行する。履歴をスクロールする前に文字入力があった場合は、履歴リストのうち入力した文字列で始まるコマンドのみを表示する。

新しいコマンドを入力せずに Enter キーを押すと直前のコマンドを実行する。

#### \***break (b) [name]**

与えられた **name** で指定した関数・操作の箇所にブレイクポイントを設定

<sup>14</sup>RTF フォーマットを使っている場合、ダブルクリックは使えない。その代わり、Microsoft Word 内にブレイクポイントを設定したい場合、(ブレイクポイントを) 設定したい箇所で Ctrl-Alt-スペースを押すと設定できる。



する。

このコマンドが実行されると、新しいブレイクポイントに番号が割り当てられ、応答 画面に表示される。新しいブレイクポイントの名前と番号がブレイクポイント 画面のブレイクポイントの一覧に足される。

引数なしで実行されると、現在設定されているブレイクポイントの一覧を表示する。

#### **\*break (b) name number [number]**

与えられたファイル名の、数字で与えられた行にブレイクポイントを設定する。2 番目の引数（数）が与えられた場合、ブレイクポイントを設定する箇所として解釈される。

元のファイルが RTF 形式でなかった場合、ソースウインドウでマウスの左ボタンをダブルクリックすることでもブレイクポイントが設定できる。RTF フォーマットを使っている場合、Word でファイルを開きカーソルを適切な箇所に当てて、Ctrl-Alt-スペースを押すと設定できる<sup>15</sup>。

#### **\*condition (cond) number [, expr]**

ブレイクポイントへのブレイク条件の設定/削除を行う。ブレイクポイントにおいて expr の値を評価し、真となる場合にブレイクする。番号のみの指定で実行すると、条件が削除される。ブレイクポイントの一覧で、条件が表示される。

#### **curmod**

仕様がモジュールで構成されている場合、現在のモジュール名を出力する。

#### **debug (d) expr**

VDM-SL の式 expr の値を評価し、表示する。有効なブレイクポイントすべてで実行がとまるが、このとき現在実行中の箇所がソースウインドウに、追跡 画面にコールスタックが表示される。ランタイムエラーが起こると、エラーの起こった箇所で実行は止まり、エラーの発生箇所がソースウインドウに、コールスタックが追跡 画面に表示される。

インタープリタで式を評価する際に、直前の評価結果を記号\$\$を使って参照することができる。詳細は、下記 print コマンドの記載を参照。

---

<sup>15</sup> ツールボックスのバージョン v8.3.2 以降に割り当てられた VDM テンプレートのバージョン VDM.dot で動く

debug コマンド実行中に**実行中断** ボタンが押されると、ボタンが押されたときに評価中だった式や文で評価は中断される。停止後、式や文のスコープ中にある変数はすべてアクセス可能である。

**\*delete number, ...**

引数 number(s) で指定したブレイクポイントを削除する。ブレイクポイントはブレイクポイント 画面からも消える。

**\*disable number, ...**

number で指定したブレイクポイントを無効にする。

**\*enable number, ...**

number で指定したブレイクポイントを有効にする。

**init (i)**

Initialises インタープリタにある仕様からのすべての定義を初期化する。これは状態 とすべての値の初期化も含む。値が多重定義されていた場合は、初期化の間に報告される。初期化コマンドは、同じセッションにあるツールボックスに読み込まれているすべてのファイルを初期化する。そのため、読み込んであるファイルを個別に初期化する必要はない。

**print (p) expr, ...**

すべてのブレイクポイントを無効にして、VDM-SL の式 **expr** の値を評価し、表示する。ランタイムエラーが起こった場合、実行は止まりエラーの箇所がソースウインドウに表示される。

通常のVDM-SL の値に加えて、**print** コマンドは **FUNCTION\_VAL** と **OPERATION\_VAL** も返すことができる。これは、評価の結果が関数や操作になる場合（例：括弧で囲まれた引数が見つからない形で 関数が評価された場合）に起きる。

インタープリタで式を評価する際に、直近の評価の結果を参照するために、記号**\$\$** を使うことができる。この記号は式として扱うことができ、下記の例に示すように、VDM-SL の式の中で使用することができる。

```
vdm> p 10
10
vdm> p $$+$$, 2*$$
20
```

40

vdm>

print コマンド実行中に**実行中断** ボタンを押すとコマンドの評価は中断される。その後はどの変数にもアクセスできない。

**\*push** モジュール名

指定した モジュール名 は、モジュールスタックにプッシュされ、init 後にアクティブなモジュールになる。

**\*pop**

現在のモジュール (curmod) がスタック (stack) から pop される。アクティブモジュールがない場合は、警告が発行され何も起こらない。

**\*stack**

Push されたモジュール名を表示する。

**tcov**

テストカバレッジコマンド **tcov** を使うと、テストカバレッジ情報の集積をコントロールすることができる。次に示すさまざまなキーワードとの組み合わせで使われる。

**tcov read filename**

filename で示されるファイルに保存されているテストカバレッジ情報を読む。

テストカバレッジファイルと呼んだ後に構文チェックをかけた場合、構文チェックをかけたファイルのカバレッジ情報はリセットされ、構文チェックをする前にどこかへテストカバレッジ情報を書き出しておかない限り失われることに注意。pretty printing 機能がいつも仕様のファイルから特定されるテストカバレッジファイルを参照していることにも注意が必要である。

**tcov write filename**

filename で指定されたファイルに存在するテストカバレッジ情報を書き込む

**tcov reset**

テストカバレッジ情報をリセットする

## オプション設定

インタプリタにはプロジェクトオプションウィンドウの実行 画面で指定できるたくさんのオプションがある（図 25 参照）。以下に示すとおり：

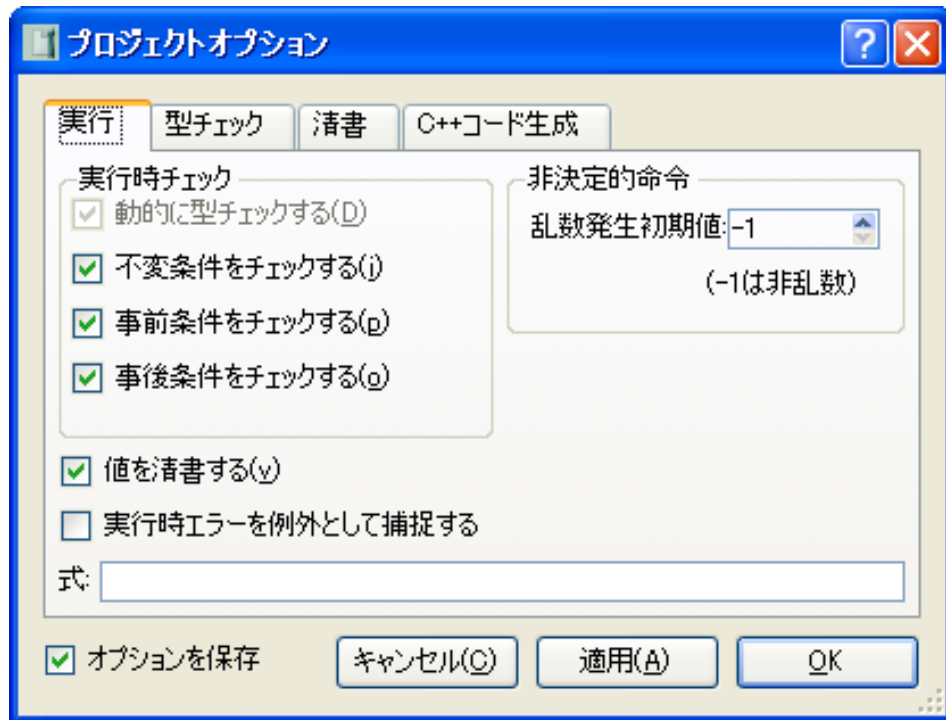


図 25: インタプリタのオプション設定

**動的に型チェックする：**このチェックを有効にすると、式の型が確定するたびに、VDM-SL の仕様に与えられた定義に沿ってチェックされる。

デフォルト：有効

**不変条件をチェックする：**このチェックを有効にすると、式の型に不変条件が存在するときはいつでも、不変条件がチェックされる。不変条件チェックを行うためには、動的な型チェックが有効になっていなければならないため、このチェック を有効にすると、自動的に動的な型チェックは有効になる。不変条件チェックが有効なままで、動的な型チェックを無効にすることはできない。デフォルト：有効

**事前条件をチェックする:** このチェックを有効にすると、評価しようとしているすべての関数・操作の事前条件が、関数・操作の呼び出しの前にチェックされる。

デフォルト: 有効

**事後条件をチェックする:** このチェックを有効にすると、評価しようとしているすべての関数・操作の事後条件が、関数・操作の呼び出しの後にチェックされる。

デフォルト: 有効

**値を清書する:** 値を表示する際に、改行や字下げを挿入して、読みやすくする。

**実行時エラーを例外として補足する:** ランタイムエラーが発生した際に、インタプリタを停止せず、`<RuntimeError>`という値の例外を発生させる。通常インタプリタでは、ランタイムエラーが発生するとその位置でインタプリタが停止する。しかし、一連の回帰テストのテスト等の場合には、テスト中にインタプリタが停止するとテストを効率的に行えない場合がある。そのため、このオプションは、テストプログラムで例外を処理することでテストを効率的に行うためにのみの使用が推奨される。この機能を使用した例外処理は、モデルの本体の仕様に含めるべきではない。

**乱数発生初期値:** 与えられた整数値で乱数ジェネレータを初期化する。これにより、非決定文を構成する文の評価をランダムな順番で行うことができる。その場合、指定する数は0以上でなければならない。負の数を指定した場合、乱数を生成しないので、非決定文はランダムな順序で評価されなくなる。  
デフォルトの値: -1

式: **print** か **debug** コマンドで評価する式を設定する。**print** か **debug** コマンドを引数なしで使用した場合に、この式が評価される。

#### 4.5.2 スタンダードライブラリ

現状、3つのスタンダードライブラリが存在する。VDMユーティリティと Maths と input/output 機能についてである。

## VDMUtil ライブラリ

インタプリタは VDMUtil という標準ライブラリを提供する<sup>16</sup>。このライブラリに関する機能・利用可能な値・及びそれらの具体的な構文は [10] で説明される。このライブラリを使うためには、VDMUtil.vdm ファイルがプロジェクトの一部でなければならない。このファイルは、vdmhome/stdlib ディレクトリに配置されている。

VDMUtil.vdm ファイルには、全てが is not yet specified で定義されたいくつかの関数が含まれている。一般的な VDM-SL 仕様では、is not yet specified で定義された関数を実行できないが、これらの特別な関数の定義は Toolbox に含まれている。したがって、プロジェクトに追加したこれら (VDMUtil.vdm ファイル) のユーティリティが持つ機能は、記述した仕様中で利用可能となる。

## Maths ライブラリ

インタプリタは math スタンダードライブラリを提供している。関数と値が利用可能であり、具体的な構文については [10]. に記述されている。このライブラリを利用するには、ファイル math.vdm がプロジェクトの一部になくてはならない。このファイルは vdmhome/stdlib ディレクトリに存在する。

math.vdm ファイルは is not yet specified として定義されているさまざまな関数を含む。一般的な VDM-SL の仕様ではそのような関数はインタプリタでは実行できないが、これらの特定の関数定義はツールボックス内に存在する。このため、math.vdm ファイルをプロジェクトに include すると、仕様内で maths 関数を利用することができる。

## IO ライブラリ

インタプリタは IO (input/output) のスタンダードライブラリを提供している。関数と値が利用可能であり、具体的な構文については [10]. に記述されている。このライブラリを使用するには、ファイル io.vdm がプロジェクトの一部になくてはならない。ファイルは vpphome/stdlib ディレクトリに存在する。

---

<sup>16</sup>一部にコード生成には対応していない機能がある。

io.vdm ファイルは is not yet specified として定義されているさまざまな関数を含む。一般的な VDM-SL の仕様ではこのような関数はインタプリタにより実行することができないが、これら特定の関数のための定義はツールボックス内に存在している。そのため、io.vdm ファイルをプロジェクトに include すると、これらの IO の関数が仕様上で利用可能になる。

### 4.5.3 コマンドラインインターフェース

インタプリタおよびデバッガは下記のコマンドで起動される：

```
vdmde -i [-DIPQ] [-R testcoverage] [-O res-file] argfile specfile(s)
```

-i オプションをつけると vdmde コマンドは argfile ファイル中、ファイル specfile(s) 中の仕様のコンテキストの VDM-SL の式（またはコンマで区切られた式のかたまり）を評価する。評価の結果は stdout に報告される。一連の式が使われると、記号\$\$を使って直前の式の結果を参照することができる。

ランタイムエラーに出くわすと、インタプリタは終了しエラーメッセージが表示される。エラーメッセージはエラーの原因となった構成要素の場所の情報と、エラーの型についてのメッセージを含む。

インタプリタで使用するその他の追加オプションは以下のとおり：

-D 動的型チェックを有効にする

-I 不変条件チェックを有効にする。

動的型チェックが有効であることが必要なので、-D オプションの有無にかかわらず、自動的に動的型チェックも有効になる。

-P 評価済みのすべての関数および操作の事前条件チェックを有効にする。

-Q 評価済みのすべての関数および操作の事後条件チェックを有効にする

-R インタプリタの実行結果が、testcoverage ファイルを生成するのに仕様のファイルと一緒に argument ファイルも使ったかのようなになる。違いはインタプリタが testcoverage ファイルのランタイムエラー情報を更新し、評



価後それをハードディスクに保存することである。具体的な例はセクション 4.9 を参照のこと

- O res-file argfile の評価結果を res-file に保存する。res-file がすでに存在する場合は上書きされる。このオプションは結果を自動的に予想される結果と比較するテストスクリプトでよく使われる。
- y ランタイムエラーが発生した際に、インタプリタを停止せず、<RuntimeError> という値の例外が発生させる。通常インタプリタでは、ランタイムエラーが発生するとその位置でインタプリタが停止する。しかし、一連の回帰テストのテスト等の場合には、テスト中にインタプリタが停止するとテストを効率的に行えない場合がある。そのため、このオプションは、テストプログラムで例外を処理することでテストを効率的に行うためのみの使用が推奨される。この機能を使用した例外処理は、モデルの本体の仕様に含めるべきではない。
- W code 仕様文書の文字コードを code に設定する

#### 4.5.4 Emacs インターフェース

Emacs インターフェースではすべてのコマンドをプロンプトから入力する。まずインタプリタの初期化を行うことで構文チェック済みの定義を使用することが出来るようになる。初期化は **init** コマンドで行う。多くのコマンドがインタプリタの初期化後でなくては使用できない（下記 **init** コマンドを参照）。これらのコマンドには (\*) マークをつけてある。

**print** または **debug** コマンドで式を評価することが出来る。これら 2 つの違いは **debug** コマンドを使うとブレイクポイントでとまるのに対し、**print** コマンドではとまらないことにある。ブレイクポイントは **break** コマンドで設定できる。ブレイクポイントに来到、**step**, **singlestep**, **stepin**, **cont**, **finish** コマンドが可能になる。ブレイクポイントは **delete** コマンドで削除できる。

**backtrace** コマンドはコールスタックを調べるのに使う。インタプリタのオプションは **set** コマンドを使って設定することが出来、**unset** コマンドを使ってリセットすることも出来る。

**curmod** コマンドを使うと現在の（実行中の）モジュール名を詳細に見ることができる。モジュールスタックの先頭に新しいモジュールを **push** を使って Push する



することもできる。**stack** コマンドを使うと現在のモジュールスタックを詳細に見ることができる。

**\*backtrace (bt)**

関数/操作のコールスタックを表示する。

**\*break (b) [name ]**

**name** で指定された関数または操作にブレイクポイントを設定する。

このコマンドが実行されると、新しいブレイクポイントに番号が割り当てられ、コマンドの実行結果として表示される。

引数なしで **break** が呼び出されると現在のブレイクポイントをすべて表示する。

**\*break (b) name number [number]**

与えられたファイル名の、数字で与えられた行にブレイクポイントを設定する。2 番目の引数（数）が与えられた場合、ブレイクポイントを設定する箇所として解釈される。

**\*cont (c)**

次のブレイクポイントまで続けて実行したいときやまたは式や文の最後まで評価が到達したときに使用する。

**curmod**

仕様がモジュールで構成されている場合、このコマンドは現在のモジュール名を出力する。

**debug (d) expr**

VDM-SL の式 **expr** の値を評価し、表示する。有効なブレイクポイントすべてで実行がとまるが、このとき現在実行中の箇所が表示される。ランタイムエラーが起こると、エラーの起こった箇所で実行は止まり、エラーがソースウインドウに表示される。

最後の評価結果を見るには、記号\$\$ を使うことが可能である。詳細については **print** コマンドの記述を参照のこと。

**\*delete name ...**

引数 **name** で指定した関数や操作に設定されているブレイクポイントを削除する。

**\*disable number**

与えられた **number** で指定したブレイクポイントを無効にする。

**\*enable number**

与えられた **number** で指定したブレイクポイントを有効にする。

**\*finish**

現在評価中の関数または操作を抜けて呼び元に戻る。もともとは **stepin** と対で使われる。

**init (i)**

インタープリタ内の仕様のすべての定義を初期化する。これには状態とすべての値も含まれる。値が多重定義されていた場合は、初期化中に報告される。初期化コマンドは Toolbox の同じセッションに読み込まれているすべてのファイルを初期化する。ゆえに **read** コマンドでファイルが読み込んであれば、個々のファイルを別々に初期化する必要はない。

**\*pop**

現在のクラスがスタックに出される。もしアクティブなクラスがない場合は、警告を発した上で何も起こらない。

**\*popd**

デバッグが入れ子に行われているときに使われる。(ある式がデバッグ中、ほかの評価でそのブレイクポイントが評価された) **popd** コマンドは、最後に **debug** コマンドが起動されたときの環境に戻す効果がある。

**print (p) expr,...**

すべてのブレイクポイントを無効にして、VDM-SL の式 **expr** の値を評価し、表示する。ランタイムエラーが起こった場合、実行は止まりエラーの箇所が表示される。

通常の VDM-SL の値に加えて、**print** コマンドは **FUNCTION\_VAL** と **OPERATION\_VAL** も返すことができる。これは、評価の結果が関数や操作になる場合(例: 括弧で囲まれた引数が見つからない形で 関数が評価された場合)に起きる。

インタープリタで式を評価する際に、直近の評価の結果を参照するために、記号 **\$\$** を使うことができる。この記号は式として扱うことができ、下記の例に示すように、VDM-SL の式の中で使用することができる。

```
vdm> p 10
10
vdm> p $$+$$, 2*$$
20
40
vdm>
```

**remove number**

number で指定したブレイクポイントを削除する。

**set option [argument]**

インタプリタ内部のオプション設定を行う。引数なしで実行された場合は現在の設定を表示する。

オプションは option が使用できる。

**dtc** 動的型チェックを有効にする

**inv** 不変条件の動的チェックを有効にする。dtc も有効になっていないと意味をなさない。

**pre** 事前条件のチェックを有効にする。

**post** 事後条件のチェックを有効にする。

**ppr** 清書のフォーマットを有効にする。すべての値が構造に従ってされて表示される。

**seed integer** 乱数ジェネレータを与えられた数字で初期化する。非決定文を構成するサブ文の評価をランダムな順序で行うためである。integer は 0 以上でなくてはならない。負の数は非決定文のランダムな評価を無効にしまうからである。

すべてのオプションはデフォルトでは false である (ppr を除く)。

**\*singlestep (g)**

次の式を実行する。サブ式・文で止まる。

**\*step (s)**

次の文を実行して止まる。このコマンドは関数や操作内部には入らない。式全体を評価するので関数には有効でない。

**\*stepin (si)**

次の式・文を実行して止まる。関数・操作内部にも入る。

**tcov**

テストカバレッジコマンド **tcov** を使うことによって、テストカバレッジ情報の集合をコントロールすることができる。

**tcov read filename**

**filename** で示されるファイルに保存されているテストカバレッジ情報を読みこむ。

テストカバレッジファイルと呼んだ後に構文チェックをかけた場合、構文チェックをかけたファイルのカバレッジ情報はリセットされ、構文チェックをする前にどこかへテストカバレッジ情報を書き出しておかない限り失われることに注意。pretty printing 機能がいつも仕様のファイルから特定されるテストカバレッジファイルを参照していることにも注意が必要である。

**tcov write filename**

**filename** で指定されるファイルに存在するテストカバレッジ情報を書き込む。

**tcov reset**

テストカバレッジ情報をリセットする。

**unset option, ...**

Toolbox 内のオプション設定を無効にする。可能なオプションについての記述は **set** コマンドの項を参照のこと。

**\*push name**

モジュール **name** がモジュールスタックの先頭に Push され、初期化の後アクティブモジュールとなる。

**\*pop**

現在のモジュール (**curmod**) がスタックから pop される (**stack**)。アクティブなモジュールがなかった場合は、ワーニングが発行され何も起こらない。

**\*stack**


Push されたモジュール名を表示する。

## 4.6 証明課題生成機能

証明課題生成機能は、仕様の潜在的にランタイムエラーが起こる箇所を調べ一連の証明課題を生成する。これはもし true であればランタイムエラーが起こりえないことを保障するに十分なものである。この機能によって 30 の異なるタイプの証明課題がチェックされる。




証明課題は適切な変数<sup>17</sup>のすべての値の数値化を含む VDM-SL の述部として表現されており、これは証明課題が true だと証明された場合には変数にどんな値が含まれていようと、その課題関連のランタイムエラーはありえないことを意味する。もちろん、証明課題が false になることもあり、その場合は仕様の該当する箇所に潜在的な問題があることを指摘していることになる。

証明課題生成機能は GUI からのみ利用できる。

証明課題生成機能を使用するには、マネージャー、のプロジェクトビュー またはモジュールビュー でファイルやモジュール（複数選択可）を選択し（アクション）ツールバーの （証明課題生成）ボタンを押す。（すでに開いていない場合）自動的にログウインドウが開き、選択されたファイルまたはモジュールそれぞれのテストの進行状況を順番に表示し、証明課題ウインドウ が開いて生成された証明課題が表示される。証明課題ウインドウ は図 26 に示す。

証明課題ウインドウ の上部には証明課題の一覧が状態の情報（選択欄）、仕様の箇所（モジュール、メンバー、位置 欄）、型（型欄）と一緒に表示される。指標欄の番号は、単に同じ箇所の違う証明課題を区別するために振られた番号である。リストの先頭をクリックすると証明課題を特別な属性に基づいて整列する。

ウインドウ上部の画面で証明課題を選択すると、下部の画面にそれに相当する VDM-SL の述部が表示される。同時に、ソースウインドウのカーソルが仕様の選択した証明課題が関連する箇所に移動する。証明課題はそれぞれ true かそうでないかを決定しようとするため詳細に調べられなくてはならない。これについての詳細はセクション 3.8.4 に記述がある。

画面左の ,  ボタンで前／次の証明課題に移動できる。 （項目の選択/非選択）ボタンは選択された証明課題の状態をチェック済み／未チェックに変える。

（項目の絞込み）ボタンは証明課題の一覧にフィルターをかけるためウインドウ中

---

<sup>17</sup>いくつかの場合においては、すべてのコンテキストが明確に示されず変数のスコープを仕様の精査によって決定しなくてはならないこともある

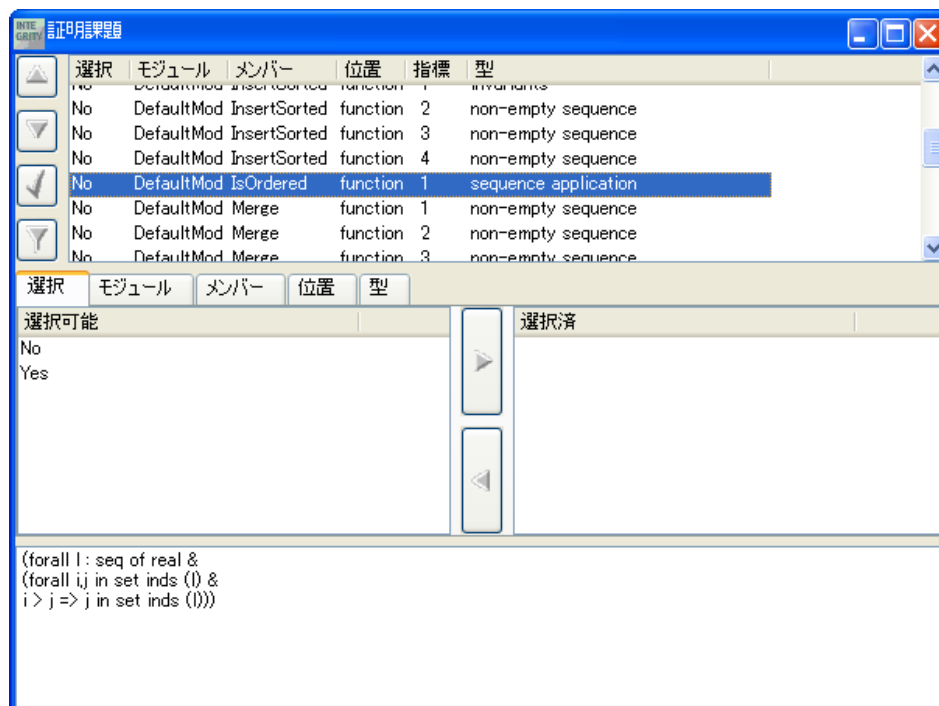





図 26: 証明課題ウインドウ

ほどの2つの画面で連動して使われる。2つのうち左側の画面はそれぞれの属性の可能な値のリストを表示し、右側の画面はフィルターが使う属性おのこの特定の値を表示する。属性の値は画面上で選択して  (選択した項目を追加) または  (選択した項目を削除) ボタンを押すとフィルターに追加・削除できる。  (Filter) ボタンを押すと証明課題の一覧がフィルターされ選択された値にマッチする属性のものしか表示されなくなる。属性が何も選択されていないときは、フィルターがかからないので証明課題はすべて表示される。

## 4.7 清書機能

清書機能は仕様を入力フォーマットから清書版に変更する。この清書版は文書化の目的で使われることが多い。清書機能の出力フォーマットは仕様を入力フォーマットに依存する。入力フォーマットがRTF形式ならば出力フォーマットもRTF形式となる。入力フォーマットが $\text{\LaTeX}$  コマンドとVDM-SL仕様の混合ならば、出力フォーマットは $\text{\LaTeX}$ 形式と成る。2つの清書機能の生み出す異なる出力結果のレイアウトの違いは、Microsoft WordがVDM-SLのASCIIバージョンを使っているのに対し $\text{\LaTeX}$ のほうはほとんどのVDMのテキストや論文で使われている数学的表現のVDM-SLを使っていることである。

清書機能は相互参照付きの索引を構築することができ、テストカバレッジ情報も考慮に入れることができる。まだカバーされていない仕様の一部が色つきになる形式と関数・操作のカバレッジがパーセンテージで記述されているテーブル形式の両方で可能である。

入力ファイルがRTF形式の場合、モジュール名を含むことによって.rtfファイルの任意の場所にVDM\_TC\_TABLE形式で書かれたテストカバレッジのパーセンテージを要約したテーブルを挿入することができ、清書機能を書いたテストカバレッジの色つきの情報は、VDM\_COV、VDM\_NCOV形式を使っている。これら3つの形式はToolboxに含まれるVDM.dotファイルにincludeされている。


$\text{\LaTeX}$ ジェネレータは $\text{\LaTeX}$ マクロを適切な相当する形式のファイルに結合する：`vdmsl.sty` for  $\text{\LaTeX}$  と `vdmsl-2e.sty` for  $\text{\LaTeX}2_{\epsilon}$  である。これらのマクロとスタイルファイルはToolboxの一部として供給されてもいる。セクション 4.9 と付録 B で、生成された $\text{\LaTeX}$ ファイルを使用して $\text{\LaTeX}$ 環境をセットアップする方法の詳細について記述されている。

テストツールについてはセクション 4.9 でも論じられている。

清書機能はGUI、コマンドライン、Emacsいずれのインターフェースを使ってもアクセス可能である。



#### 4.7.1 GUI

清書機能を GUI で起動するには、マネージャーのプロジェクトビュー<sup>18</sup>で Toolbox に清書させたいファイルを選択し、 (清書) ボタンを押すことで起動する。

##### オプション設定

清書機能にはプロジェクトオプション ウィンドウの清書 タブで設定できるオプションがいくつかある。(図 27 参照) これらは以下のとおり。

索引の出力なし: 索引を生成しない。

デフォルト: 選択

定義のみの索引を出力: 関数、操作、型、状態、モジュール の定義の索引を生成する。

デフォルト: 非選択

定義と使用の索引を出力: 関数、操作、型、状態、モジュール の定義、型や関数・操作の使用された事象の索引を生成する。Microsoft Word では清書機能がこれらの使用のすべてを考慮に入れることができないため、Windows 環境ではこのオプションと最初のオプションに差異はない。

デフォルト: 非選択

テストカバレッジの色付け: 仕様のうちテストされていない箇所をハイライト表示する。このオプションが有効なとき、カバレッジ情報は通常のカバレッジ情報と一緒にテストカバレッジファイルに書き込まれる。

デフォルト: 無効

#### 4.7.2 コマンドラインインターフェース

```
vdmde -l [-nNr] specfile(s) ...
```

---

<sup>18</sup> マネージャー のモジュールビュー でもモジュールを選択することができる。清書機能は実際には選択したモジュールの含まれるファイル一式に適用される。これはもし特定のファイルが複数のモジュール定義を含んでいて、そのうちのいくつかだけを選択していた場合には、暗黙のうちに同じファイルの他 モジュールが処理に含まれることを意味する。



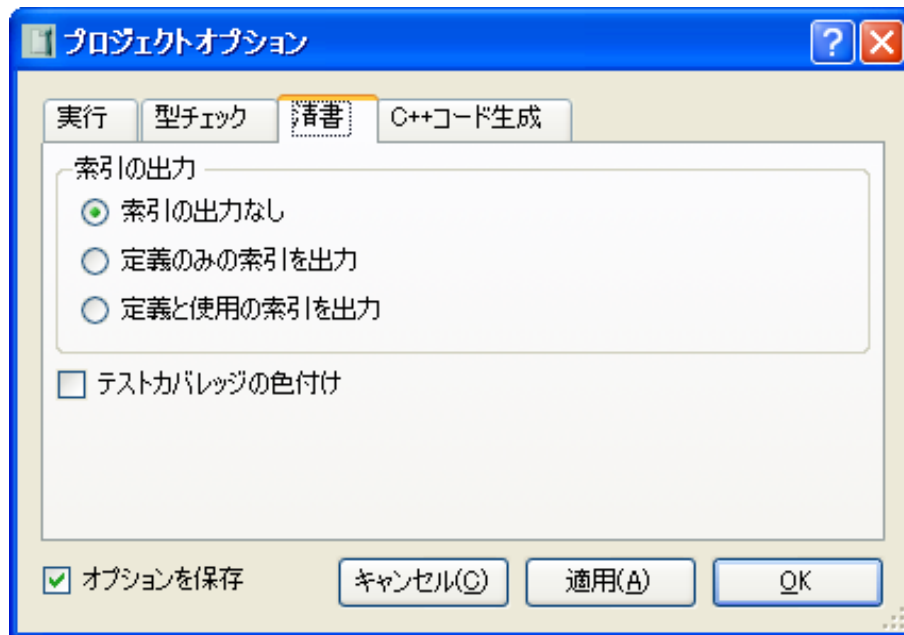


図 27: 清書機能のオプション設定

`vdmde` に `-i` オプションをつけると引数で指定した VDM-SL 仕様のファイルを入力ファイルとして清書したドキュメントを作成する。作成される文書のフォーマットは入力フォーマットに依存する。入力フォーマットが RTF の場合は、出力ファイルのファイル名は入力ファイルと同じ名前に `.rtf` がついた形となる。生成されたファイルは単独で直接 Word に取り込める。入力フォーマットが  $\text{\LaTeX}$  と VDM-SL 仕様の混ざったものだった場合は出力ファイルの名前は入力ファイルと同じ名前に `.tex` がついた形となる。この生成されたファイルは直接  $\text{\LaTeX}$  文書として扱える。

清書機能で利用できるオプションは以下のとおり：

- r テストカバレッジファイルから得られた追加のカバレッジ情報を挿入して清書機能を実行する。 $\text{\LaTeX}$  文書には、テストスイートによってどの部分が実行されたかされていないかを示すための特別なマクロが使用される。現在のバージョンでは、テストカバレッジファイルは `vdm.tc` という名前でワーキングディレクトリ (`pwd` コマンドで表示される) になくなくてはならない。テストカバレッジファイルは構文チェック機能を `-R` オプションつきで実行すると生成される (セクション 4.3 を参照)。

セクション B でこのコマンドで生成される L<sup>A</sup>T<sub>E</sub>X ファイルからテストカバレッジレポートを生成する方法の詳細を記述している。

- n RTF 文書に対してはこのオプションは、すべての関数・操作の定義に索引をつける。生成された .rtf ファイル内に VDM\_TC\_TABLE 形式で書かれたモジュール名を含めることですべての索引付きのテーブルを好きな箇所に挿入することができる。L<sup>A</sup>T<sub>E</sub>X 文書に対しては索引を生成するために使われるすべての関数、操作、型、状態モジュールの定義にはたらく L<sup>A</sup>T<sub>E</sub>X マクロを挿入する。そうすると makeindex ユーティリティを使って索引を生成することができる。
- N RTF 文書に対しては、-n オプションと同様。L<sup>A</sup>T<sub>E</sub>X 文書に対しては -n オプションと同じように働くが、すべてのアプリケーションの関数、操作、型、値に対してはたらくマクロも挿入する。
- W code 仕様文書の文字コードを code に設定する

#### 4.7.3 Emacs インターフェース

Emacs インターフェースでは、清書機能向けのコマンドは 1 つしかない。このコマンドは歴史的な理由から **latex** と呼ばれており、Emacs インターフェースで使えるほかのコマンドと同様コマンドプロンプトから入力しなくてはならない。

##### latex (l) [-nNr] file

清書機能が file とともに起動する。L<sup>A</sup>T<sub>E</sub>X フォーマットが使われていた場合、VDM-SL の部分が VDM-SL の  $V_{DMSL}$  マクロで数学的なフォントとして表示される。テキストの部分が合った場合、それらと VDM-SL の部分 (VDM-SL の  $V_{DMSL}$  マクロ) は file 中と同じ順番でファイルにマージされる。-n または -N オプションを使うと定義され使用された発生事象に索引が振られる (付録 B を参照)。


-r オプションはテストカバレッジファイルである vdm.tc に集められたカバレッジ情報を挿入する。RTF 形式のドキュメントでは VDM\_COV や VDM\_NCOV 形式が入力文書で定義されていなくてはならない。L<sup>A</sup>T<sub>E</sub>X 文書ではこのオプションはすべてのテストスイートでまだカバーできていないすべての仕様の部分に印がつくように  $V_{DMSL}$  マクロで色をつける。

## 4.8 VDM-SL から C++コード生成

VDM-SL から C++ へのコード生成のライセンス を持っていれば、Toolbox を使って仕様から自動的に C++のコードへ変換させることができる。ここではコードジェネレータの起動方法とどんなオプションがあるかについてのみ記述し、詳細は [12] で説明する。

C++ へのコード生成機能は GUI、コマンドライン、Emacs の各インターフェースを使ってアクセスすることができる。

### 4.8.1 GUI

GUIで C++へのコード生成機能は、まずマネージャでツールボックスに変換させたいファイルまたはクラスを選択し、 (C++生成) ボタンを押すことで起動する。ファイル/クラスが複数選択されていた場合、それらすべてが C++に変換される。

以下のコード生成向けオプションは、図 28 で示すプロジェクトオプション ウィンドウの C++コード生成 タブで設定することができる。

位置情報を出力する ランタイムエラーのための位置情報を含むコードを生成させる。

デフォルト : off

事前／事後条件をチェックする 関数の事前条件と事後条件、操作の事前条件のインラインチェックを含むコードを生成させる。

デフォルト : on

### 4.8.2 コマンドラインインターフェース

```
vdmde -c [-r] specfile, ...
```

vdmde コマンドに -c オプションをつけると、specfile からコードを生成する。仕様はまず解析され、構文エラーがなければ 'pos' タイプの型チェック がされる。最

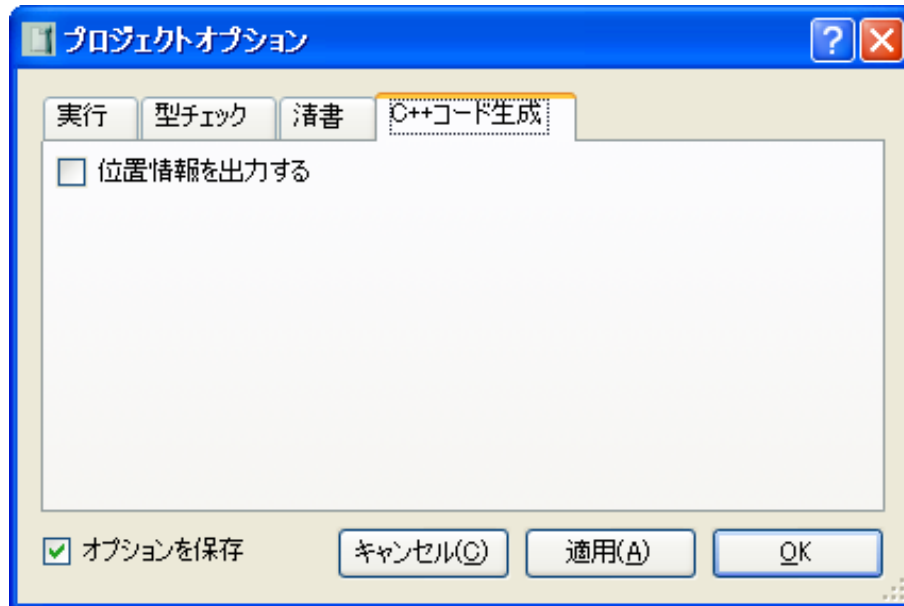


図 28: C++コード生成オプション設定

後に、型エラーが見つからなければ仕様がたくさんの C++ ファイルに変換される。生成されたコードの構造とその結び付け方は [12] に記述されている。

VDM-SL から C++ へのコード生成では追加のオプションがひとつ使用可能である。

-r ランタイム位置情報を生成した C++ コードに含める（詳細は [12] 参照）

-W code 仕様文書の文字コードを code に設定する

### 4.8.3 Emacs インターフェース

Emacs インターフェースではコード生成向けに使えるコマンドは 1 つしかない。このコマンドは **codegen** と呼ばれ Emacs インターフェースで使えるほかのコマンドと同様コマンドプロンプトから入力しなくてはならない。

**\*codegen (cg) [module] [rti]**

モジュール module.(何も指定されていない場合、現在のモジュール) . の

C++コードを生成する。rti オプションが使われるとランタイム位置情報が生成した C++コードに含まれる。

## 4.9 VDM モデルの体系的テスト

評価をサポートするものの一部として、Toolbox は VDM-SL 仕様のテストの便利ツールを提供する。これにはテストカバレッジの計測も含まれる。テストカバレッジの計測は与えられたテストスイートがどのくらい仕様をカバーできているかを見る手助けになる。これはテストスイートの実行中に評価された文や式についての特別なテストカバレッジファイル 情報を集めたことによってなされる。ここで記述されるアプローチはスクリプトベースのものであり、アプリケーションに多くのテストケースをさせることを意図したものである。セクション 3 で記述された `tcov` コマンドを使ったアプローチはテストケースが少ないとき向けである。

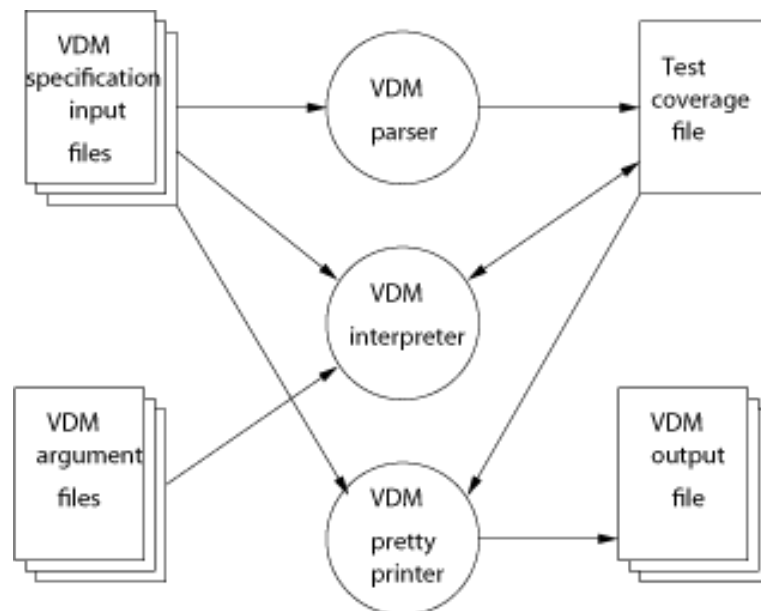


図 29: VDM モデルの体系的テスト

テストカバレッジリポートを作成するには3つのステップがある（図 29 参照）

1. *test coverage file* を用意する。片方の入力フォーマットの VDM-SL のファイルはまず特別なオプション付きの VDM-SL 構文解析ツールに渡される。これが仕様の構造についての情報は含むが何の定義も含まないテストカバレッジファイルを生成する。

2. VDM-SL インタープリタがたくさんの小さいファイルを引数として呼ばれる。インタープリタはすべての仕様ファイルとテストカバレッジファイル、それに加えて評価結果を返す引数ファイルに使用され、テストカバレッジファイルの異なる構成要素がどれぐらいの頻度で実行されたかについての情報を更新する。インタープリタはテスト環境で考慮の対象となる程度まで繰り返し呼び出される。
3. 最後に、すべての仕様ファイルとテストカバレッジファイルを入力して、テストカバレッジ情報の詳細を示す仕様の清書版を生成する特別なオプションつきで清書機能を使用される。VDM-SL 仕様の入力ファイルでは VDM-SL の定義を含まないテキスト形式の部分だけがこのプロセス実行中に更新されることに注意。もし VDM-SL 部分の変更があっても、テストカバレッジファイルはその情報をどのように VDM-SL の仕様ファイルに反映したらよいかわからない。ウインドウズ上、ツールボックスの現在のバージョンではテストカバレッジファイルが `vdm.tc` という名前にしておかなくてはならず、ワーキングディレクトリにおいておかなくてはならない。

#### 4.9.1 テストカバレッジファイルの準備

テストカバレッジ情報を生成するにはコマンドプロンプトから実行しなければならない（ウインドウズ上ではウインドウズセットアップのプログラム一覧からコマンドプロンプトを選択、Unix では通常のシェル）。構文解析ツールは `-R` オプションで起動する。パラメータの詳細についてはセクション 4.3.3 を参照のこと。

例：

```
"vdmhome/bin/vdmde" -p -R vdm.tc sort.rtf
```

#### 4.9.2 テストカバレッジファイルの更新

テストスイートは通常ディレクトリ階層で構成され、これは小さい引数となるファイルがテストされることになっているものに依存する異なるカテゴリーに置かれている。開発中のプロジェクトでは、このようなテスト環境を構築し、テストプロセスを自動化するスクリプトファイルを作成し、期待される結果と実際の結果を比較することが望ましい。付録 E には、ウインドウズおよび Unix 両方に向け

たこのようなスクリプトファイルの例がある。テストスクリプトはツールボックスのコマンドラインインターフェースから-R オプションつきで呼ばれなくてはならない。使用可能な引数の詳細はセクション 4.5.3 を参照のこと。

例：

```
"vdmhome/bin/vdmde" -i -R vdm.tc -O sort.res sort.arg sort.rtf
```

#### 4.9.3 テストカバレッジの統計データ作成

このようなテストスイートの実行結果は適切なオプションを有効にしていれば清書機能を使って表示することができる。清書機能は GUI およびコマンドライン、Emacs の各インターフェイスを使ってアクセス可能である。組み込まれているカバレッジ情報を有効にするオプションを使うことが大切である。 利用可能な引数の詳細については、セクション 4.7 を参照のこと。

例：

```
"vdmhome/bin/vdmde" -lr sort.rtf
```

生成された清書版のファイルでは、仕様の関数および操作のカバレッジのパーセンテージのテーブルを表示することができる。加えて、そのテストでどのぐらいの関数と操作がカバーできているかを示す詳細なテストカバレッジ情報も利用できる。

コマンドラインインターフェースからでも GUI のインタプリタの Dialog 画面からでも入力できる **rtinfo** コマンドは、下記で説明するようにテストカバレッジ情報を表示する。

##### **rtinfo vdm.tc**

このコマンドを適用する前に、テストスイートの **vdm.tc** にランタイム情報が収集されていなくてはならない。テストスイートは読み込まれ、すべての関数と操作の概要が表示される。リストの項目それぞれに対して、評価の回数と定義のカバレッジのパーセンテージが表示される（このパーセンテージは、評価済みの関数/操作の式の数すべてで割ったもの）。リストのすべてのパーセンテージの平均だが、テストカバレッジファイルの全部のカバレッジも表示される。



#### 4.9.4 L<sup>A</sup>T<sub>E</sub>X を使ったテストカバレッジ例

入力フォーマットに依存するテストカバレッジについて、VDM-SL 入力ファイルの異なる部分のやり方が違う。テストカバレッジ情報の RTF ファイルへの組み込み方を示す例がセクション 3.9 にある。L<sup>A</sup>T<sub>E</sub>X ファイルのプロセスは全く異なり、ここではこのマニュアルを通じて使用されている Sorting の例であらわされうソートのアルゴリズムのひとつで説明する。仕様は `vdmhome/examples/sort/sort.vdm` ファイルにある。

この例では、`DoSort([-12,0,45])` が `DoSort` の仕様のどの程度をカバーするかを示すため評価される

まずは構文解析ツールを使ってテストスイートを生成する。

```
prompt> vdmde -p -R vdm.tc sort.vdm
Parsing and installing "./sort.vdm" ...
prompt>
```

これで argument ファイル `sort.arg` を評価することができるようになった。このファイルには VDM-SL 関数: `DoSort` の呼び出しが含まれている。

```
prompt> vdmde -i -R vdm.tc sort.arg sort.vdm
Evaluating the arguments ...

The expression evaluates to :
[ -12,0,45 ]
prompt>
```

インタープリタが `-R` オプションとともに呼ばれると、テストカバレッジファイル `vdm.tc` を更新する。

ファイル `vdm.tc` に記録されたばかりの sorting 仕様のカバレッジレベルは Toolbox で表示される。 `rtinfo` コマンドを使うと仕様のすべての関数と操作を一覧にしたテーブルが、関数・操作が呼び出された回数とカバレッジのパーセンテージと一緒に表示される。

パーセンテージはそれぞれ、相当する関数/操作内で評価済みの式の数を当該関数/操作内の式の総数で割ったものである。テストカバレッジファイル全体の合計カバレッジも（個々の関数/操作のパーセンテージの平均であるが）表示される。図 30 はこれを示したものである。

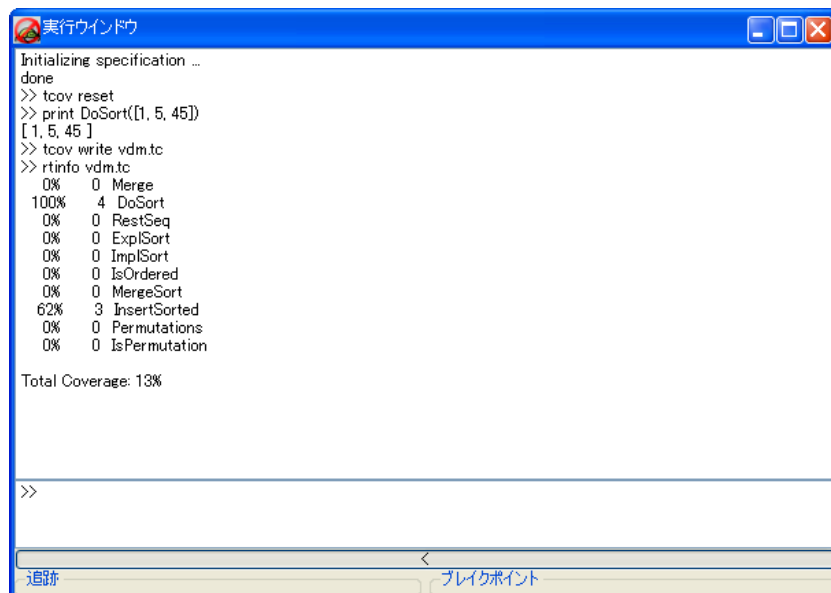


図 30: Showing test coverage information in the Toolbox

この L<sup>A</sup>T<sub>E</sub>X フォーマットの入力ファイルと一緒に清書機能呼び出してみる。

```
prompt> vdmde -lr sort.vdm
Generating latex
Latex generated to file sort.vdm.tex
prompt>
```

清書機能が `-r` オプションつきで呼び出されると先ほどのテストでカバーできていない `DoSort`、`InsertSorted` 関数の部分にマークがつく。古いバージョンの L<sup>A</sup>T<sub>E</sub>X は色付けをサポートしていないため、`-r` オプションは L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> に対してのみ使われることに注意。`sort.vdm.tex` ファイル上で L<sup>A</sup>T<sub>E</sub>X を走らせると図 31 のように結果が表示される。

`InsertSorted` の case 文の `others` 節が、まだカバーできていないが、これはすでにソート済みの列を引数にして `DoSort` を呼んでいるためである。

```
1.0  DoSort :  $\mathbb{R}^* \rightarrow \mathbb{R}^*$ 
.1   DoSort (l)  $\triangleq$ 
.2     if l = []
.3     then []
.4     else let sorted = DoSort (tl l) in
.5       InsertSorted (hd l, sorted);

2.0  InsertSorted : PosReal  $\times$  PosReal*  $\rightarrow$  PosReal*
.1   InsertSorted (i, l)  $\triangleq$ 
.2     cases true :
.3       (l = [])  $\rightarrow$  [i],
.4       (i  $\leq$  hd l)  $\rightarrow$  [i]  $\curvearrowright$  l,
.5       others  $\rightarrow$  [hd l]  $\curvearrowright$  InsertSorted (i, tl l)
.6   end
```

図 31: Sorting example のテストカバレッジ

付録 B に、VDM-SL 仕様と L<sup>A</sup>T<sub>E</sub>X 部分の分け方について詳細な記述がある。

## L<sup>A</sup>T<sub>E</sub>X テストカバレッジ向け入力ファイルフォーマット

このセクションでは、仕様のテストスイートでカバーできていない色つきの部分と L<sup>A</sup>T<sub>E</sub>X 形式でカバレッジのパーセンテージを示すテーブルをどのように組み込むかを記述する。上で使用されたソートの例と同じものを使ってあらわすこととする。テストカバレッジテーブルの生成と色の生成について順に議論する。

### テストカバレッジテーブルと L<sup>A</sup>T<sub>E</sub>X テストカバレッジ環境

L<sup>A</sup>T<sub>E</sub>X 環境で関数・操作の呼び出し回数およびカバレッジのパーセンテージを記述するテーブルを挿入するには、`rtinfo` コマンドを使用する。まずこの環境の使用を表す例を示し、形式 BNF ライクな定義の使用の部分を示す。

ソートの例でフラット形式の仕様が定義されており、`rtinfo` 環境は `DefaultMod` モジュール向けに定義されている。 `rtinfo` 環境は以下のようにになっている：

Test Suite : vdm.tc  
Module : DefaultMod

Name	#Calls	Coverage
DoSort	4	✓
InsertSorted	3	62%
<b>Total Coverage</b>		<b>76%</b>

図 32: テストカバレッジテーブルの例

```
\begin{rtinfo}
[TotalxCoverage]{vdm.tc}[DefaultMod]
DoSort
InsertSorted
\end{rtinfo}
```

`rtinfo` 環境の最初の引数（例では `TotalxCoverage`）はオプションである。関数・操作名のテーブルの欄の幅を指定するのに使われる。幅は引数で指定したものになる。第2引数（例では `vdm.tc`）はテストカバレッジファイル名になる。この引数は必須である。第3引数はオプションで、テーブルを特定のクラスのものに制限したい場合その（例では `DefaultMod`）モジュール名となる。この引数が省略された場合、すべてのテストカバレッジファイル中のクラスがテーブルに一覧表示される。

`rtinfo` 環境下では、特定の関数・操作名が記述されるが、これはテーブル内に一覧表示されている場合だけである。そうでない場合はすべての関数・操作が一覧表示される。例では関数 `DoSort`, `InsertSorted` のみが一覧表示されている。

実行結果のテーブルは図 32 に示す。

テストカバレッジ環境の構文は以下のように定義されている：

```
test coverage environment = '\begin{rtinfo}', test coverage section,
                             '\end{rtinfo}';
```

```
test coverage section = [ long name ], test suite file, [ module name ],
                        [ function list ] ;
```

`long name = '[' , string , ']' ;`

`test suite file = '{' , file identifier , '}' ;`

`file identifier = identifier , { '.' , identifier } ;`

`module = '[' , identifier , ']' ;`

`function list = { identifier } ;`

## Colouring

テストスイートでカバーできていない仕様の部分を色つきで示す機能は  $\text{\LaTeX}2\epsilon$  を使っていれば使える。テストカバレッジの色つき情報を入れるには  $\text{\LaTeX}$  ファイルは VDM-SL の仕様から生成されたものでなくてはならない。GUI ではテストカバレッジの色付けオプションを有効にするか、コマンドラインまたは `emacs` で `-r` オプションをつける必要もある。

以下の例では、色つけを説明するのに必要な拡張スタイルのファイルと定義を示す。

```

\documentclass[dvips]{article}
\usepackage[dvips]{color}                <--- extra style
\usepackage{vdmsl-2e}

\definecolor{covered}{rgb}{0,0,0}        %black   <--- extra
                                           %         definition
\definecolor{not-covered}{gray}{0.5}     %gray     <--- extra
                                           %         definition

\begin{document}
...
\end{document}

```

生成された L<sup>A</sup>T<sub>E</sub>X コードには、マクロ `\color{covered}` と `\color{not-covered}` が仕様のテストスイートでカバーされた/されていない部分の前にそれぞれに挿入されている。`\definecolor` マクロはカバーされたところは黒で、カバーされていないところはグレーで表すよう定義されている。結果は上記図 31 に示す。

InsertSorted の case 式の others 節が L<sup>A</sup>T<sub>E</sub>X の出力結果でグレーになっているということは、その部分がカバーされていないことを表す (DoSort はすでにソート済みの列からしか呼ばれていないため)。この情報をもとに、テストスイートは仕様のより多くの部分をカバーすることができるようになる。

カラー画面・カラープリンタのためにカバーされていない箇所は赤を使うこともできる。その場合の定義マクロは以下ようになる：







```

\definecolor{not-covered}{rgb}{1,0,0}   %red

```

色付け機能を使うために、`rtinfo` 環境も入っていないと不行き届き。これは色をつけるのに使われる情報はテストカバレッジファイルに保存されているからである。

## 参考文献

- [1] I.P. Dickinson and K.J. Lines. Typesetting VDM-SL with VDM-SL macros. Technical report, National Physical Laboratory, Teddington, Middelsex, TW11 0LW, UK, July 1995.
- [2] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.
- [3] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-oriented Systems*. Springer, New York, 2005.
- [4] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. VDM++  IuWFNgwVXeiv .  j , 2010.  .
- [5] P. G. Larsen and B. S. Hansen and H. Brunn N. Plat and H. Toetenel and D. J. Andrews and J. Dawes and G. Parkin and others. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.
- [6] SCSK. *The Dynamic Link Facility*. SCSK.
- [7] SCSK. *The Rose-VDM++ Link*. SCSK.
- [8] SCSK. *The VDM++ Language*. SCSK.
- [9] SCSK. *VDM-SL Installation Guide*. SCSK.
- [10] SCSK. *The VDM-SL Language*. SCSK.
- [11] SCSK. *VDM-SL Sorting Algorithms*. SCSK.
- [12] SCSK. *The VDM-SL to C++ Code Generator*. SCSK.
- [13] SCSK. *VDM Toolbox API*. SCSK.

## 用語集

**C++コード生成機能:** 仕様書から C++ のコードを自動生成する。Toolbox から C++コード生成機能 にアクセスするためには別ライセンスが必要である。

**デバッガ:** デバッガを使えば仕様書の振る舞いを調査することができる。デバッガは仕様書を実行しアプリケーションの関数や操作 でブレークすることもできる。実行中いつでも、仕様書内のローカルまたはグローバルな状態、ローカル変数などを調査することができる。

**動的セマンティクス:** 動的セマンティクスは言語の意味を記述する。すなわち動的セマンティクスは実行された場合に言語がどう振舞うかを記述する。

**Emacs:** ASCII エディタ。

**GUI:** グラフィカルユーザーインターフェース

**インタープリタ:** インタープリタは言語の動的動作に従って仕様書を解釈する。いわばプログラム/仕様書を実行する。

**L<sup>A</sup>T<sub>E</sub>X:** 一般的な組版システム

**清書機能:** ファイルを処理して VDM-SL の入力ファイルの VDM-SL の箇所の清書版を生成する。出力フォーマットは入力フォーマットに依存する。

**プロジェクト:** 仕様書を構成する ASCII のファイル名の集合

**RTF:** 「Rich Text Format」の頭字語。Microsoft Word で使用できるフォーマットのひとつ。

**セマンティクス:** 言語の意味を記述したもの

**仕様書:** (おそらく) 異なる入力フォーマットを使って書かれた 1 つ以上のファイルからなるシステムの VDM-SL モデル

**静的セマンティクス:** 構文的に正しい仕様書を適格にするため (矛盾のない意味を持たせるため) に従わなくては成らない言語の記号間の関係を記述したもの。適格な仕様書とは典型的に正しい仕様書とも言える。



**構文:** 言語の構文は、言語の記号要素（キーワード、識別子など）がどのように関連しているかを記述したものである。構文は言語中で記号がどのように命令されるかを記述しており、命令の意味を記述するものではない。

**構文チェック機能:** 仕様書の構文が正しいかどうか確認する。

**テストカバレッジ情報:** 仕様書の構成物が各々何回実行されたについての情報

**テストカバレッジファイル:** テストカバレッジ情報を含むファイル。

**型チェック機能:** 仕様書の型が正しいかどうかチェックする。'def' タイプと'pos' タイプ 2 種類のチェックがある。

**VDM:** ウィーン開発手法

**VDM-SL:** *Vienna Development Method*. の形式仕様言語。ISO 標準言語である [5]。

**VDM++:** オブジェクト指向仕様言語。ISO VDM-SL の拡張。

**Well-formedness:** 仕様書が言語の構文、静的セマンティクスに関して適格であるということ。

## A VDM技術の情報源

この短い付録には VDM や Toolbox を使う上で参考になる情報源を記述する。

### モデリングの本

Toolbox を使うのであれば下記のテキストがほぼ適切だ。抽象化や ISO 標準 VDM-SL の記法のサブセットを使った形式モデルの分析は Toolbox にサポートされている。より難解な数学的記法よりも ASCII 記法が使われ、内容も VDM 技術の産業アプリケーションの例に基づいて解説されている。より重要なのは、それが Toolbox あるいは Toolbox の特別なチュートリアル版（本に付属の CD-ROM に含まれている）を使って取り組むたくさんの演習を含んでいることである。

サポート情報（追加の演習、スライド、Web サイトへのリンクなど）が <http://www.csr.ncl.ac.uk/modelling-book/>にある。

下記のような本が存在する：

J. フィッツジェラルド・P.G. ラーセン / 著,  
荒木 啓二郎・張 漢明・荻野 隆彦・佐原 伸・染谷 誠 / 訳,  
“ソフトウェア開発のモデル化技法”,  
岩波書店 2003,  
ISBN 4-00-005609-3

佐原 伸 / 著,  
“～ソフトウェアトラブルを予防する～ 形式手法の技術講座”,  
ソフト・リサーチ・センター 2008,  
ISBN 978-4-88373-258-6

John Fitzgerald and Peter Gorm Larsen,  
“Modelling Systems: Practical Tools and Techniques in Software Development”,  
Cambridge University Press 1998,

ISBN 0-521-62348-0

<http://uk.cambridge.org/order/Webbook.asp?ISBN=0521623480>

(本書は絶版となっている)

John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat and  
Marcel Verhoef

“Validated Designs for Object-oriented Systems”

Springer, New York, 2005

ISBN 1-85233-881-4

[http://www.springer.com/east/home/generic/-  
search/results?SGWID=5-40109-22-33837368-0](http://www.springer.com/east/home/generic/-search/results?SGWID=5-40109-22-33837368-0)

## Web サイト

Web サイトには一般的な形式手法や VDM についてのたくさんの情報が載っている。ここでは他に使えるサイトへのリンクを含むものをいくつかあげる。

**The VDM Web Site** VDM に関する基本的な情報を含む Web ページ。参考文献、VDM メーリングリストの情報、VDM の例題の置き場へのリンクなどが含まれる。

<http://www.csr.ncl.ac.uk/vdm/>

**The VDM Bibliography** VDM 理論、実践、体験などの論文や文献の検索可能な参考文献。 <http://liinwww.ira.uka.de/bibliography/SE/vdm.html>

**The VDM++ Bibliography**

<http://liinwww.ira.uka.de/bibliography/SE/vdm.plus.plus.html>

**The NASA Formal Methods Page** 一般的な形式手法の良い導入教材

<http://shemesh.larc.nasa.gov/fm.html>

**Formal Methods Europe** この組織の Web ページには特に興味深いアプリケーションのデータベースがある。形式技術のアプリケーションが一覧になっているデータベースだが、そのほとんどが商用または産業用の内容である。

<http://www.fmeurope.org/>



**The Formal Methods Archive** かなり大きな形式手法のアプリケーションや研究についての情報源。形式手法の会社や組織へのリンク、導入として推奨される論文や本など。

<http://www.comlab.ox.ac.uk/archive/formal-methods/>

**VDM information Web site** VDM・VDMToolsに関する情報発信、意見交換などを行なうためのサイト。

<http://www.vdmttools.jp/>

## B VDM-SL と $\text{\LaTeX}$ の結合

このセクションでは、VDM-SL の仕様部分を含む  $\text{\LaTeX}$  文書の構築の仕方について、一般的なことを記述する。

### B.1 仕様ファイルのフォーマット

システムを処理するのに  $\text{\LaTeX}$  文書を使用したい場合、2つの異なる入力フォーマットを使うことができる：ひとつは純粋な VDM-SL の ASCII 仕様であり、もうひとつは原文の混ざった ASCII 仕様である。後者は仕様の部分とそうでない部分が “ $\text{\begin{vdm\_al}}$ ” と “ $\text{\end{vdm\_al}}$ ” に囲まれているかそうでないかで区別される。仕様ブロックの外側にあるテキスト部分は解析ツールには無視される（清書機能は使う）。“ $\text{\begin{vdm\_al}}$ ” は仕様中の任意の箇所に置くことはできず、`module`, `state`, `functions`, `operations`, `values`, `types` といったキーワードの前か `functions`, `operations`, `types`, `values` の定義の前にしかおくことができない。これは例えばインポートされたモジュールのセクションや関数の中にテキスト部分を挿入することができないということを意味している。

ファイル

```
vdmhome/examples/sort/sort.vdm
```

にどのように仕様部分とテキスト部分が混ざっているかの例がある。

### B.2 $\text{\LaTeX}$ 文書のセットアップ

VDM-SL と  $\text{\LaTeX}$  を結合するとき、通常の機能は  $\text{\LaTeX}$  でも  $\text{\LaTeX}2\epsilon$  でも使うことができるが、テストカバレッジとの結合は  $\text{\LaTeX}2\epsilon$  を使わないとできない。文書の色付け機能など  $\text{\LaTeX}2\epsilon$  でのみ使用できる機能がいくつかあるからだ。

以下の  $\text{\LaTeX}$  コードの例は VDM-SL の部分を含む一般的な  $\text{\LaTeX}$  文書を作成するためには含まれていなくてはならない  $\text{\LaTeX}$  形式のファイルを示す：

`vdmsl-2e` スタイルファイルは `Toolbox` に含まれている。

```
\documentstyle[vdmsl]{article}

\begin{document}
...
\end{document}
```

図 33: 旧バージョン L<sup>A</sup>T<sub>E</sub>X 文書の例

```
\documentclass{article}
\usepackage[vdmsl-2e]{}

\begin{document}
...
\end{document}
```

図 34: L<sup>A</sup>T<sub>E</sub>X2<sub>ε</sub> 文書の例

L<sup>A</sup>T<sub>E</sub>X の見出しは VDM-SL 仕様ファイルのひとつか VDM-SL の仕様を含む単独のファイルのどちらにも挿入できる。どちらのケースでも仕様のファイルは Toolbox により L<sup>A</sup>T<sub>E</sub>X ファイルに変換されていなくてはならないが、これは GUI から清書ボタンを押すか Emacs インターフェースで latex コマンドを使うかコマンドラインから vdmde を -l オプションつきで使うかのどれかで可能である。ソートの例では見出しは sort.vdm ファイルに挿入されている。

## 行番号つけ

生成された L<sup>A</sup>T<sub>E</sub>X ファイルにあるすべての定義はデフォルトで定義番号と行番号が与えられる。これらの番号は以下のコマンドを使えば削除することができる。

```
\nolinenumbering
\setindent{outer}{\parindent}
\setindent{inner}{0.0em}
```

詳細は [1] を参照のこと。

## インデックス

インデックス番号をつくるマクロは清書出力の L<sup>A</sup>T<sub>E</sub>X 文書の一部として生成される。インデックス番号は最終的な L<sup>A</sup>T<sub>E</sub>X 文書のページを参照する。インデックスは2つのレベルで生成することができる：関数、操作、型、状態、モジュールの定義すべてに対するインデックスと関数、操作、型のすべての使用に対するインデックスである。定義に対するインデックスマクロは GUI で定義の索引を出力オプションを有効にするか、コマンドラインまたは Emacs インターフェースで `-n` オプションを使うかで生成される。定義または使用に対するインデックスマクロいずれも GUI で定義のインデックスを出力するオプションを有効にするか、コマンドラインまたは Emacs インターフェースで `-N` オプションを使うかで生成される。

インデックス番号はインデックスがどんな種類の構成要素を参照するかを分類するため、自動的に異なる L<sup>A</sup>T<sub>E</sub>X マクロに挿入される。定義に対するマクロは：

- `StateDef` 状態の定義されている箇所を示す。
- `TypeDef` 型が定義されている箇所を示す。
- `FuncDef` 関数または操作が定義されている箇所を示す。
- `ModDef` モジュールが定義されている箇所を示す。

使用するマクロは：

- `TypeOcc` 型が使用されている箇所を示す。
- `FuncOcc` 関数または操作 が使用されている箇所を示す。ドキュメントで明確に定義されている関数/操作の仕様についてのみインデックスが振られる。

これらの L<sup>A</sup>T<sub>E</sub>X マクロはインデックス付けを使用するためには L<sup>A</sup>T<sub>E</sub>X 文書の最初で定義されていなくてはならない。例を以下に示す：

```
\newcommand{\StateDef}[1]{\bf #1}
\newcommand{\TypeDef}[1]{\bf #1}
\newcommand{\TypeOcc}[1]{\it #1}
```

```
\newcommand{\FuncDef}[1]{\bf #1}
\newcommand{\FuncOcc}[1]{#1}
\newcommand{\ModDef}[1]{\tiny #1}
```

インデックスに入れておきたい場合は4つの追加箇所を  $\text{\LaTeX}$  文書に含めておく必要がある。

1. `makeidx` スタイルオプションをインクルードする ( $\text{\LaTeX}$  なら `\documentstyle`、 $\text{\LaTeX}2\epsilon$ .. ならパッケージに含まれる)
2. 文書の序文に `\makeindex` をインクルードする
3. マクロ `InstVarDef`, `TypeDef`などを定義する
4. `\printindex` を文書のインデックスを入れたい箇所に含める

## サポート外構成要素


現状、 $\text{\LaTeX}$  の清書機能ではサポートしていない構文の構成要素が1つある：コメントを活字に組むことができないのだ。 $\text{\LaTeX}$  の清書機能では単に無視されるだけである。VDM-SL コメントを使用する代わりに、仕様とテキストを混ぜて使うことを推奨する。



## C VDMTools 環境の設定

個人の好みによって、ツールボックス向けに多くのオプションを設定することができる。これらについて以下で記述する。

### C.1 一般

Toolbox は、たくさんのフォントをサポートしているが、これにより対応する範囲の活字が仕様自体（仕様で使われる識別子の名前など）とそれに伴う一般的なテキスト形式の両方で使用できるようになる。関連する言語のサポートは Toolbox の実行される OS（Windows または Unix）のレベルでまず適切にインストールされ、それからツールオプション ウィンドウの **一般** タブ（図 35 参照）で Toolbox に設定することができる。この画面はプロジェクトツールバー/メニューの **ツールオプション** () の項目を選択すると起動する。**ツールオプション** の **一般** タブでフォントの選択ボタンを押すと利用できるフォントの一覧（OS レベルで）を含むブラウザが開き、好きなフォントを選択することができる。最後に、同じ画面の Text Encoding メニューから適切なエンコーディングを選択する。

また、一般タブでは、構文の色付けと自動構文チェックをするかどうかを選択することも可能である。（両方ともデフォルトで選択されている。）構文の色付けが選択されている時に、VDM の構文が持つキーワードはソースウィンドウで強調表示される。自動構文チェックが選択されている場合、ファイルはファイルシステム上で新たに保存された時に自動で構文チェックされる。

ログ行数は、ログウィンドウ、および実行ウィンドウの最大バッファ行数を指定する。これらのウィンドウは出力のバックスクロールをサポートしているので、出力に伴って、Toolbox の使用メモリ量が増加する。デフォルト値はどちらも 1000 行に設定される。0 を設定すると無限大となる。

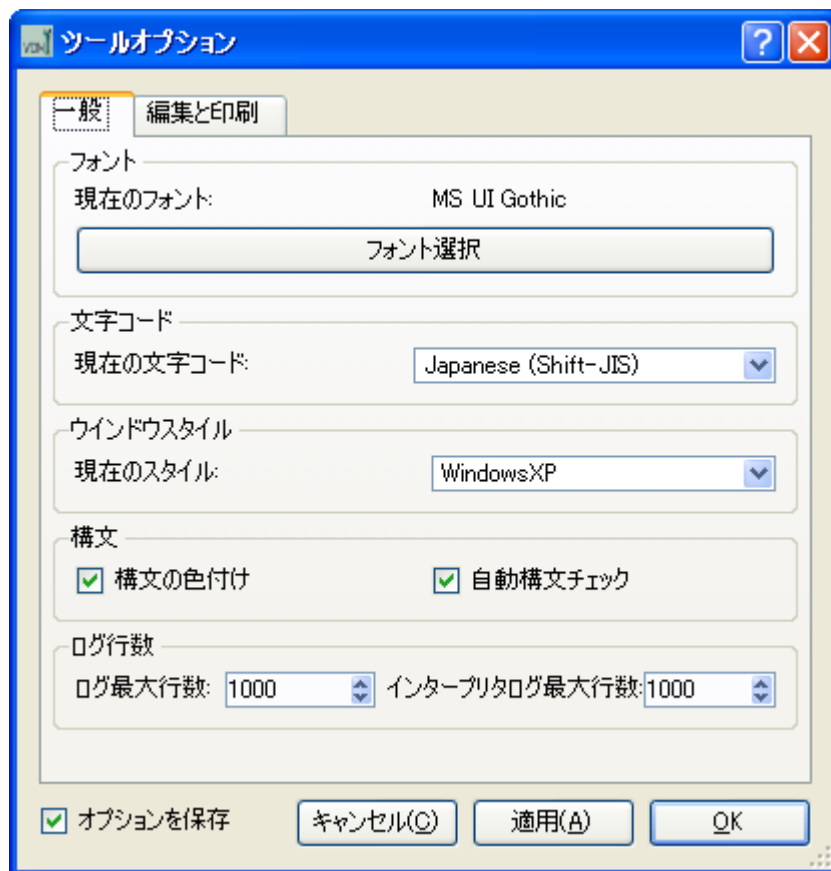



図 35: 一般オプションの設定

Toolbox で利用可能な活字の範囲は一般的な Qt インターフェースでサポートされているものに比べて限られている。テキストエンコーディングの設定が現在サポートされているものを表している場合、選択が可能になる。

## C.2 インターフェースオプション

以下のインターフェースオプションは、ツールオプション ウィンドウの編集と印刷 タブで設定可能である。この画面は図 36 に示すプロジェクト ツールバー/メニューから ツールオプション (  ) の項目を選択することで起動する。

外部エディタ: Toolbox から起動することができる外部エディタを定義するオブ

ション

エディタ名: 使用する外部エディタの名前。Unix でのデフォルト値は `emacsclient`。Windows では `notepad`。MS Word を直接起動することはできない。

ファイルを開く方法: デフォルト値: `+%l %f` Windows では、サクラエディタ、秀丸、TeraPad、GVim はエディタの指定のみでよい。

複数のファイルを開く方法: デフォルト値: `%f`

印刷命令: デフォルト: `lpr` **Note:** Windows プラットフォームでは使用不可能。Windows ではパイプや Print アイコンは出現しない。

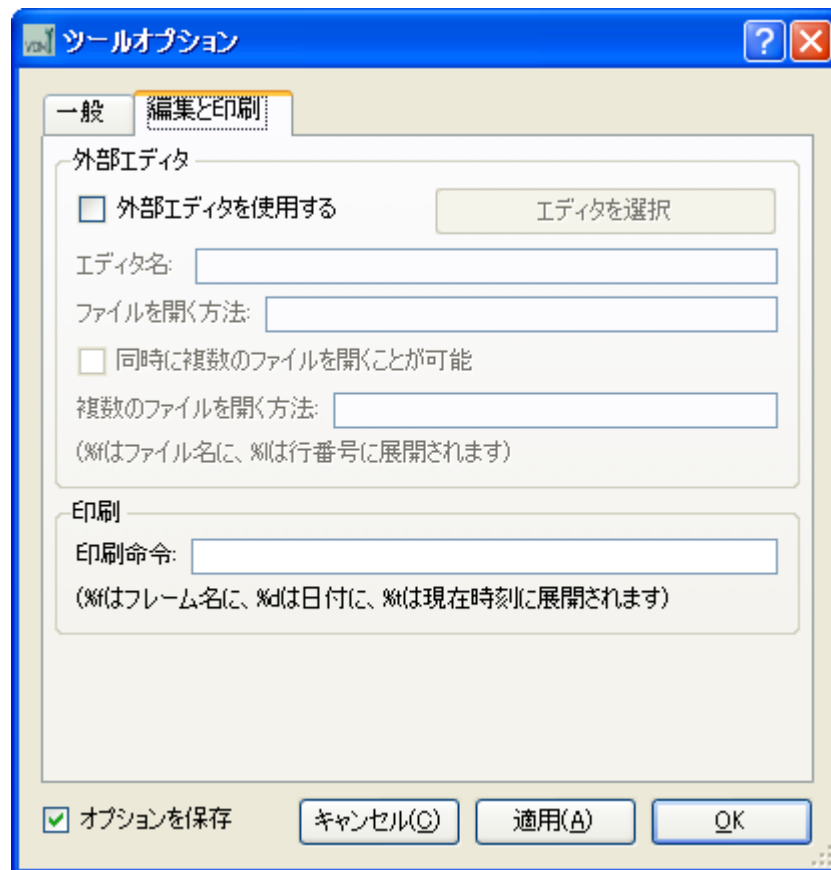


図 36: エディタと印刷オプションの設定

## D Emacs インターフェース

Emacs インターフェースは Toolbox の利用可能な異なる Unix プラットフォームでのみサポートされている。ファイル `vdmde.el` は Emacs から直接 Toolbox の機能にアクセスすることを可能する Emacs マクロを含む。セクション 3 からのガイドツアーは `sort-init.rtf` ファイルの代わりに `sort.init` ファイルを使うことにより以下に従う。

Toolbox をインストールし、ドキュメント [9] に記述されているように `.emacs` file ファイルを更新すると、Emacs エディタを使うことができる。

M-x `vdmde` (`vdmde` の引数としてメタキーと `x` キーを押す) とタイプすることで Emacs エディタから VDM-SL 開発環境 (`vdmde`) を起動する。これはどの Emacs バッファからでもできる。これで VDM-SL 形式の仕様を含むと思われるファイルの名前にプロンプトが表示されているはずだ。`sort.vdm` とタイプしてリターンキーを押す。

`vdmde` コマンドは入力ファイルを解析し、構文エラーが見つかった場合は Emacs ウィンドウが2つに分かれる。半分（以後仕様ウィンドウと呼ぶ）は `sort.vdm` ファイルを表示しており、もう半分は `vdmde` のコマンドウィンドウとなる（以後コマンドウィンドウと呼ぶ）。構文エラーは仕様ウィンドウでは `=>` マークで示される。

これはセクション 3 での説明と似ており、以下の例でガイドツアーの続きをすることができる。コマンドウィンドウのコマンドラインで `help` とタイプすれば、異なる特性へのアクセス方法の概要を見ることができる。

## E Sort 例題向けテストスクリプト

ここで示すテストスクリプトは2つのスクリプトから構成されている。

- 単独の argument をテストする特別なスクリプト。パラメータに argument ファイル名を取る。
- いくつもの argument ファイルをループするトップレベルのテストスクリプト。これらの argument ファイルはディレクトリ階層で構成される。各々の argument ファイルはファイル名とともに特別なテストスクリプトを呼び出す。

テストスクリプトはプラットフォームに依存するため、この付録はプラットフォームによって2つの部分に分かれる。より高度なテストスクリプトを作成することもできる。ここで示されているものは基本的なアプローチを実演することを意図した単純なものである。

### E.1 Windows/DOS プラットフォーム

トップレベルのテストスクリプトは `vdmlloop.bat` という名前であり、DOS プロンプトから実行することができる。このファイルは Toolbox から利用可能であり、`examples/sort/test` ディレクトリにある。以下のようになる：

```
@echo off
rem -- Runs a collection of VDM-SL test examples for
rem -- the sorting example
set SPEC=..\sort.rtf

vdmde -p -R vdm.tc %SPEC%

for /R %%f in (*.arg) do call vdmtest "%%f"
```

argument ファイルをひとつだけとるテストスクリプトは `vdmtest.bat` と呼ばれる。このファイルも Toolbox から利用可能であり以下のようなものである。

```
@echo off
rem -- Runs a VDM test example for one argument file

rem -- Output the argument to stdout (for redirect) and
rem -- "con" (for user feedback)
echo VDM Test: '%1' > con
echo VDM Test: '%1'
set SPEC=..\sort.rtf

vdmde -i -R vdm.tc -O %1.res %1 %SPEC%

rem -- Check for difference between result of execution
rem -- and expected result.
fc /w %1.res %1.exp

:end
```

## E.2 UNIX プラットフォーム

トップレベルのテストスクリプトは `vdmloop` と呼ばれ通常のシェルから実行することができる。このファイルは Toolbox で利用でき、`examples/sort/test` ディレクトリにある。以下参照：

```
#!/bin/sh

## Runs a collection of VDM-SL test examples for the sorting example.
SPEC=../sort.vdm

## Generate the test coverage file vdm.tc
vdmde -p -R vdm.tc $SPEC

## Find all argument files and run them on the specification.
find . -type f -name \*.arg -exec vdmtest {} \;
```

argument ファイルをひとつだけとるテストスクリプトは `vdmtest.bat` と呼ばれ

る。このファイルも Toolbox から利用可能であり以下のようなものである。

```
#!/bin/sh

## Runs a VDM test example for one argument file.

## Output the argument to stdout (for redirect) and
## "/dev/tty" (for user feedback)
echo "VDM Test: '$1'" > /dev/tty
echo "VDM Test: '$1'"

SPEC=./sort.vdm

## Run the specification with argument while collecting
## test coverage information, and write the result to an
## output file.
vdmde -i -R vdm.tc -O $1.res $1 $SPEC

## Check for difference between result of execution
## and expected result.
diff -w $1.res $1.exp
if test $? = 0 ; then
    echo "SUCCESS: Result equals expected result" > /dev/tty
    echo "SUCCESS: Result equals expected result"
else
    echo "FAILURE: Result differs from expected result" > /dev/tty
    echo "FAILURE: Result differs from expected result"
fi
```

## F Microsoft Word についてのトラブルシューティング問題

行を指定してブレイクポイントを設定する

**VDMTools** に含まれるテンプレートファイル `VDM.dot` には、関数や操作内部の行にブレイクポイントを設定することを可能にする特別なマクロが入っている。しかしこの機能は VDM-SL Toolbox のバージョン v8.3.2 以降でないと入っていない。このマクロはカーソルを現在のプロジェクトに含まれる RTF ファイルのブレイクポイントを設定したい行にカーソルをあて `Control-Alt-スペースキー` を押すことで有効になる。

1. このマクロを有効にする前に **処理系を初期化** ボタンを押してツールボックスのインタープリタを初期化していなくてはならない
2. `VDM.dot` ファイルをテンプレートディレクトリに移すのを忘れないように。通常テンプレートディレクトリは

`C:\Program Files\Microsoft Office\Templates`

3. 使用中の文書にはおそらくテンプレートとしての `VDM.dot` がないと思われる。これは Microsoft Word 内の **ファイル->プロパティ機能** を使ってチェックすることができる。もし `VDM.dot` がなかったらマクロは代わりに使用したいテンプレートにコピーされることを確認しなくてはならない。

テストカバレッジファイルの発見

テストカバレッジファイルが正しく `include` されていない場合は、おそらく原因は Samba 経由でウィンドウズからアクセスされる Unix サーバ上におかれているファイルを使用しているためである。この場合、ファイル名に使用されている大文字・小文字の区別をつけることが現状正しくできないため、すべてのファイル名を小文字にするべきである。



## 索引

- .vdmde ファイル, 45
- .emacs ファイル, 102
- \$\$, 59, 60, 67, 68
- vdmde, 102
- vdmde
  - コマンドラインオプション, 65
  
- backtrace コマンド, 67
- break コマンド, 58, 59, 67
  
- C++コード生成, 34
- C++ファイル, 34, 78
- codegen コマンド, 78
- condition コマンド, 59
- cont コマンド, 67
- cquit コマンド, 44
- curmod コマンド, 58, 59, 67
  
- debug コマンド, 58, 59, 67
- def 型, 50, 51, 53
- delete コマンド, 58, 60, 67
- dir コマンド, 44
- disable コマンド, 60, 68
  
- Emacs インターフェース, 102
- enable コマンド, 60, 68
- encode コマンド, 44
  
- finish コマンド, 68
- first コマンド, 48, 54
- functions コマンド, 43
  
- GUI, 35
  - スタート, 7, 35
- help コマンド, 44
  
- info コマンド, 44
- init コマンド, 60, 68
  
- last コマンド, 49, 54
- latex コマンド, 76
  
- modules コマンド, 43
  
- next コマンド, 49, 54
  
- operations コマンド, 43
- Options
  - C++ Code Generator, 78
  - Editor and Print, 101
  - Interpreter, 25
  - Pretty Printer, 75
  - Type Checker, 52
- Options:Interpreter, 62
- option コマンド, 69
  
- popd コマンド, 68
- pop コマンド, 58, 61, 68, 70
- pos 型, 50, 51, 77
- previous コマンド, 49, 54
- print コマンド, 22, 58, 60, 68
- push コマンド, 58, 61, 70
- pwd コマンド, 32, 44
  
- quit コマンド, 45
  
- read コマンド, 48
- remove コマンド, 69
- rtinfo コマンド, 31, 82
  
- script コマンド, 44
- set full コマンド, 55
- set コマンド, 55

- singlestep コマンド, 69
- stack コマンド, 58, 61, 70
- states コマンド, 43
- stepin コマンド, 69
- step コマンド, 69
- system コマンド, 44
  
- tcov read コマンド, 61, 70
- tcov reset コマンド, 31, 61, 70
- tcov write コマンド, 31, 61, 70
- tcov コマンド, 61, 70
- typecheck コマンド, 54
- types コマンド, 43
  
- unset full コマンド, 55
- unset コマンド, 70
  
- values コマンド, 44
- VDM.dot ファイル, 6, 59, 73, 106
- vdmgde コマンド, 7, 35
- vdmsl.sty ファイル, 73
- VDM 標準, 1
  
- 陰関数, ⇒
  - 関数,
  - 陰
- インタプリタ, 19
  - コマンド, 56
  - 初期化, 19, 56, 58
  - 停止, 60, 61
  - 呼び出し可能関数, 20
- インタプリタウィンドウ, 9, 19, 56
- インプット
  - 一般, 1
  - 作成, 6
  - フォーマット, 1
  
- エラーリスト, 9, 12, 46, 51
  
- オンラインヘルプ, 7
  
- 外部エディタ, 14, 41
- 型エラー, 51
- 型チェック, 15, 50
  - 動的, 24, 65
- 型の正しさ, ⇒
  - pos 型,
  - def 型
- 関数
  - 陰, 20, 56
  - 不変条件, 20
  
- 構文エラー, 12, 46, 102
  - 修正, 12
  - フォーマット, 47
- 構文チェック機能, 14
- 構文チェック, 11
- 構文の色付け, 99
- コード生成, ⇒
  - C++コード生成,
  - Java コード生成
- コールスタック, 22, 23, 56
- コマンドラインインターフェース, 42
  - C++コード生成, 77
  - インタプリタ, 65
  - 開始, 42
  - 型チェック機能, 53
  - 構文チェック機能, 47
  - 清書, 74
  - デバッグ, 65
  - ファイルの初期化, 45
  
- 識別子検索ウィンドウ, 9
- 事後条件チェック, 25, 63, 65
- 事前条件チェック, 25, 62, 65
- 実行不可能な構成要素, 20
- 自動構文チェック, 99

証明課題, 26  
証明課題ウインドウ, 9  
清書  
    L<sup>A</sup>T<sub>E</sub>X 文書のセットアップ, 95  
    インデックス, 97  
    コメント, 98  
    例, 95  
清書機能, 32  
ソースウインドウ, 9  
適格性, ⇒  
    pos 型,  
    def 型  
テスト, ⇒  
    テストカバレッジ  
テストカバレッジ, 80, 83  
    環境, 86  
    テストスイート, 31, 80  
    ファイル, 31, 75, 80  
    例, 83  
テストスイート, ⇒  
    テストカバレッジ,  
    テストスイート  
動的型チェック, ⇒  
    型チェック,  
    動的  
動的リンク機能, 34  
不変条件チェック, 25, 65  
ブレイクポイント, 21, 59, 67  
    Microsoft Word の設定, 59  
    削除, 23, 56, 58, 60, 69  
    設定, 21, 58, 59, 67  
    無効, 24, 56, 60, 68  
    無視, 22, 58, 66  
    有効, 24, 56, 60, 68  
プロジェクト, 36  
    ファイル追加, 10  
プロジェクトビュー, 9  
ヘルプ, 42  
マネージャー, 9, 36  
モジュールビュー, 9  
呼び出し不可能関数, ⇒  
    関数,  
    不変条件  
ライセンス, 34, 77  
ログウインドウ, 9