

VDMTools

VDMTools ユーザマニュアル
(VDM++)
ver.1.0



How to contact:

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

VDMTools ユーザマニュアル (VDM++) 1.0

— Revised for VDMTools v9.0.6

© COPYRIGHT 2016 by Kyushu University

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

This document is subject to change without notice

目 次

1	はじめに	1
2	VDMTools 概略	4
3	VDMTools ガイドツアー	7
3.1	VDMTools への入力物を作成する	7
3.2	GUI で VDM++ を始める	8
3.3	オンラインヘルプ	8
3.4	メニュー、ツールバー、サブウィンドウ	9
3.5	プロジェクトを作成する	11
3.6	VDM 仕様書の構文チェック	12
3.6.1	仕様書の解析	12
3.6.2	構文エラーの修正	13
3.7	VDM 仕様の型チェック	15
3.8	仕様書の検証	19
3.8.1	インタープリタを使用した式の評価	20
3.8.2	ブレイクポイントの設定	23
3.8.3	動的型チェック	26
3.8.4	証明課題のチェック	27
3.8.5	マルチスレッド・モデル	32
3.9	体系的テスト	32
3.10	清書機能	34
3.11	コード生成	35
3.12	VDMTools API	35
3.13	VDMTools の終了	36
4	VDMTools リファレンスマニュアル	37
4.1	GUI 全般	37
4.1.1	プロジェクト・ハンドリング	38
4.1.2	仕様書の操作	41
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	42
4.1.4	ファイルの編集	45
4.1.5	インタープリタを使う	45
4.1.6	オンラインヘルプ	45

4.2	コマンドラインインターフェース全般	46
4.2.1	ファイルの初期化	48
4.3	構文チェック機能	49
4.3.1	GUI	49
4.3.2	構文エラーのフォーマット	50
4.3.3	コマンドラインインターフェース	50
4.3.4	Emacs インターフェース	51
4.4	型チェック機能	53
4.4.1	GUI	54
4.4.2	エラーおよびワーニングのフォーマット	55
4.4.3	コマンドラインインターフェース	56
4.4.4	Emacs インターフェース	57
4.5	インタープリタとデバッガ	59
4.5.1	GUI	59
4.5.2	スタンダードライブラリ	67
4.5.3	コマンドラインインターフェース	69
4.5.4	Emacs インターフェース	70
4.5.5	スレッドのスケジューリング	76
4.6	証明課題生成機能	77
4.7	清書機能	79
4.7.1	GUI	80
4.7.2	コマンドラインインターフェース	80
4.7.3	Emacs インターフェース	82
4.8	VDM++から C++コード生成	83
4.8.1	GUI	83
4.8.2	コマンドラインインターフェース	83
4.8.3	Emacs インターフェース	84
4.9	VDM++から Java へのコード生成	85
4.9.1	GUI	85
4.9.2	コマンドラインインターフェース	86
4.9.3	Emacs インターフェース	87
4.10	VDM モデルの体系的テスト	88
4.10.1	テストカバレッジファイルの準備	89
4.10.2	テストカバレッジファイルの更新	89
4.10.3	テストカバレッジの統計データ作成	90
4.10.4	L ^A T _E X を使ったテストカバレッジ例	91

用語集	98
A VDM 技術の情報源	100
B VDM++ と L ^A T _E X の結合	103
B.1 仕様書ファイルのフォーマット	103
B.2 L ^A T _E X 文書のセットアップ	103
C VDMTools 環境の設定	107
C.1 インターフェースオプション	107
C.2 多言語サポート	108
C.3 UML リンクオプション	110
D Emacs インターフェース	111
E Sort 例題向けテストスクリプト	112
E.1 Windows/DOS プラットフォーム	112
E.2 UNIX プラットフォーム	113
F Microsoft Word についてのトラブルシューティング問題	115
G プライオリティファイルのフォーマット	116
索引	117

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 [13] あるいはオブジェクト指向形式仕様言語 VDM++ [4] [12] によって表現される。このマニュアルでは VDM++ ツールボックスに関して記述するが、このツールは実装に先立ち、VDM++ で表現されたモデルの自動チェックと検証をするためのツールである。その範囲は古くからある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する強力なインタプリタにまで及び、実行中は自動的に整合性チェックを実行する。実行しやすいので分析・設計の早期からテスト技術の利用が可能であり、確立したソフトウェアエンジニアリングの慣習に沿ってテスト全体を実行することが出来る。そのうえ、このインタプリタではブレイクポイントの設定、命令文のステップ実行、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++ 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++ 言語には別に言語マニュアルがある。[4]。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

本 Toolbox は MS Word または \LaTeX のどちらかの文書を組み込んだ VDM++ の仕様をサポートしているので、仕様書の分析をする際に特別なファイルを作成することなく行うことができる。ドキュメントを使ってシステムのモデルを組み合わせるよい方法としてこれらの 2 つのアプローチのうちどちらかを使用することを推奨する。作成された仕様書のバージョンと実務の不整合を避けるために、仕様の形式はどちらかに統一したほうがよい。Word も \LaTeX も好みでなく、テキストエディタがお好みの場合はもちろん ASCII のみで書かれたテキストを仕様書として書くことも出来る。

本 Toolbox は異なる言語やスクリプトも入力文書としてサポートしている。付録 C.2 で異なるスクリプトで Toolbox がどのように構成されるか説明している。

VDM++ の仕様を書くために MS Word を利用するならば、VDM++ による仕様記述を含む文書を *Rich Text Format*(RTF) フォーマットで保存しなくてはならない。Toolbox 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、Toolbox に含まれるファイルを使った例を見ることができる。ファイル名が拡張子 .rtf のファイルがサンプルファイルで、これがリッチテキストフォーマットであることを示している。

このマニュアルでは、通常 Toolbox の機能を Word か RTF で紹介する。仕様を書くのに \LaTeX を使う場合、Toolbox 上、入力文書が付録 B に書かれているフォーマットとスタイルを使った、 \LaTeX のコマンドと VDM 仕様とが混ざり合ったものと想定されていることに注意してほしい。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 “.vpp” 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.vpp 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。

ASCII 形式のテキストファイルで仕様を記述した場合、テキストの説明文を (VDM++ の) 仕様書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++ 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 “.vpp” 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 [3]。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 \LaTeX 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。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そして付録 G ではインタープリタで使うファイルの優先順位の定義用のフォーマットが記述されている。

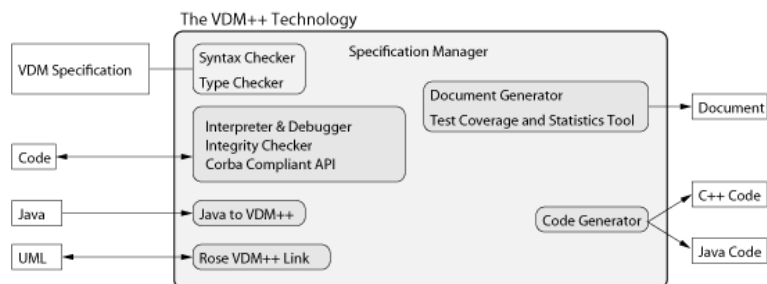


図 1: VDMTools 概略

2 VDMTools 概略

VDM++ 仕様はシステムの特性を精巧な方法で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いくつかのファイルに分散される。図 1 は Toolbox の機能の概要とその付属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 いくつかのファイルに分かれている仕様書におけるクラスの状態トラックを保持する。

構文チェック機能: VDM++ 言語の定義的に VDM++ の仕様書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 内の他ツールでも大丈夫である。

型チェック機能 強力な型推測メカニズムを持ち、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++ で実行可能な構成物すべてを実行する。その範囲は集合内包や列の列挙など単純な構成子からより進化した構成物（例外の扱い、ラムダ表現、ルーズな表現やパターンマッチング、マルチスレッドモデルまでも）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、仕様書の小（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、実行可能な使用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仕事をする上で必要不可欠な機能である。VDM++ デバッガは通常のプログラミング言語向けのデバッガのような機能をサポートするが、これに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が含まれる。

証明課題生成機能 証明課題生成機能は仕様書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、VDM++ ツールボックスの静的チェックの可能性を拡張したものである。データ型の不変条件、事前条件、事後条件、シーケンス境界や map ドメインの違反チェックが含まれる。証明課題は VDM++ の式では true と評価され、表される。(- もし変わりに false と評価されていたら、それは仕様書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未定義の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自動的に記録され、特定できる箇所まで戻って表現されるが、これは仕様書の一部において頻繁に評価されるところと全くカバーされていないところを示すものである。テストカバレッジ情報は使用されている形式 (Word または L^AT_EX) で書かれたソースファイル文書に直接表示される。

自動コード生成 Toolbox は VDM++ の仕様書からの C++ と Java の自動コード生成をサポートしているが、この機能により仕様と実装の間に一貫性が出る。コード生成は VDM++ の構成物の 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の、ユーザによる定義コードを含む機能も併せ持つ。いったん仕様がテストされれば、コード生成を自動的に迅速な実装を実現するために適用することができる。C++ のコード生成機能の使い方は本ドキュメント [7] に記載されている。Java のコード生成機能の使い方は本ドキュメントに記載されている。[7] そして Java コード生成の使い方は本ドキュメント [8] に記載されている。

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 はまたグラフィカルなフロントエンドやツールボックスを制御するレガシーコードなど、どんなコードでも可能にする。

Rose-VDM++ リンク Rose-VDM++ リンク は UML と VDM++ を結びつける。Toolbox と Rational Rose を緊密に結びつける双方向の翻訳を提供する。ゆえにこのリンクは UML と VDM++ のラウンドトリップエンジニアリングをサポートし、形式的な記述が機能の詳細な振る舞いを記述するの

に使われる一方で、グラフィカルな記述が構造的でダイアグラムを利用したモデルの概略を提供するのに使われる。Rose-VDM++ リンクについてはドキュメント [\[2\]](#) に記述されている。

Java から VDM++ への変換ツール この機能は現存する Java アプリケーションを VDM++ にリバースエンジニアリングするものである。アプリケーションの分析が VDM++ レベルで実行され、新たな機能が特定される。最終的に、新たな仕様書が Java から翻訳される。Java から VDM++ への変換ツールの使い方は [\[1\]](#) に記述されている。

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」を記述する。VDM のシステムモデリングを新たに学ぶ人は、“ソフトウェア開発のモデル化技法” [11], J. フィッツジェラルド、P.G. ラルセン著、または、“*Validated Designs for Object-oriented Systems*” [12] 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は、Toolbox を使うことによって生み出されるであろう VDM 仕様を使って作られたさまざまな例を含む。[11] is using the ISO standard VDM-SL notation whereas [12] is using the object-oriented extension called VDM++. これらの一般的な概念を知っているが、VDM++ でオブジェクト指向向けに拡張された部分に詳しくない人には、VDM++ の言語リファレンスマニュアルである “*The VDM++ Specification Language*” [4] を一読することをお勧めする。

3.1 VDMTools への入力物を作成する

Toolbox を使用するためには、VDM++ 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^AT_EX 文書を使いたい場合は、付録 B¹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 vpphome/examples/sort/MergeSort.rtf ファイルを開く。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++ 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 (通常は C:\Program Files\Microsoft Office\Templates) にコピーされるので、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

¹ASCII のみの VDM++ を使用することも可能

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。

MergeSort.rtf ファイルの最後を見てみよう。クラス名 MergeSort が VDM_TC_TABLE の形式で書かれてい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++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 [6] 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、vpphone/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 vppgde 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。

ツールについて (?): Toolbox のバージョン番号を表示する。

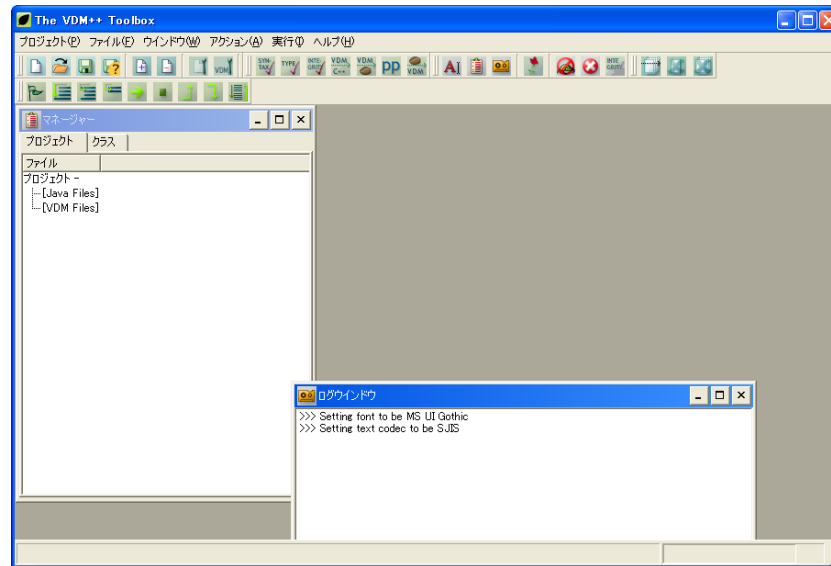



図 2: スタートアップ画面

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

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列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++ の仕様書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下にある画面に表示されているウインドウのコントロールを決定する。それぞれのメニュー項目は実際のウインドウの開く/閉じるとトグルしている。

構文や型チェック、証明課題の生成、コード生成、Java から VDM++ へのリバースエンジニアリング、清書など仕様書に適用されるさまざまなアクションを提供する。

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)

以下ではこのメニューの 6 つ²あるツールバーの項目について同様のアクションを提供するものを示す。

最終的に、メインウインドウ下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状態を表示する。以下 2 つのパートからなる。

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クラス (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。

クラスビュー VDM ビューと Java ビューの両方があり、プロジェクトに含まれる個々の VDM++ クラス・Java ファイルをそれぞれ表示する。

ソースウインドウ 発見されたエラー一覧内で現在選択中のエラーの元仕様の一部を表示する。

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レポート

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

Toolbox の起動時には、マネージャーとログウインドウのみが開いている。

²Toolbox が開始された時、1 番目と 3 番目と 4 番目のメニューに相当するものだけが開く。残りの 3 つはその上にアイコン化されて表示される

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選択したファイルを現在のプロジェクトに追加を選択するか (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³。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。

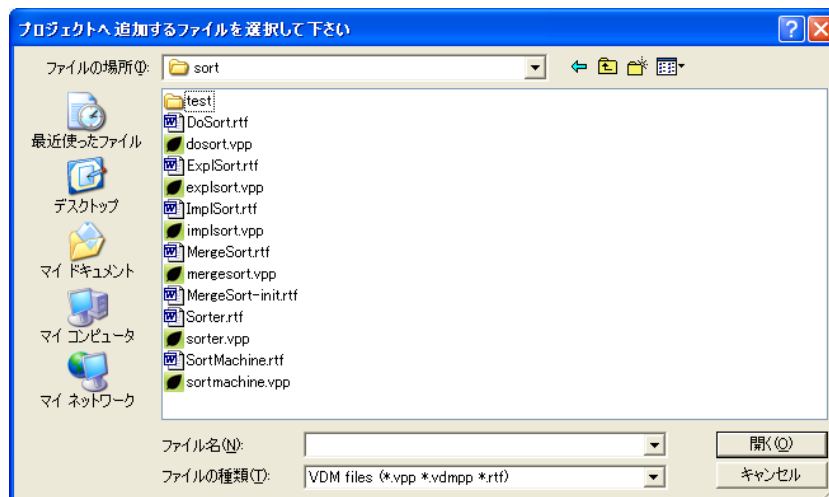


図 3: プロジェクトにファイルを追加する

MergeSort.rtf を除く 6 つの .rtf ファイルを選択して (Ctrl + マウスの左ボタンクリックを順番に繰り返す) 「開く」ボタンを押す。そうすると選択したファイルがプロジェクトに追加される。単純にマウスの左ボタンをダブルクリックするだけでもファイル 1 つであれば追加することができる (ただし、ダイアログが閉じてしまうので複数のファイルを追加することはできない)。また Shift キーを押しながら最初のファイルと最後のファイルを選択してもよい。MergeSort-init.rtf ファイルにはこのガイドツアーで見せる目的で、いくつかのエラーが入っていることに注意してほしい。

6 つの .rtf ファイル は下記図 4 に示すメインの Toolbox 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。

³このガイドツアーではツールバーのボタンを経由してのやりとりを中心に話を進めていることを忘れないでほしい。もしお望みならばもちろん、いつでも同等のメニュー項目を使うことができる




図 4: ファイル追加後のメインウインドウ

3.6 VDM 仕様書の構文チェック

プロジェクトを作成したら、すべての クラスが、VDM++ 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書の構文が正しいかどうかチェックする。元ファイルを変更したら、その他のツールが変更が生じていることに注意する前に、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 仕様書の解析

マネージャのプロジェクトビューで 6 つの .rtf ファイルを選択したら、(アクション) ツールバーの  (構文チェック) を選択すると構文チェック機能が起動する。(「デフォルト」フォルダを含むレベルを選択すること、および構文チェック

の操作を適用することは同じ効果をもたらす - つまり、これはそのフォルダ内のファイルそれぞれに対する操作として適用される) ここで、ログウインドウが自動的に開き (すでに開いていない場合)、“Parsing “.../DoSort.rtf” ...” 等 のメッセージを表示することに注目してほしい。 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わざと二つ構文エラーを入れてある。

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ワーニングの生じた箇所のリスト (ファイル名、行数、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。

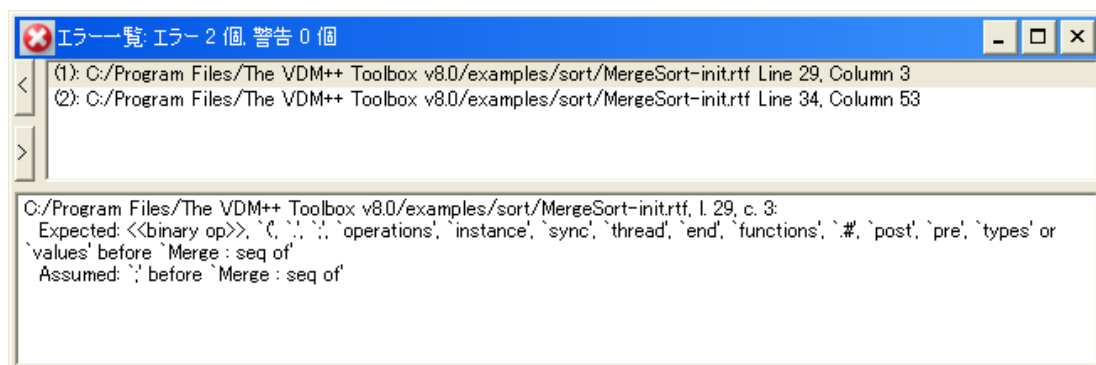


図 5: エラー一覧

ソースウインドウは、仕様書の現在選択中のエラーが発見された部分を表示するが、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。

最初のエラーメッセージは下記のとおり：

```
C:\vpphome\examples\sort\MergeSort-init.rtf, l. 29, c. 3:
Expected: <<binary op>>, '(', '.', ';', 'operations', 'instance',
          'sync', 'thread', 'end', 'functions', '.#', 'post'
```

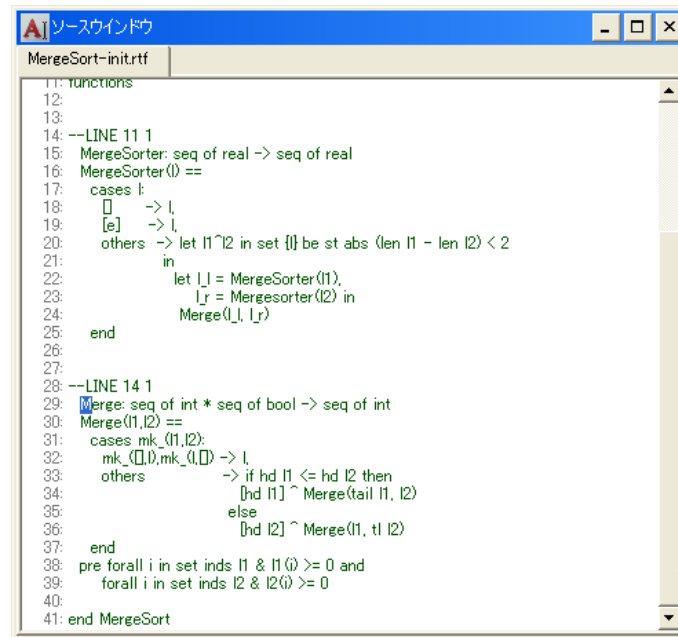


図 6: 最初のエラーがソースウインドウに表示された所


‘pre’, ‘types’ or ‘values’ before ‘Merge : seq of’
Assumed: ‘;’ before ‘Merge : seq of’


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するためにはどうすべきだったのか予測を行う。

この例では、エラーメッセージが構文チェックを続けるためには Merge 関数の前に ‘;’ が見つからないと予測される、ということを伝えている。[\[4\]](#) によりこの予測が正しいことが分かる。 - 2つの関数定義はデリミタ ‘;’ で区切らなくてはならない。ゆえにこのエラーはファイルエディタを使い、MergeSort 関数の定義の最後に ‘;’ を追加することで修正される。

(構文チェック機能を使っても元ファイルは修正されないことに注意してほしい。想定されることが複数ある場合、元ファイルの修正はユーザの手作業でなされるべきだからである)

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) メインウインドウで MergeSort-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。外部エディタの起動時に複数のファイルが選択されていた場合、この方法だと選択したファイルそれぞれが外部エディタで表示される。

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:\vpphome\examples\sort\MergeSort-init.rtf, l. 34, c. 53:
```

```
Expected: <<binary op>>, '(', ')', ',', '.', '[', '~', '.#',  
          '' or ', ... ,' before 'l1 , l2 )'
```

```
Assumed: '*' before 'l1 , l2 )'
```


これは l1, l2 の前に構文エラーが起こっていて、構文エラーを回避するためには記号 '*' が 'tail' ('tail' は識別子の名前であると予測) と 'l1' の間に必要なのではないかと予測していることを伝えている。この場合、予測は間違っており実際は列から先頭を除いたものを返す 'tl' 演算子を 'tail' と書き間違えたことによるエラーである。ファイルエディタを使って修正ができる。

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。今度はファイルは構文的に正しいはずであり、マネージャー の(クラスビュー) 内にある VDM ビュー にウインドウが移り、仕様書にある 6 つのクラス (DoSort, ExplSort, ImplSort, MergeSort, Sorter, SortMachine) の状態が確認できるだろう。

そして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、仕様書の型チェックやコード生成、清書などがまだ 1 度も行われていないことを意味している。

構文チェックが成功したので、ファイルは VDM ビュー から直接選択して次の処理に進めることができる。

3.7 VDM 仕様の型チェック

仕様書がいったん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。

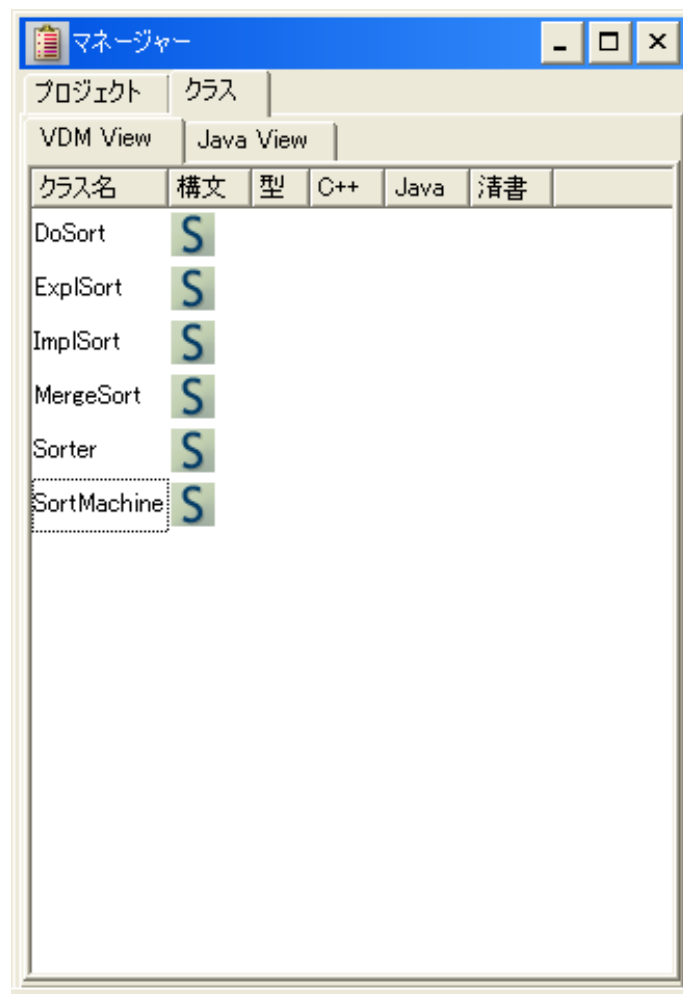


図 7: VDM ビュー

VDM ビュー で 6 つのクラスすべて を選択し、型チェックを行う。型チェックが終わると、Toolbox は記号 **T** と **✗** (**T** には赤いラインもつく) を使って各々のクラスに関して型チェックが通ったか通らないかを示し、ビューの状態表示を更新する。

この例では、実は MergeSort クラスが 型チェックに失敗するようになっている。エラーが 3 つ⁴とワーニングが 1 つ発生するがこれらは以前と同様、エラー一覧に表示される。

⁴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

最初のエラーは図 8 でも示しているが、関数名 `Mergesorter` の `s` が小文字であることが原因である。(ソースウインドウ の表示については図 9 を参照のこと) この関数名は `MergeSorter` でなくてはならない。

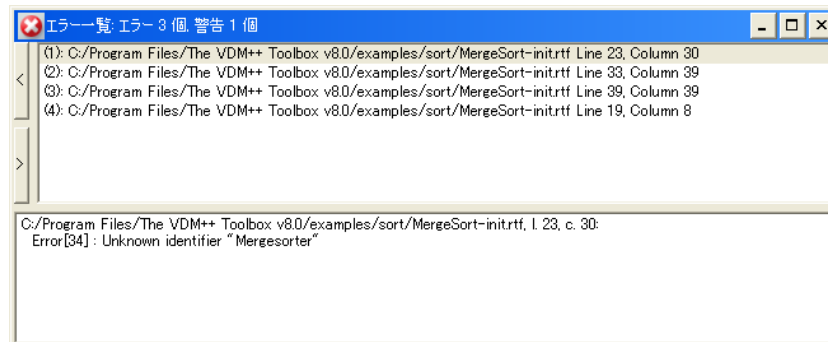


図 8: 型チェック後最初のエラー表示

2 番目のエラーは図 10 にも示すとおり、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、実際の引数は (エラーメッセージ中では `act:` と表示) `bool` 型で宣言されているが予想される引数は (エラーメッセージ中では `exp:` と表示) `real` 型である。(`real` 型は最も一般的な数値型の型である) エラーの原因を特定しようとする場合のこのような情報は、様々である。

このエラーは実際には `Merge` 関数のシグネチャ中で、`seq of int` であるべきものが `seq of bool` であったせいである。3 つ目のエラーも同じ原因であり、2 番目のものとも関係しているため、2 番目のエラーが修正されると消える。このため、最初の 2 つのエラーを修正して 再度仕様書の構文チェック、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、VDM 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。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。

型チェック処理はワーニングを返してきたとしても成功であることに注意。これはワーニングは一般的に実際のエラーというよりは仕様の中の余計なものを表しているためである。例えば、ある特定の `bool` 式がいつも `false` であると予想され

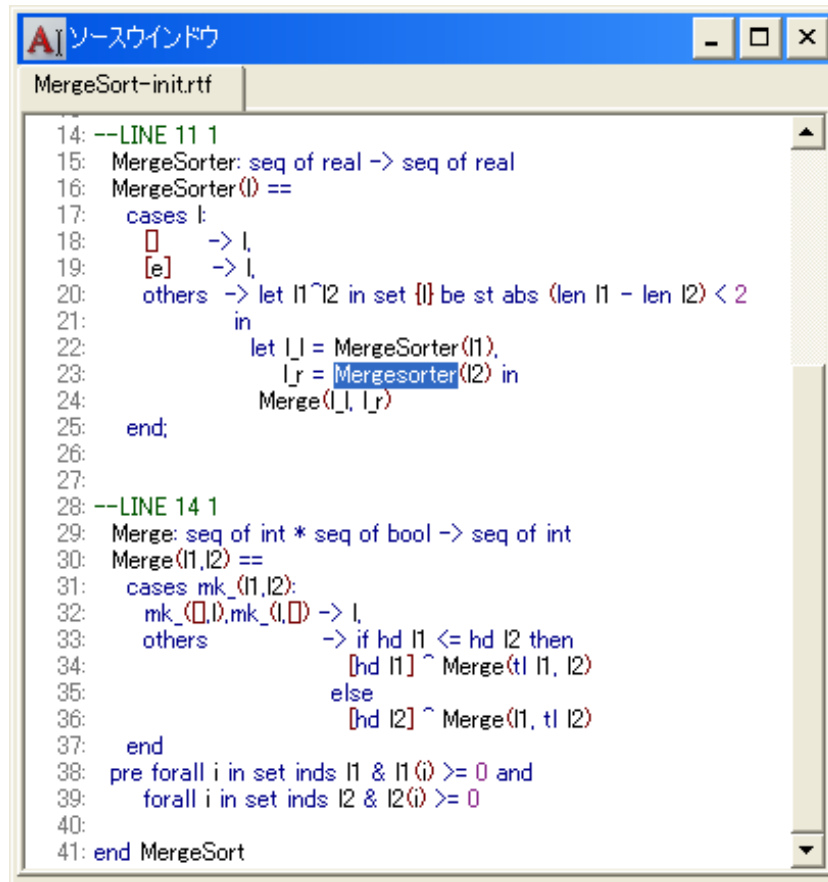


図 9: 型エラー時のソースウインドウ

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、このように余計なものは実際にはエラーの原因となることもある（ワーニングの数を増加させているものの中には、表現や記述のタイプミスもあるかもしれない。）そのためワーニングをチェックすることは実際取るに足らないことであることを確認するためにも有用である。例で言えばこのワーニングはMergeSort 関数内の case 文の 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言っている。これは現実にはエラーではなく、仕様書は正しく意味をなしている。ただしワーニング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。

仕様書は型チェックをパスしたが、これは（仕様書が）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（プログラミング言語に適した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

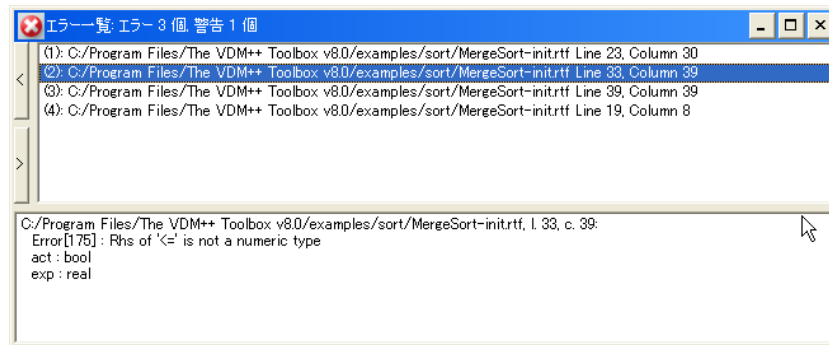


図 10: 2 番目の型チェックエラー表示

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潜在的な元となる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書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。そのため、もし報告された潜在的なエラーが起こり得ないと自分自身が納得するなら、ランタイムエラーは起こらないだろう。これについてのより詳しい情報は、このマニュアルのセクション 4.4 を参照のこと。

3.8 仕様書の検証


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だったり、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だったり、次の詳細設計や実装の基礎に役立てるためだったりする。その目的は何であれ、単に仕様書の構文や型が正しいだけでは不十分で、たとえ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書があれば、広範囲の検証技術が使える。仕様書は詳細に調べることができ、テストもできる。仕様書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（仕様書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書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命令文を評価しデバッグすることができる。これらはかなり複雑で、Toolbox に読み込まれている仕様書で定義されているアプリケーションの関数や操作、変数の使用を含んでいる。デバッガを使うと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。

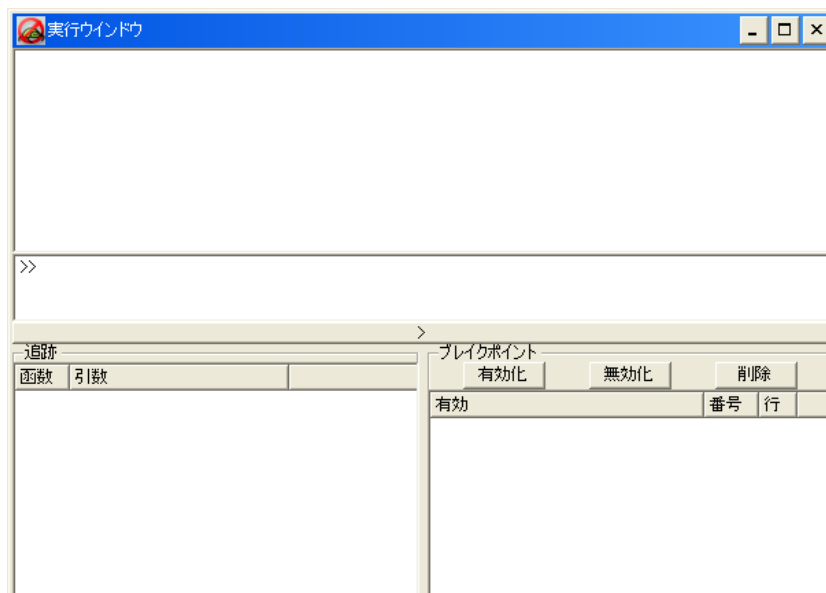



図 11: 実行ウインドウ

ツールの画面 2 つの上部には、それぞれ Response と Dialog 画面がある: Dialog 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 Response 画面に表示される。VDM++ の表現を評価するためには、Dialog 画面からコマンドラインで直接タイプする。

Dialog 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 ([4]) で説明する。

仕様書のファイル から読み込んだ VDM++ の構造を参照することもできるが、それをやる前に  (初期化) ボタン を押してインタプリタの初期化 をしなくてはならない。初期化中、定数は評価され インスタンス変数 は初期化される。初期化後は、関数、操作、インスタンス変数、値、型など仕様書 のクラス内で定義されているものなら何でも参照できる。

どの関数が MergeSort クラスから呼び出し可能かを見るために、Dialog 画面のプロンプトで functions MergeSort とタイプする。これで MergeSort クラスの呼び出し可能な関数のリストが表示される。リストからは関数 pre_Merge のような、付随する事前条件がある場合に生成される事前条件関数を呼び出すことが可能であることが見て取れる。

関数や操作、インスタンス変数の精査、値のアプリケーションはオブジェクトを通じてのみ実行される。オブジェクトが生成されているために、その後インタプリタ内でそれらを利用する事ができる。

以下の 2 つのコマンドは、Dialog 画面で使うと、ms という名の MergeSort クラスのオブジェクトを生成し、引数 [3.1415, -56, 34-12, 0] で ms の Sort 関数を呼び出し、結果を表示する。

```
create ms := new MergeSort()
print ms.Sort([ 3.1415, -56, 34-12, 0 ])
```

また、オブジェクトはインタプリタ中の記述の評価中でもローカルに生成することが可能である。例えば、前の例の ms オブジェクトのようにオブジェクト名をつけなくても MergeSort.Sort は以下のコマンドで実行できる

```
print new MergeSort().Sort([ 3.1415, -56, 34-12, 0 ])
```

この例では、評価の最中ローカルにオブジェクトが生成されるが Sort 操作のテスト中だけオブジェクトが存在することとなる。

インスタンス変数を調べるため、まず SortMachine クラスのオブジェクトを生成する。

```
create sm := new SortMachine()
```

SortMachine クラスは `srt` という名前のインスタンス変数を持ち、これは `Sorter` オブジェクトの参照である。初期値は `srt` には `MergeSort` オブジェクトをさしている。まず `srt` の値を見てオブジェクトのさす `Sort` 関数と呼んでみる。

```
print sm.srt
print sm.srt.Sort([ 3.1415, -56, 34-12, 0 ])
```

図 12 は上記の評価の結果を表している。

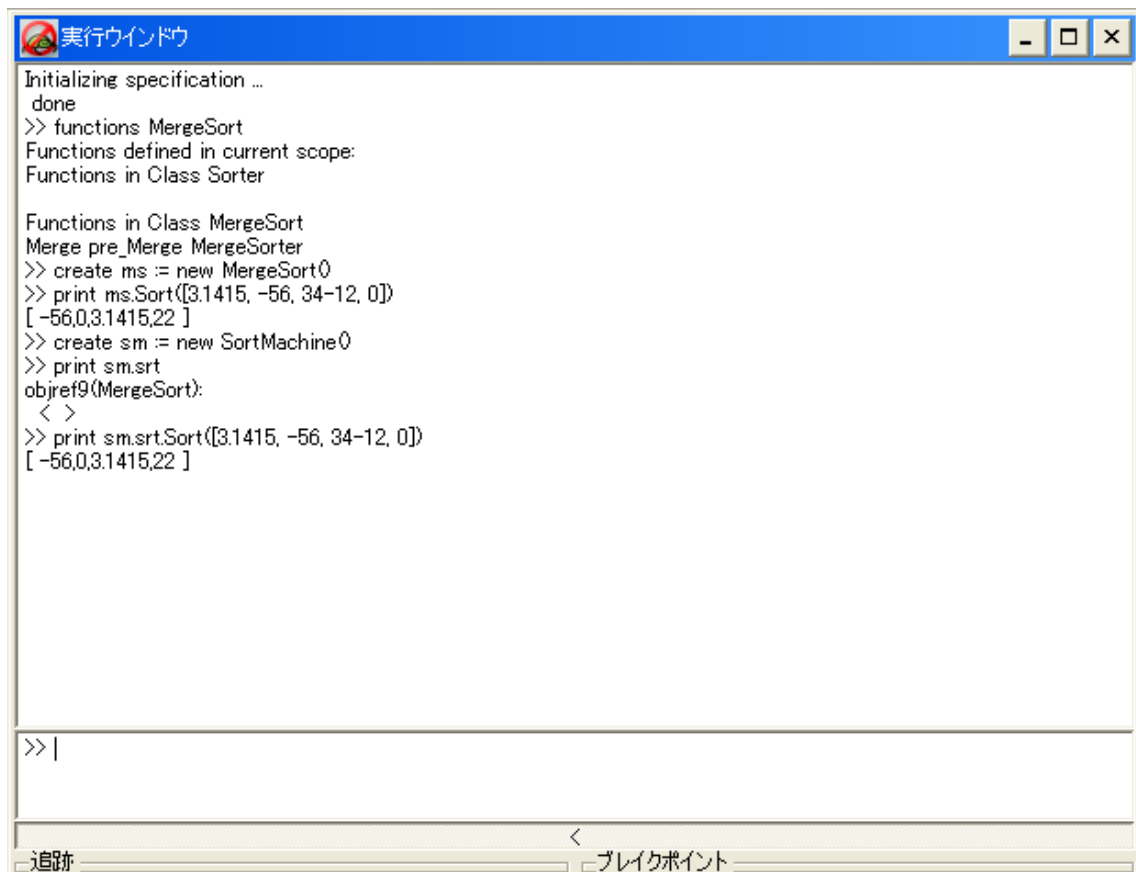


図 12: 式の評価

VDM++ の構成物すべてが実行可能なわけではないので、実行可能でないものはインタープリタを使った評価が出来ないことに注意。例えば、陰関数 `ImplSort`、`ImplSorter`

を数字の引数を使って呼び出そうとした場合、インタプリタは評価中に実行不可能な構成物に出くわしたとエラーを返す。図 12 を参照

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。

関数 クラス MergeSorter の MergeSort に、break MergeSort 'MergeSorter とタイプすることで ブレイクポイントを設定することができる。

(ブレイクポイントを設定するとき、定義されたクラス名と関数名の両方が必要) ブレイクポイントの箇所は実行ウインドウの右下、ほかの設定されたブレイクポイント (この例では1つしかブレイクポイントを設定していないので、1つである) とブレイクポイントが有効であることを示す記号 ☒ と一緒にブレイクポイント画面に表示される。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強制的にインタプリタがストップすることである。

以下のようにタイプして Sort 関数を呼び出してみる。

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


するとインタプリタは Sort 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、Sort 関数を含む仕様書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 Sort 関数の最初のところ) にカーソルが当たっている。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ている。



ここで、Sort 関数のパラメータの値を print コマンドを使うか (Dialog 画面で print 1 とタイプする)、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 '...' 部分でマウスの左ボタンをクリックすると詳細に見ることが出来るはずだ。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

適切な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、マウスの左または真ん中のボタンをソースウインドウのファイル中、設定したい位置でダブルクリックするとブレイクポイントが設定できる。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。それではソースファイルを使って、MergeSort クラスの Merge 関数内にブレイクポイントを設定してみよう。上記のようになる。

インタプリタに戻って (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 Sort の再帰的な呼び出しによって、インタ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。 (現在の関数を呼び出ししている関数に戻るところまで実行) 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。 (現在の関数が下位の関数を呼び出すところまで実行) ボタンは追跡を元に戻すときに使う。


 (関数内をステップ実行) 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カーソルがソースウインドウ で、実行中評価の現在の箇所の変化をマークするのにどのように動くかを確認してほしい。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めたすべての変数にアクセス可能になり、例えば print コマンドを使うなどして値の中身を見ることがもできる。

図 13 にデバッグの例を示す。

ブレイクポイントはデバッグ実行中でも削除することができる。実行ウインドウの Dialog 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) 実行ウイ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

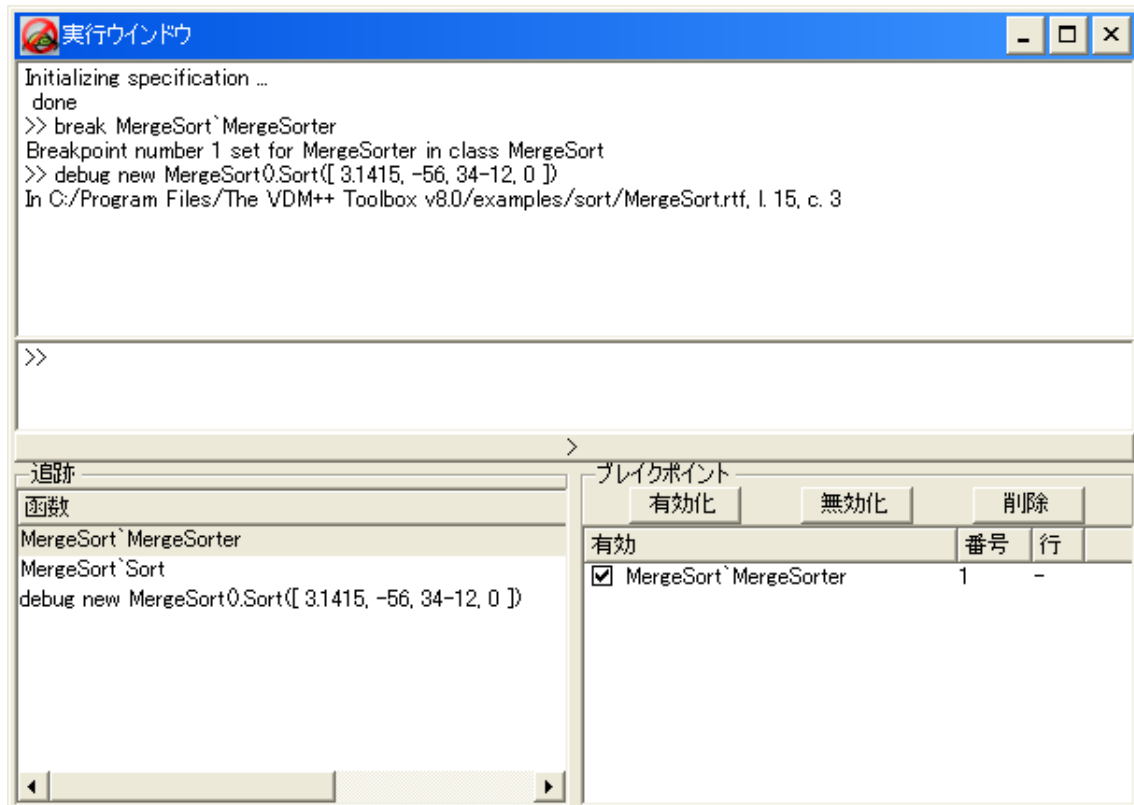



図 13: 仕様のデバッグ

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`MergeSorter 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書のエラーを見つけなくても、単に静的な型の情報（関数宣言のみの型チェックを分析した場合など）を分析しただけですべての型エラーを見つけることが不可能なので、依然として型エラーが存在する可能性がある。例えばある関数が引数に `int` 型の値をとると宣言されたが、式が `real` 型の数字だった場合静的型エラーにはならない。これは `int` 型が `real` 型のサブタイプであるために、アプリケーションは実際にはその関数が `real` 型のパラメータで実行時に呼ばれても、正しいとしてしまう。

仕様レベルでこの手の型エラーを発見するために、インタプリタが評価実行中に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。

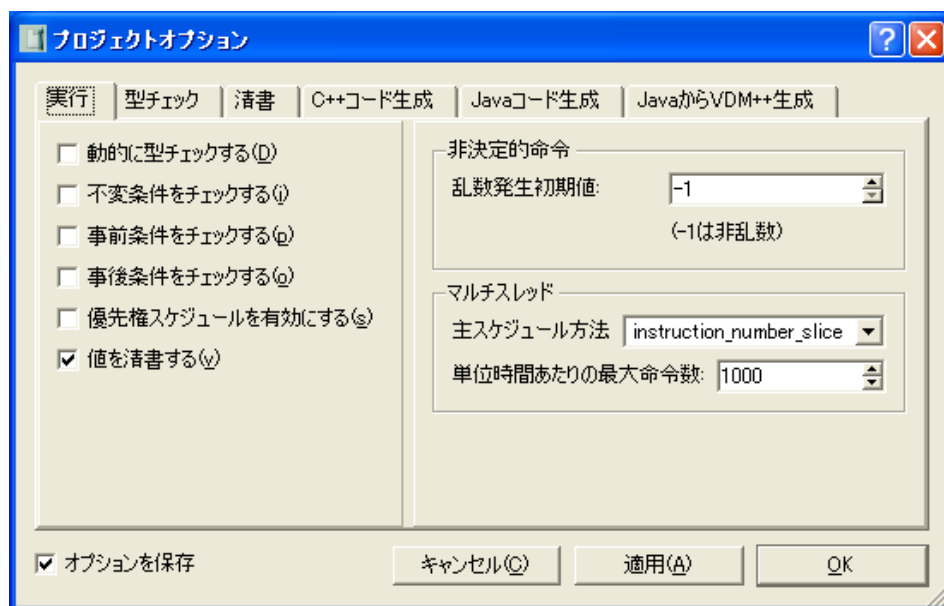


図 14: インタプリタオプションの設定

「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 new MergeSort().Sort([ 3.1415, -56, 34-12, 0 ])
```

のところでインタ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 `MergeSort` の `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、仕様書のエラーの可能性を明らかにしている - `MergeSorter` クラスの `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、同じクラスの `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 `MergeSort` の `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。

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がチェックする。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書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⁵がとりうるすべての値の定量化を含む VDM++ の記述として表現さ

⁵ 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書の精査

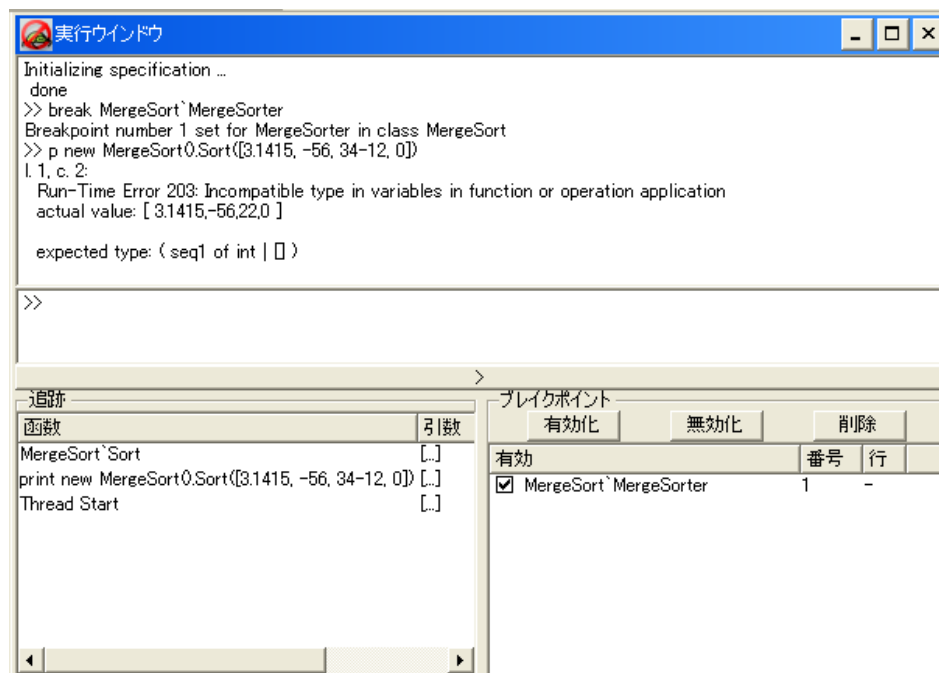



図 15: 動的型チェックエラー

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 を示すこともあり、その場合仕様書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、ExplSort クラス 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クラス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。

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）、仕様書の場所（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、小さい仕様書であっても、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ることによって定義される。

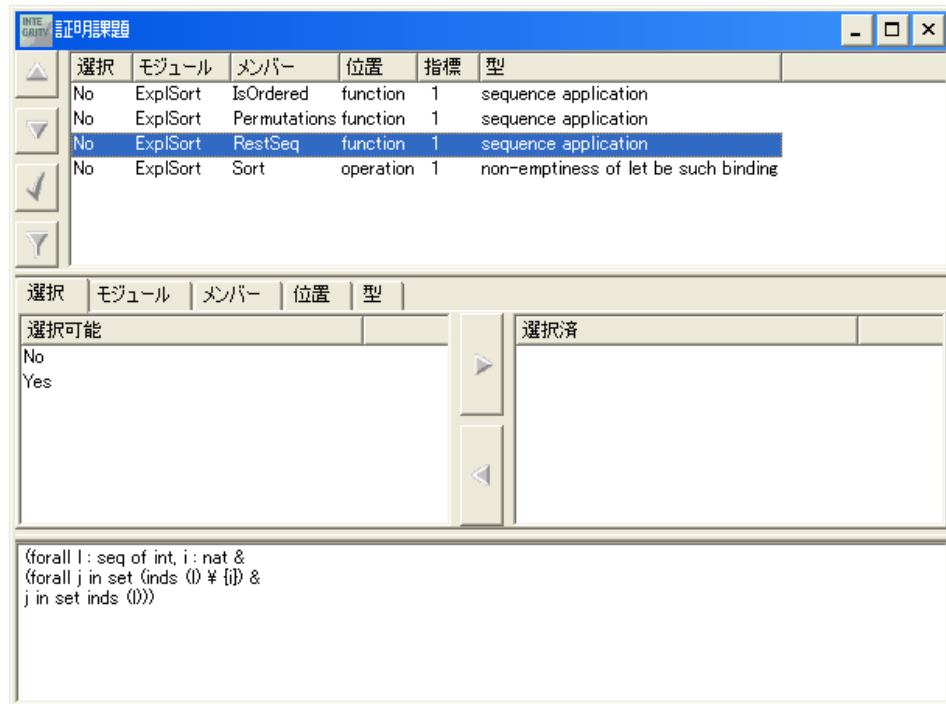


図 16: 証明課題ウインドウ

ている - そのため、大きな仕様書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 (詳細はセクション 4.6 参照)。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++ 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書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。

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 int &
(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

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 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 `Sort` 関数にある。これは基本的には、暗黙の `let` 命令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（さもないと仕様書は意味を成さない

⁶) が、これに関する記述が仕様書にないのである。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

⁶ ここで、変数 l 、これは仕様書の記述によれば任意の `int` の引数群であるが、について暗黙の定量化がなされている

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、両方とも `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 - 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。

ソースウインドウの `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を意味している。こうすれば、新しい証明課題は

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

となり、これはもちろん正しい。

他クラスの証明課題は同じようなやり方で扱うことができる。

3.8.5 マルチスレッド・モデル

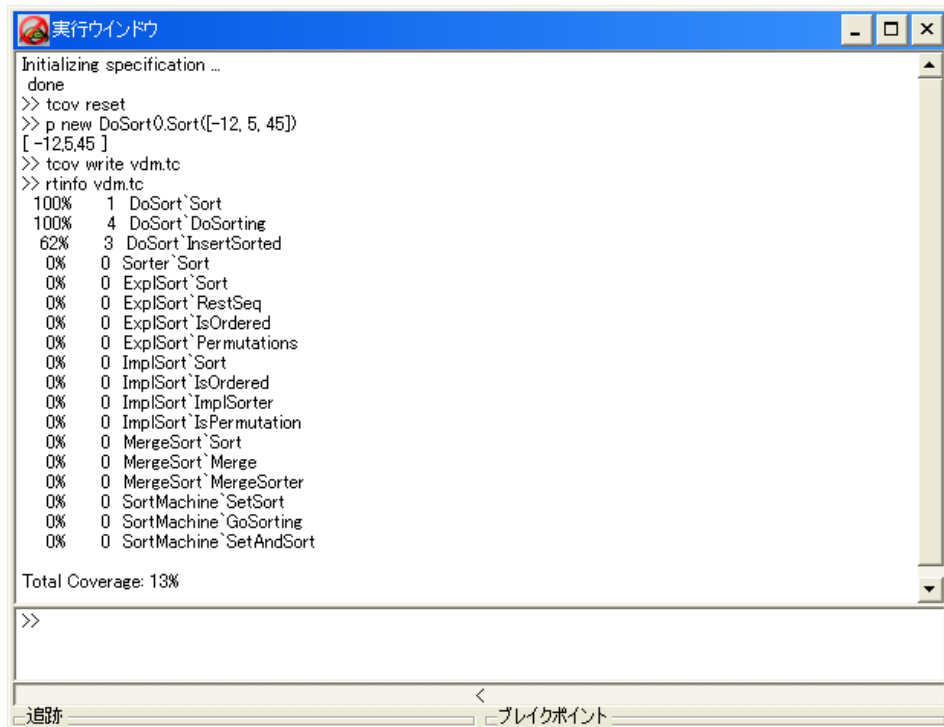
VDM++ はモデル中でマルチスレッドをサポートしており、この言語の特徴は特定のスレッドにブレイクポイントを設定したりステップ実行をしたりできるインタープリタでもサポートされている。特定のスレッドを選んでステップ実行することもできる。インタープリタにおけるスケジューリングのアルゴリズムは、Toolbox 内から選択される。

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++ の仕様書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書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書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書の構成物の呼び出しを実行させることで仕様書をテストする。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。清書機能は仕様書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書を作り出す。以下についてはセクション [3.10](#) でまた記述する。



```
Initializing specification ...
done
>> tcov reset
>> p new DoSort0.Sort([-12, 5, 45])
[-12,5,45]
>> tcov write vdm.tc
>> rtinfo vdm.tc
100%    1  DoSort`Sort
100%    4  DoSort`DoSorting
62%     3  DoSort`InsertSorted
0%      0  Sorter`Sort
0%      0  ExplSort`Sort
0%      0  ExplSort`RestSeq
0%      0  ExplSort`IsOrdered
0%      0  ExplSort`Permutations
0%      0  ImplSort`Sort
0%      0  ImplSort`IsOrdered
0%      0  ImplSort`ImplSorter
0%      0  ImplSort`IsPermutation
0%      0  MergeSort`Sort
0%      0  MergeSort`Merge
0%      0  MergeSort`MergeSorter
0%      0  SortMachine`SetSort
0%      0  SortMachine`GoSorting
0%      0  SortMachine`SetAndSort

Total Coverage: 13%

>>
```

図 17: テストカバレッジ情報の収集

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、与えられた仕様書のテスト情報には何も載っていない。それから仕様書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。これが仕様書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、仕様書の 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++ Toolbox (`vppde`) 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 -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

清書機能は仕様書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 Dialog 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6つの .rtf ファイルすべてを選択し、(アクション) ツールバーの  (清書) ボタンを押す。ログウインドウ にそれぞれの選択された入力ファイルに対応する .rtf.rtf ファイル が出来ているのがわかるはずだ。dosort.rtf.rtf ファイルで Word を起動してみる。VDM++ 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。仕様書のその他の部分は、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）

dosort.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⁷


⁷前のセクションのインタープリタの Response 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。


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%
total		15%

最後に、ファイルの最後にいて、Word の挿入 プルダウンメニューから Index and Tables ... を選択する。dosort.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 \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 4.10 参照) で説明する。

3.11 コード生成

VDM++ 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書から 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、[7] を参照のこと。

同様に、VDM++ から Java へのコード生成のライセンスを持っていれば、 (Java 生成) ボタンを押すことで自動的に仕様書から Java のコードを生成することができる。Java コード生成の詳細については、[8] を参照のこと。

3.12 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 [9] を参照のこと。

3.13 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++ 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 `vppgde` 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。

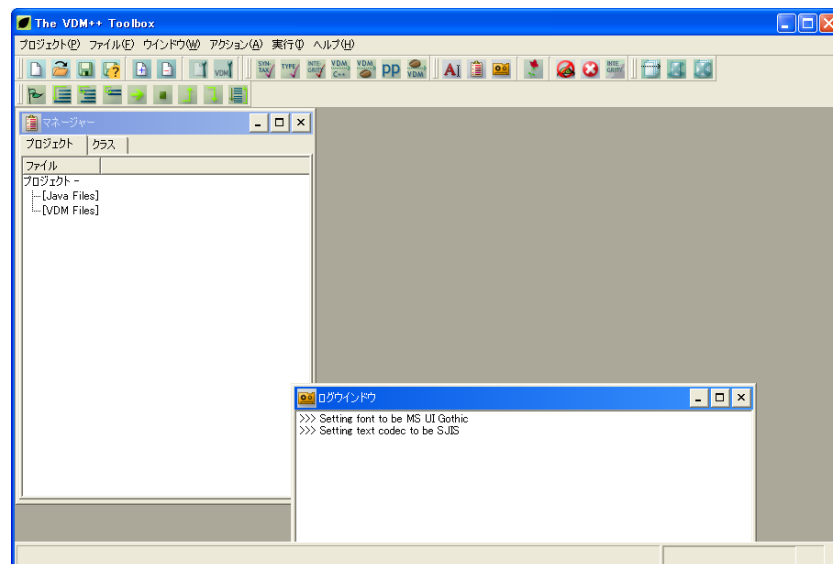


図 18: GUI スタートアップ画面

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⁸から成る 6 つのツールバー⁹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


⁸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

⁹Toolbox 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++ 形式の仕様書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 とクラスビューの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クラス の内容をツリー構造で表示する。図 19 にそれを示す。

クラスビュー は VDM ビュー と Java ビューで構成される。

VDM++ 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クラス の名前がVDM 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クラス それぞれの状態を記号 **S**, **T**, **C**, **J**, **P** で各々適切な欄にこれらは、当該クラス 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++）、Java コードを生成済み（translated to Java）、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 での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。

Java ビュー はVDM ビュー と似ているが Java ファイルで定義されたクラスの状

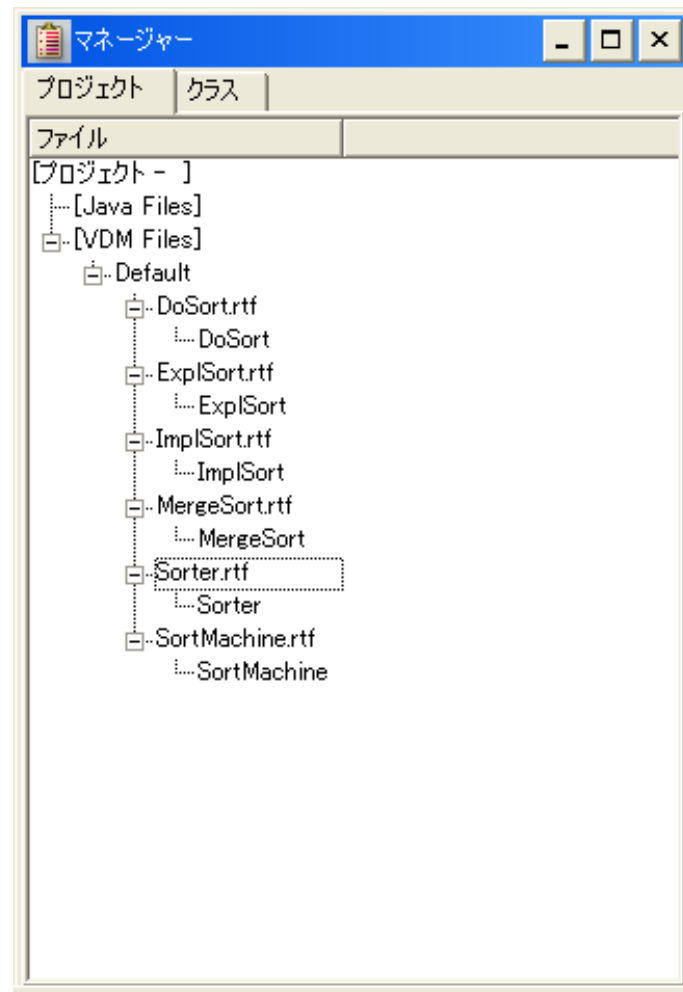









図 19: プロジェクトビュー

態と名前を表示するところが異なる（しつこいようだが構文チェックが成功していないと表示されない）。クラスが正しく構文チェック/型チェックを通ったかを示す記号 , , ,  が再度該当する欄に表示される。3 つ目の欄にはクラスがそれぞれ Java から VDM++ へ正しく変換されたかどうか (, ) が表記される。もう一度言うが空欄はまだ個々の処理が一度も実行されていないことを意味する。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、構文チェック 欄に  マークが表示され、現在の Toolbox での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。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。これを図 20 に示す。

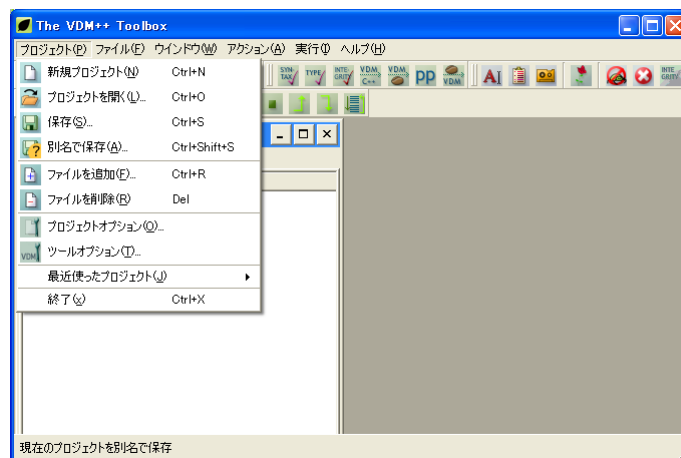


図 20: プロジェクトメニューとプロジェクトツールバー

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


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 に示したようなウインドウが表示され、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)
- Java コード生成 (セクション 4.9 参照);
- Java から VDM++生成 ([1] 参照).

ツールオプション (

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 仕様書の操作

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++/Java のコード生成、Java から VDM++の生成、; 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。

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・クラス 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、そのうちのいくつかは選択されたクラス 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クラス 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 (

41

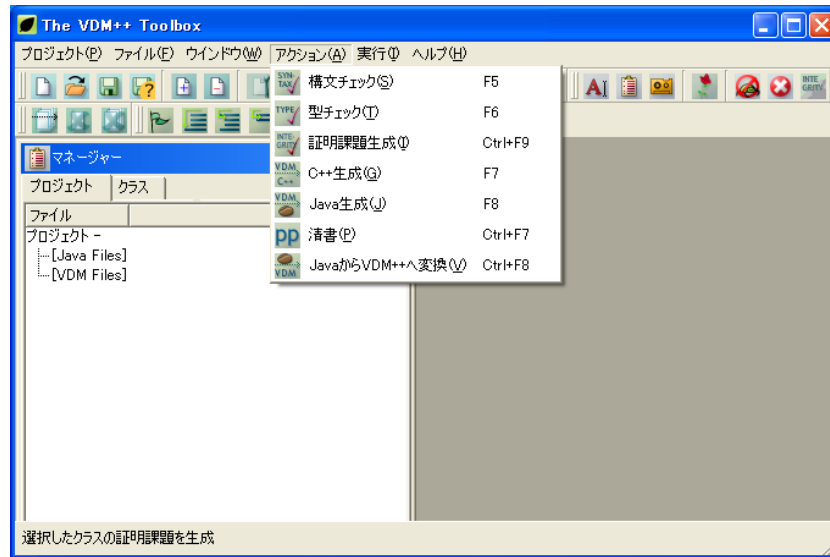






図 21: アクションメニューとツールバー


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

Java コード生成 (): セクション 4.9 参照

清書 (): セクション 4.7 参照

Java から VDM 生成 (): [1] 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。

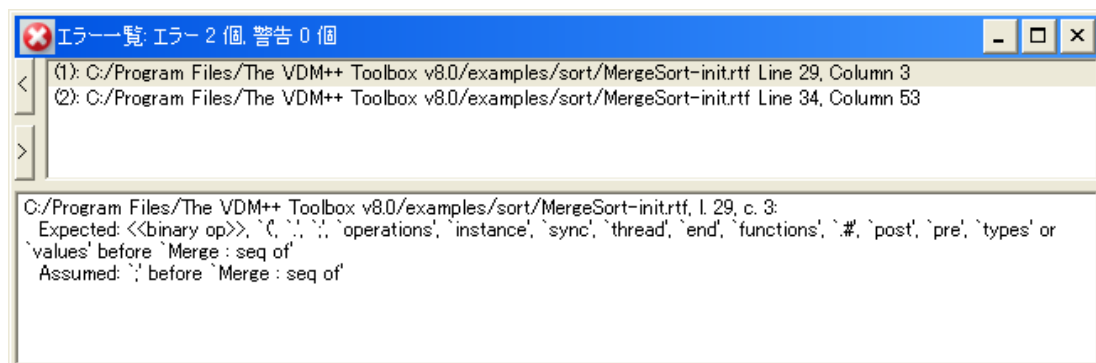







図 22: エラー一覧

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ったと自動的に開く。
代わりに ウィンドウ ツールバーで  ボタンを押すかウィンドウ メニューで相当する項目を選ぶことで手動でウィ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り、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書の一部を表示し、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たければ、ソース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クラスのひとつ）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

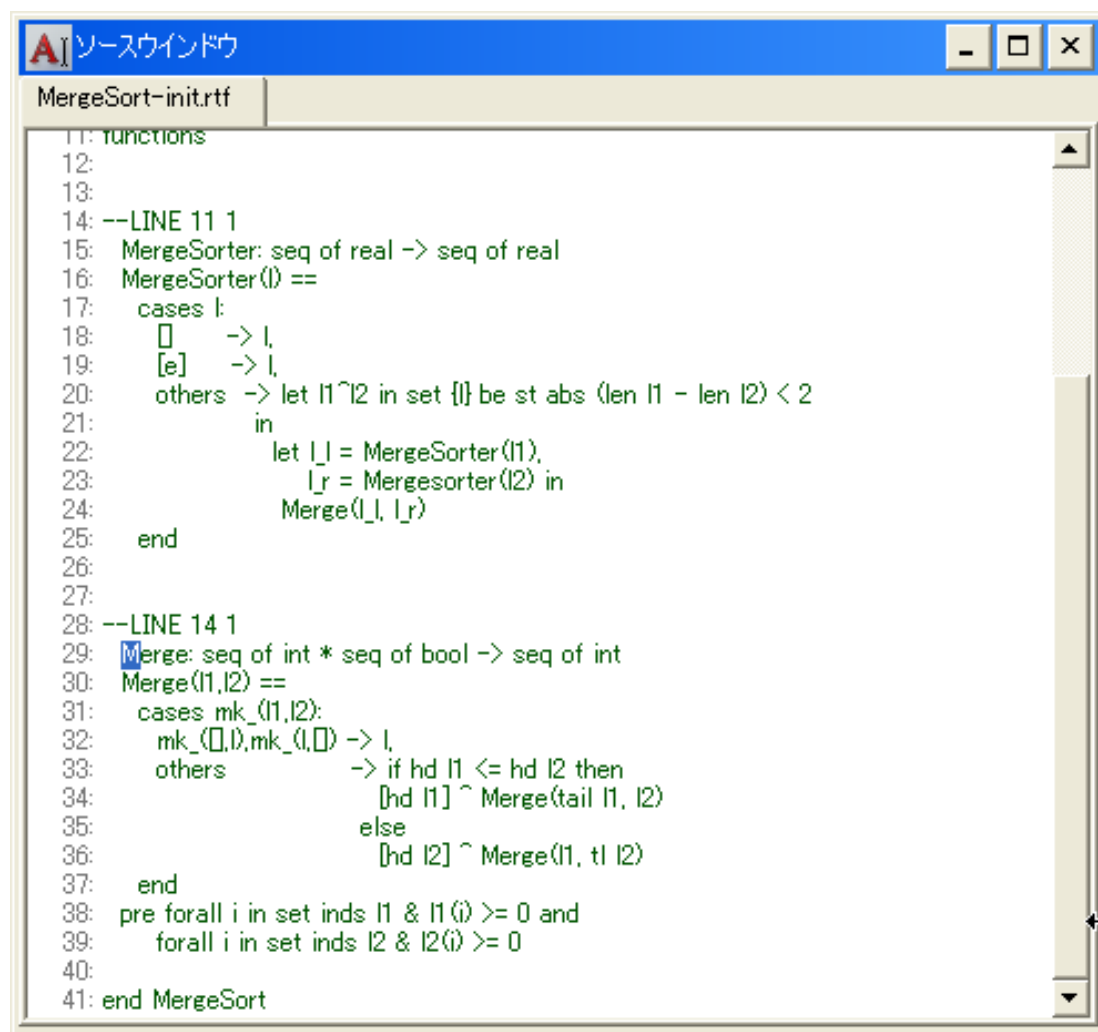




図 23: ソースウインドウ

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命令文のデバッグ、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。¹⁰:

```
vppde specfile
```

コマンドラインから vppde がオプションなし・1つのファイル名 (VDM++ の仕様書を含んでいなければならない) のみで入力されると、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。

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、仕様書の操作、実行、デバッグができる。下記で与えられたコマンドは vppde 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、仕様書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。

異なる構成物の名前を表示するのに、多くのコマンドが使われる。classes, functions, operations, instvars, 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。

*classes

定義済みのクラス名とその状態を表示

*functions class

クラス class 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書がそれを含む場合自動的に作成される

*operations class

与えられたクラス名で定義される操作名を表示する

¹⁰実行可能な vppde 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

***instvars class**

与えられた class 名で定義されるインスタンス変数名を表示する

***types class**

与えられたクラス名で定義済みの型名を表示する

***values class**

与えられたクラス名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++ 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書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や Rose-VDM++ リンクが書き込みをするところとなる。

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 .vppde 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書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書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書が構文的に正しい前提で動くため、仕様書は 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。

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書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書の構文エラーを報告することができる。仕様書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、仕様書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クラス（複数選択可能）¹¹ をマネージャー のプロジェクトビュー または VDM 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。ログウインドウ 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 とソースウインドウ が自動的に起動する。

¹¹ VDM ビュー でクラス を選択すると、構文チェック機能は実際には選択したクラス 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クラス 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クラス が処理に含まれる。

4.3.2 構文エラーのフォーマット

仕様書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。仕様書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 [4]) 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:

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

-p オプションをつかえると、vppde コマンドはそれぞれ1つ以上のクラス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++ 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書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っても大きくしてしまう傾向がある。

-R VDM++ 仕様書のテスト中違う構成物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 [4.10](#) を参照。

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)` は関数、値、型、インスタンス変数などを含むクラス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++ での構成物は何も含まれないということの意味する。またこれはもしそのファイルが複数のクラスを含んでいた場合も含む(クラス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クラス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書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書」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「明確に適格な仕様書」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。そのため仕様書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書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VDM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。ログウインドウが自動的に開き、選択されたファイルやクラス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インドウが自動的に起動される。Toolbox はクラス間の依存関係をすべて把握しているため、選択されたクラスのスーパークラスもすべて型チェックされる。

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 ウイ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、“Result of ‘conc’ can be an empty sequence ” 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。

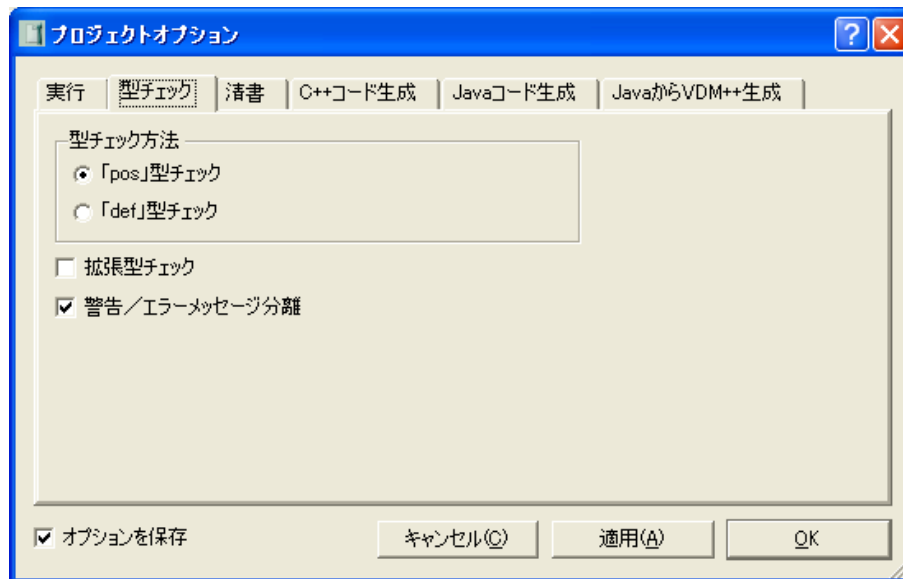


図 24: 型チェック機能のオプション設定

4.4.2 エラーおよびワーニングのフォーマット

型チェック機能の吐き出すすべてのワーニング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予想する型 (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++ の型の構文と同様である。(下記は例外)

- `seq of A` `seq1 of A` | `[]` と同じ意味。 `[]` は空シーケンスの型。
- `map A to B` `map A to B` | `{|->}` と同じ意味。 `{|->}` は空マップの型。
- `set of A` `set of A` | `{}` と同じ意味。 `{}` は空セットの型。
- `[A] A` | `nil` と同じ意味。
- `#` 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) という言葉を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取りうる。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

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

-t オプションを使うと vppde コマンドは specfile(s) の型チェックを行う。まず、仕様書が解析される。それから構文エラーが見つからなければ、仕様書の型チェック (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。([4])。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。

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) class option`

与えられたクラスの静的型チェックを行う。(カレントディレクトリの全クラスを型チェックするには、“*” 記号を用いる) `option` は `pos` または `def` でありこれは仕様書が `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++ の仕様書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クラス の型チェックをする必要はない (が仕様書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アクセス可能である。

VDM++ の構成物で実行できないものは、陰関数や陰操作、仕様の記述、型の縛り、モデリングに VDM++ の 3 値論理を課す制約に従った式である。VDM++ の平行性とリアルタイム部分も、インタープリタではまだ利用できない。

4.5.1 GUI

実行ウインドウ は (ウインドウ) 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。ウインドウ 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 Response 画面と Dialog 画面である : Dialog 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 Response 画面で見ることが出来る。VDM++ の式を評価するには、Dialog 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 (🔄): 仕様書の初期化をする。これによりグローバル変数とインスタンス変数が初期化される。インタプリタの初期化は構文チェック済みの定義の使用を可能にするため、最初になさなければならない。

ステップ実行 (👉): 次の命令文を実行する。関数内部に入らず次の行でとまる。このボタンは記述の全体を評価するため関数にはさほど有用ではない。

関数内をステップ実行 (👉): 次の式または命令文を実行する。関数内部に入ってとまる。

1 ステップ実行 (👉): 次のサブ式またはサブ命令文を実行する。関数内には入らずにとまる。

実行再開 (👉): 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命令文の最後まで評価が到達したときに使用する。


関数の実行を終了 (👉): 現在の関数や操作の評価を終了し、呼び出し元に戻る。このコマンドはもともと関数内をステップ実行とともに使われた。

一段上の関数の呼び出し位置を表示 (👉): このコマンドは仕様書が初期化された後、デバッガがブレイクポイントでとまったときに使用が可能。現在ディスプレイウインドウに表示されている箇所に比べて1レベル上にコンテキストをシフトする効果がある。このため、コンテキストは現在の関数から呼び元の関数へと移る。

一段下の関数の呼び出し位置を表示 (👉): このコマンドは仕様書が初期化された後、デバッガがブレイクポイントでとまったときに使用が可能。このコマンドは仕様書が初期化された後、デバッガがブレイクポイントでとまったときに使用が可能。現在ディスプレイウインドウに表示されている箇所に比べて1レベル下にコンテキストをシフトする効果がある。このため、コンテキストは現在の関数から呼び先の関数へと移る。

実行中断 (🛑): 式の評価を Stop する。ローカル変数やグローバル変数へのアクセスは print や debug コマンド（これらのコマンドの記述については下記を参照）内でこのボタンが押されたかどうか依存する。このコマンドはもともと仕様書内の潜在的な無限ループをブレークするのに使用された。

Dialog 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、Dialog 画面でこれらをタイプすることでインター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ープリタの初期化 ( (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 コマンドまたはソースウインドウにて希望する箇所をダブルクリックすると設定できる ¹²。

ブレイクポイントに来ると、ステップ実行 (), 1 ステップ実行 (), 一段上の関数の呼び出し位置を表示 (), 実行再開 (), 関数の実行を終了 () 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。

create コマンド を使うとオブジェクトを生成することが出来、destroy コマンドを使うとオブジェクトを破棄することが出来る。objects コマンド では現在のオブジェクトの名前がリスト表示される。

スレッド関連のコマンドが 3 つある。現在 (実行中) のスレッドは curthread コマンドで得られる。現在実行中のスレッドの一覧を見るのは threads コマンドで、違うスレッドの選択は selthread コマンドでそれぞれ可能である。

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 Dialog 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す

¹²RTF 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。その代わりに、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。

る。name は定義されたクラス名で分類された関数操作名で構成されていない。 (クラス名 ‘操作名’)

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、Response 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¹³。

***create (cr) name := stmt**

参照名 name でオブジェクトを生成し、まず stmt に割り当てる。stmt は、オブジェクトを参照する呼出し命令または新しい命令文 (他の種類の命令文の説明については [4] を参照のこと)。その後、name オブジェクトはデバッグのスコープに置かれる。

curthread

現在実行中のスレッドの識別子を出力する。

debug (d) expr

VDM++ 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。

インタプリタ内の式を評価したい場合は、記号\$\$を使って最後の評価の結果を参照する。詳細は下記 print コマンドの記載を参照。

¹³ ツールボックスのバージョン v9.0.6 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命令文でコマンドの評価は中断される。その式や命令文のスコープ中にある変数はすべて後で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。

***destroy name**

name で指定したオブジェクトを破棄する。

***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書からのすべての定義を初期化する。これはインスタンス変数 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。

***objects**

デバッガ中で生成されているオブジェクトを表示する。

print (p) expr, ...

すべてのブレイクポイントを無効にして、VDM++ 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 Source Window に表示される。

通常のVDM++ 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った場合（例：関数名だけ与えられていて、引数を与えられず（）で囲まれている関数が評価された場合）

インタプリタで式を評価する場合、最後の評価の結果を参照するのに記号\$\$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他の VDM++ の式に組み込まれている。

```
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。

priorityfile (pf) [filename ...]

有効なファイル名を指定した場合、指定したファイルから priority 情報を読み込んでスレッドのスケジューリングに利用する。(priority スケジューリングが有効なとき)

引数なしで実行された場合、現在インタプリタで使用されている priority ファイルを一覧表示する。

Priority ファイルのフォーマットの詳細については付録 G を参照のこと。

***push name**

指定したクラス name がモジュールスタックにプッシュされ、init 後にアクティブなクラスになる。

***pop**

現在のクラスがスタックから取り出される。もしアクティブなクラスが存在しない場合、警告が発行され何も起きない。

***popd**

popd コマンドは、デバッグを入れ子にした時に、最上位のデバッグを取り出す。これは、1つの大きな VDM モデルをデバッグしていて、スペルミスされた値をプリントしようとする時に、非常に便利である場合がある。そして、popd コマンドを用いることで、1つ前のデバッグセクションまで戻ることができる。

selthread id

id.

現在実行中のスレッドを id に置き換える

threads

現在実行中のスレッドの一覧を下記のフォーマットで表示する。

< thread id > < object ref > < status >

thread id はスレッドの識別子であり、ユニークな値である。オブジェクトの参照はスレッドの定義中でのオブジェクトの識別子（インタープリタ主導でコントロールされているスレッドの場合は *none* ）、*status* は下記のうちいずれかである：

状態	意味
Blocked	実行許可待ち
Stopped	ブレイクポイントで Stop 中.
Running	インタープリタにより現在実行中.
MaxReached	このスレッドによりタイムスライス毎の命令の最大数に達した.

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を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 機能がいつも仕様書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インドウの実行 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：

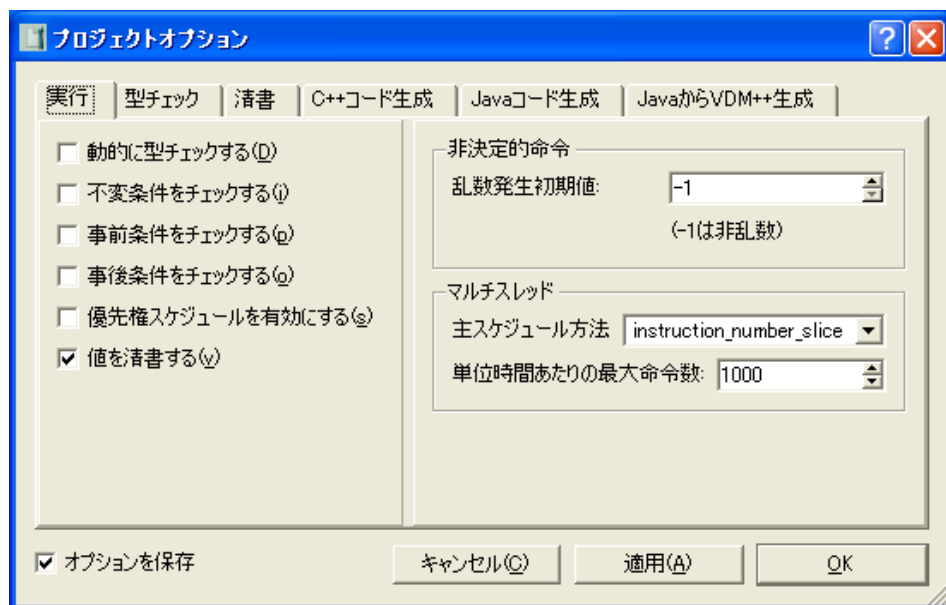


図 25: インタプリタのオプション設定

動的に型チェックする： このチェックを有効にすると、型がすでに定着したものであっても、式の型が VDM++ の仕様書に与えられた定義に沿ってチェックされる。

デフォルト：無効

不変条件をチェックする： このチェックを有効にすると、式がそのような不変条件が存在するときはいつでも、型と照合する

デフォルト：無効

事前条件をチェックする: このチェックを有効にするとすべての関数の評価済み事前条件が関数呼び出しの前にチェックされる。

デフォルト: 無効

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済みのすべての関数・操作の事後条件が、関数・操作の評価後にチェックされる。

デフォルト: 無効

値を清書する: 清書機能によりよく、読みやすいスタイルを使わせるために改行やインデントを挿入する。

乱数発生初期値: 与えられた `Int` 値で乱数ジェネレータを初期化する。これにより非決定性の命令文を構成するものにおいてサブ命令文の評価をランダムな順番で行うことができる。指定する数は 0 以上でなければならない。負の数はランダムな非決定性の命令文の評価には使えない。

デフォルトの値: -1

優先権スケジュールを有効にする: ラウンドロビンスケジューリングの代わりにプライオリティスケジューリングの使用を有効にする

デフォルト: 無効

単位時間あたりの最大命令数: タイムスライスごとの命令の数を指定する。(スライススケジューリングの命令数に使う)

デフォルトの値: 1000

主スケジュール方法: 主なスケジューリングアルゴリズムを指定する。pure cooperative scheduling か instruction number slicing のどちらか。

デフォルト: Instruction number slicing

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¹⁴。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 [5] で説明される。このライブラリを使うためには、VDMUtil.vpp 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、vpphome/stdlib ディレクトリに配置されている。

VDMUtil.vpp 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++ 仕様では、is not yet specified で定義された関数を実行できないが、これら特定の関数が実行できるための定義が 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.vpp 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 [5] 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.vpp 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 vpphome/stdlib ディレクトリに存在する。

math.vpp 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++ の仕様書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.vpp ファイルをプロジェクトに include すると、仕様書内で 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 [5] 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.vpp 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。

¹⁴インタプリタ上でのみ使用が可能であり、コード生成には対応していない。

io.vpp 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++ の仕様書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.vpp ファイルをプロジェクトに include すると、これらの IO の関数が仕様書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：

```
vppde -i [-O res-file] [-R testcoverage] [-D [-I]] [-P] [-Q]  
        [-Z priority-file] [-M num] argfile specfiles
```

-i オプションをつけると vppde 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++ 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物の場所の情報と、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され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。

 -D オプションも有効にしないと意味を成さない。

-P 評価済みのすべての関数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書のファイルと一緒に argument ファイルも使ったかのようになる。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 4.10 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。
- Z priority-file priority ベースのスケジューリングの使用を評価する。マルチスレッドモデルでのみ効果を発する。
- M num num をタイムスライスごとの命令数とする
- S algorithm 特定のスケジューリングアルゴリズムを使う。次のうちいずれか：
 - pure_cooperative Pure cooperative scheduling;
 - instruction_number_slice Instruction number sliced scheduling.

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。

create コマンドを使って新しいオブジェクトを生成することが出来、destroy コマンドを使って破棄することが出来る。現在のオブジェクトの名前を見るには objects コマンドが使える。

***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。
指定された関数/操作名は定義されたクラス名で分類されたもので構成されていなくてはならない。

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。

***create (cr) name := stmt**

参照名 name でオブジェクトを生成し、まず stmt に割り当てる。stmt は、オブジェクトを参照する呼出し命令または新しい命令文 (他の種類の命令文の説明については [4] を参照のこと)。その後、name オブジェクトはデバッグのスコープに置かれる。

***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命令文の最後まで評価が到達したときに使用する。

curthread

現在実行中のスレッドの識別子を出力する。

debug (d) expr

VDM++ 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。関数名・操作名は定義されたクラス名で分類された関数操作名で構成されていない

***destroy name**

name で参照されるオブジェクトを破棄する。

***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書のすべての定義を初期化する。これにはインスタンス変数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。

***objects**

デバッガ中で生成されたオブジェクトを表示する。

***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++ 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++ 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った場合（例：関数名だけ与えられていて、引数を与えられず（）で囲まれている関数が評価された場合）

最後の評価の結果を参照するのに記号 \$\$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他の VDM++ の式に組み込まれている。

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

priorityfile (pf) [filename ...]

priority スケジューリングが有効なときに有効なファイル名を指定した場合、指定したファイルから priority 情報を読み込んでスレッドのスケジューリングに利用する。

引数なしで呼び出された場合、インタプリタで使用されている現在のプライオリティファイルをリスト表示する。

プライオリティファイルで要求されるフォーマットについての詳細は、[G](#) を参照のこと。

*push name

name で指定したクラスがモジュールスタックにプッシュされ、初期化後にアクティブなクラスとなる。

remove number

number で指定したブレイクポイントを削除する。

selthread id

id.

現在実行中のスレッドを識別子 id のものにする。

set option

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。非決定性の命令文で構成されたもののうちのサブ命令文の評価をランダムに行うためである。integer は 0 以上でなくてはならない。負の数は非決定性の命令文のランダムな評価を無効にしまうからである。

primaryalgorithm string インタプリタで使用するプライマリのスケジューリングアルゴリズムを設定する。string は下記 2 つのうちいずれか

pure_cooperative (pc) - use pure cooperative scheduling;

instruction_number_slice (in) - use instruction number slicing scheduling

ここで、() 内の名前は使用される略語である。スケジューリングアルゴリズムについての詳細は、セクション 4.5.5 を参照のこと。デフォルトは instruction_number_slice

maxinstr integer タイムスライス毎の命令の最大数を maxinstr integer で指定する。指定した値がどう使われるかはセクション 4.5.5 を参照のこと。デフォルトは 1000。

priority priority ベースのスケジューリングを有効にする。詳細はセクション 4.5.5 参照のこと。

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・命令文で止まる。

***step (s)**

次の命令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・命令文を実行して止まる。関数・操作内部にも入る。

threads

現在実行中のスレッドの一覧を下記のフォーマットで表示する：

< thread id > < object ref > < status >

thread id はスレッドの識別子であり、ユニークな値である。*object ref* はスレッドの定義中でのオブジェクトの識別子（インタープリタ主導でコントロールされているスレッドの場合は *none* ）、*status* は下記のうちいずれかである：

状態	意味
Blocked	実行許可待ち
Stopped	ブレイクポイントで Stop 中.
Running	インタープリタにより現在実行中.
MaxReached	このスレッドによりタイムスライス毎の命令の最大数に達した.

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を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 機能がいつも仕様書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。

4.5.5 スレッドのスケジューリング

下記のプライマリスケジューリングアルゴリズムが利用できる：

Pure Cooperative このアルゴリズム下で、スレッドは下記のようになるまで実行される：

- 正常に完了；
- 相当するパーミッション `xx` が `false` となるオペレーションコールに到達；
- ブレイクポイントに到達または、インタプリタが割り込みされた

Instruction number slicing このアルゴリズム下で、スレッドは下記のようになるまで実行される：

- 正常に完了；
- 相当するパーミッション `xx` が `false` となるオペレーションコールに到達；
- スケジューリングされ実行された (内部の) 命令数が `maxinstr` 定数を超えた
- ブレイクポイントに到達または、インタプリタが割り込みされた

次にどのスレッドをスケジューリングするかを選択 (セカンダリのスケジューリングアルゴリズム) は単純なラウンドロビンスケジューリングに従うが、これは任意でプライオリティベースのオプションを選択することができる (Enable priority-based scheduling オプションを設定する。セクション 4.5.1 を参照)。


プライオリティベースのスケジューリングが使われている場合、メインのスレッド (ユーザが開始したスレッド) は常に最上位のプライオリティとなる。プライオリティファイルで特定されるどんなプライオリティよりも高い。

4.6 証明課題生成機能

証明課題生成機能は、仕様書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¹⁵のすべての値の数値化を含む VDM++ 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、その場合は仕様書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 VDM ビュー でファイルやクラス（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クラス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）、仕様書の箇所（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++ 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書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

¹⁵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書の精査によって決定しなくてはならないこともある

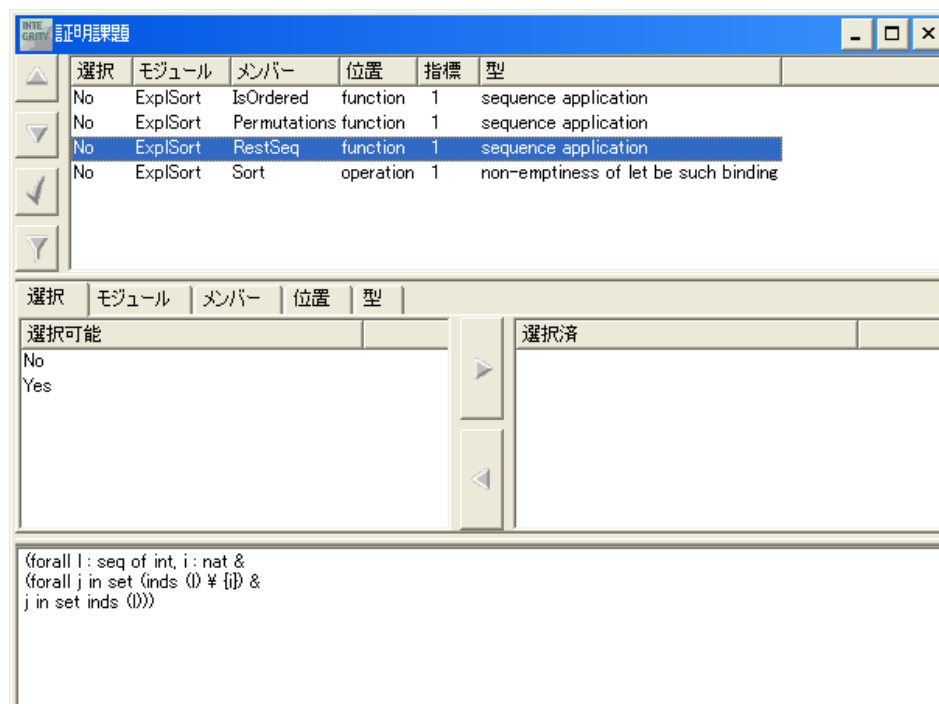


図 26: 証明課題ウインドウ

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

清書機能は仕様書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書の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 \LaTeX コマンドと VDM++ 仕様書の混合ならば、出力フォーマットは \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++ の ASCII バージョンを使っているのに対し \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++ 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書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、クラス 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。

\LaTeX ジェネレータは VDM++-VdmSl マクロを適切な相当する形式のファイルに結合する：vpp.sty for \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 4.10 と付録 B で、生成された \LaTeX ファイルを使用して \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 4.10 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¹⁶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。

定義の索引を出力: 関数、操作、型、インスタンス変数、クラス の定義のインデックスを生成する。

デフォルト: 無効

定義と仕様の索引を出力: 関数、操作、型、インスタンス変数、クラス の定義、型や関数・操作の使用された事象のインデックスを生成する。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: 無効

テストカバレッジの色付け: 仕様書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

いつでも、定義の索引を出力 と 定義と仕様の索引を出力のどちらかだけが有効になることに注意。

4.7.2 コマンドラインインターフェース

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

¹⁶ マネージャー の VDM ビュー でもクラスを選択することができる。清書機能は実際には選択したクラス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クラス定義を含んでいて、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 クラスが処理に含まれることを意味する。

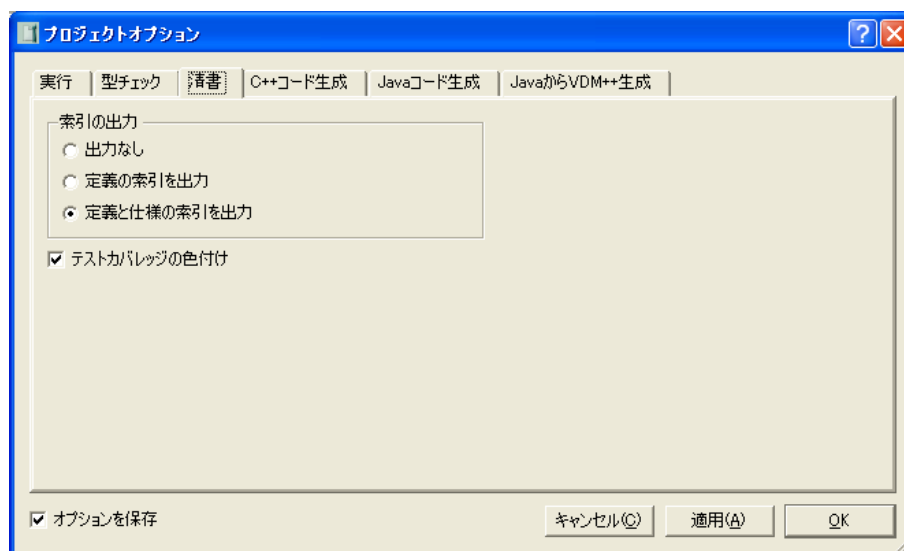


図 27: 清書機能のオプション設定

vppde に `-l` オプションをつけると引数で指定した VDM++ 仕様書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 \LaTeX と VDM++ 仕様書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 \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。 \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) になくても構わない。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 \LaTeX 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 形式で書かれたクラス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^AT_EX 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^AT_EX 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^AT_EX 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。

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^AT_EX フォーマットが使われていた場合、VDM++ の部分が VDM++ の VDM++-V_{DM}S_L 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++ の部分 (VDM++ の VDM++-V_{DM}S_L 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^AT_EX 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書の部分に印がつくように VDM++-V_{DM}S_L マクロで色をつける。

4.8 VDM++から C++コード生成

VDM++ から C++ へのコード生成のライセンス を持っていれば、Toolbox を使って仕様書から自動的に 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 [7] 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

```
vppde -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。仕様書はまず解析され、構文エラーがなければ 'pos' タイプの型チェック がされる。

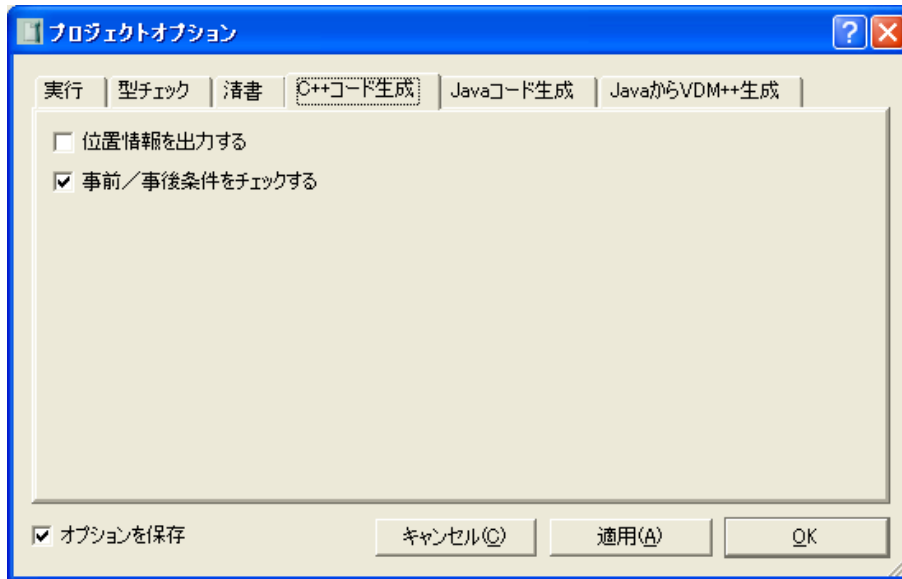


図 28: C++コード生成オプション設定

最後に、型エラーが見つからなければ仕様書がたくさんの 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 [7] に記述されている。

VDM++ 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 [7] 参照）

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) class [rti]**


クラス `class.` 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++ から Java へのコード生成

VDM++ から Java へのコード生成のライセンスを持っていれば、Toolbox を使って仕様書から自動的に Java のコードへ変換させることができる。ここではコード生成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 [8] で説明する。

Java へのコード生成機能は GUI、Toolbox のコマンドライン版、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。

4.9.1 GUI

Java のコード生成を GUI で起動するためには、まずマネージャーを使って Toolbox に変換させたいファイルまたはクラスを選択し、 (Java 生成) ボタンを押して生成を起動させる。複数ファイル/クラスを選択することもでき、すべて Java に変換される。

以下のコード生成向けオプションは図 29 で示すプロジェクトオプション ウィンドウの Java コード生成 タブで設定できる。

型以外は骨組みのみ生成する コード生成にクラスのスケルトンのみを生成させる (型、値、インスタンス変数の定義をフルに含むが関数や操作の定義は何もないクラス)。
デフォルト : off

型のみ生成する VDM++ の型定義に相当するコードのみ生成する。(値、インスタンス変数、関数、操作は無視される)
デフォルト : off

整数は long 型で生成する VDM++ の integer 値と変数を Java の integer の代わりに long に変換する。
デフォルト : off

並列構成でコードを生成する 並行処理のサポートを含むコードを生成する。
デフォルト : on

事前 / 事後条件関数を生成する 事前条件、事後条件に相当するコードを生成する。

デフォルト : on

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

名前の前に “vdm_” を付加しないで生成する ユーザが定義した操作と関数の名前の前に “vdm_” を追加しないでコード生成をする。これは、もし Java で定義済みの関数をオーバーロードするような時に便利である場合がある。

インターフェースの選択 VDM++モデルにおいて多重継承されている時に、それを Java コード生成用に単一継承にさせるために必要。もしそれができない場合は、Java コード生成する前に VDM++モデルを再構築する必要がある場合がある。

パッケージ 選択されたクラスを特定の Java パッケージに生成したい場合に使用することができる。

加えて、仕様書のどのクラスを Java インターフェースに変換するか選択し、コード生成が Java コードを保存するために生成したパッケージに名前をつけることができる。

4.9.2 コマンドラインインターフェース

```
vppde -j [options] specfile, ...
```

-j オプションをつけると vppde コマンドは specfile から Java のコードを生成する。仕様書はまず解析される。構文エラーが見つからなければ、'pos' タイプの型チェックがされる。型エラーが見つからなければ、最後に仕様書が多くの Java 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 [8] に記述されている。他の利用可能なオプションの一覧もそちらにある。

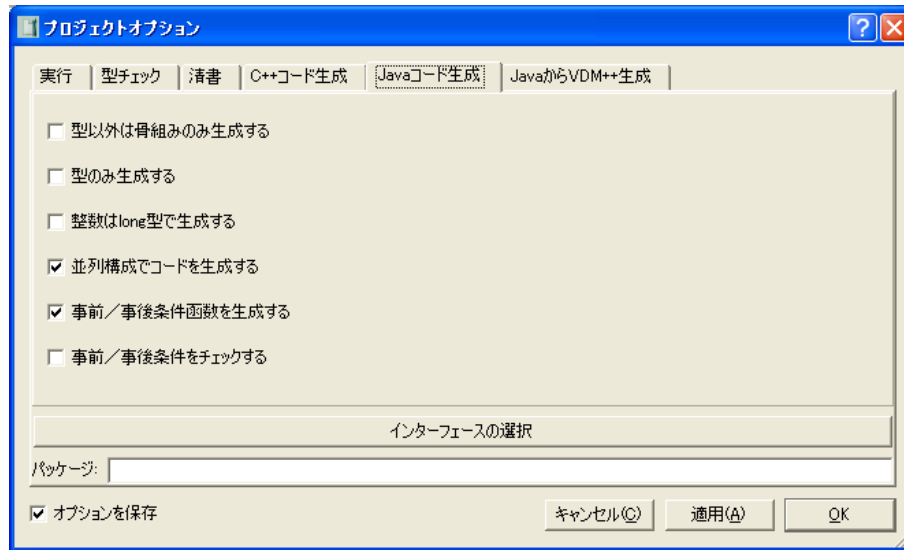


図 29: Java コード生成のオプション設定

4.9.3 Emacs インターフェース

Emacs インターフェースからは、Java コード生成向けのコマンドは 1 つしかない。このコマンドは `javacg` と呼ばれ、他の Emacs インターフェースのコマンドと同じようにコマンドプロンプトから入力しなくてはならない。

`*javacg (jcg) class [options]`
クラス `class` の Java コードを生成する。

4.10 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++ 仕様書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書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命令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。

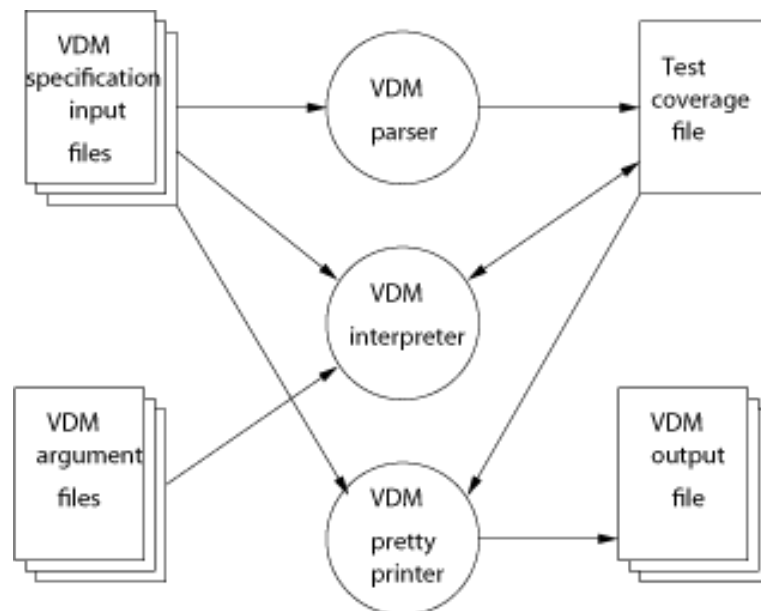


図 30: VDM モデルの体系的テスト

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 30 参照)

1. *test coverage file* を用意する。片方の入力フォーマットの VDM++ のファイルはまず特別なオプション付きの VDM++ 構文解析ツールに渡される。これが仕様書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生成する。

2. VDM++ 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書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物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、すべての仕様書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書の清書版を生成する特別なオプションつきで清書機能が使用される。VDM++ 仕様書の入力ファイルでは VDM++ 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++ 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++ の仕様書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。

4.10.1 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：

```
"vpphome/bin/vppde" -p -R vdm.tc Sorter.rtf DoSort.rtf ExplSort.rtf  
ImplSort.rtf MergeSort.rtf SortMachine.rtf
```

4.10.2 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：

```
"vpphome/bin/vppde" -i -R vdm.tc -O dosort.res sort.arg
Sorter.rtf DoSort.rtf ExplSort.rtf ImplSort.rtf MergeSort.rtf
SortMachine.rtf
```

4.10.3 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：

```
"vpphome/bin/vppde" -lr Sorter.rtf DoSort.rtf
ExplSort.rtf ImplSort.rtf MergeSort.rtf SortMachine.rtf
```

生成された清書版のファイルでは、仕様書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 のインタープリタの 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。

4.10.4 L^AT_EX 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++ 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^AT_EX 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。仕様書は `vpphome/examples/sort/*.vpp` ファイルにある。

この例では、`new DoSort().Sort([-12,5,45])` が `DoSort` クラスの仕様書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> vppde -p -R vdm.tc sorter.vpp dosort.vpp
Parsing "sorter.vpp" ... done
Parsing "dosort.vpp" ... done
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++ 操作: `DoSort.Sort` の呼び出しが含まれている。

```
prompt> vppde -i -R vdm.tc sort.arg sorter.vpp dosort.vpp
Initializing specification ...
[ -12,5,45 ]
prompt>
```

インタプリタが `-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 `DoSort` クラス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書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で当該関数/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 31 はこれを示したものである。

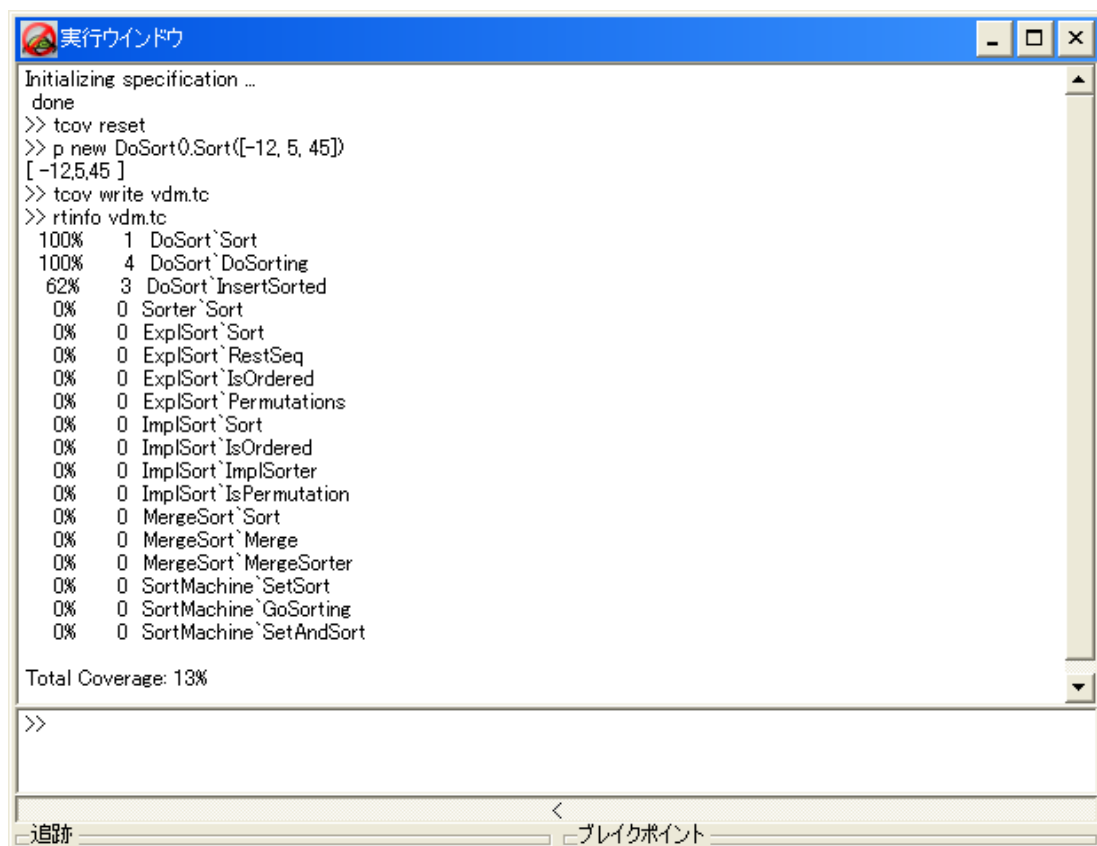


図 31: Showing test coverage information in the Toolbox

この \LaTeX 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> vppde -lr sorter.vpp dosort.vpp
```

```

Parsing "sorter.vpp" ... done
Parsing "dosort.vpp" ... done
Generating latex to file sorter.vpp.tex ... done
Generating latex to file dosort.vpp.tex ... done
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`、`Sort` といった関数や操作の部分にマークがつく。古いバージョンの LATEX は色付けをサポートしていないため、`-r` オプションは LATEX2 ϵ に対してのみ使われることに注意。sorter.vpp.tex ファイル上で LATEX を走らせると図 32 のように結果が表示される。

```

DoSort :  $\mathbb{R}^* \rightarrow \mathbb{R}^*$ 
DoSort (l)  $\triangleq$ 
  if l = []
  then []
  else let sorted = DoSort (tl l) in
    InsertSorted (hd l, sorted);

InsertSorted : PosReal  $\times$  PosReal $^*$   $\rightarrow$  PosReal $^*$ 
InsertSorted (i, l)  $\triangleq$ 
  cases true :
    (l = [])  $\rightarrow$  [i],
    (i  $\leq$  hd l)  $\rightarrow$  [i]  $\frown$  l,
    others  $\rightarrow$  [hd l]  $\frown$  InsertSorted (i, tl l)
  end

```

図 32: Sorting example のテストカバレッジ

`DoSort`、`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`、`Sort` を呼んでいるためである。

付録 B に、VDM++ 仕様書と LATEX 部分の分け方について詳細な記述がある。

LATEX 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、仕様書のテストスイートでカバーできていない色付きの部分と LATEX 形式でカバレッジのパース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 \LaTeX テストカバレッジ環境

\LaTeX 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 `rtinfo` 環境は `DoSort` クラス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：

```
\begin{rtinfo}
[TotalxCoverage]{vdm.tc}[DoSort]
DoSorting
InsertSorted
Sort
\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 `DoSort`）クラス名となる。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/操作 `DoSort`、`DoSorting`、`DoSort`、`InsertSorted`、`DoSort`、`Sort` のみが一覧表示されている。

実行結果のテーブルは図 33 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 Suite : vdm.tc
Class : DoSort

Name	#Calls	Coverage
DoSort'DoSorting	4	✓
DoSort'InsertSorted	3	62%
DoSort'Sort	1	✓
Total Coverage		79%

図 33: テストカバレッジテーブルの例

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

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

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

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

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

```
function list = { identifier } ;
```

Colouring

テストスイートでカバーできていない仕様書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 \LaTeX ファイルはVDM++ の仕様書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{vpp}

\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 \LaTeX コードには、マクロ`\color{covered}`と`\color{not-covered}`が仕様書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 32 に示す。

DoSort‘InsertSorted の case 式の others 節が \LaTeX 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‘Sort 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書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てはならない。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] CSK. *The Java to VDM++ User Manual*. CSK.
- [2] CSK. *The Rose-VDM++ Link*. CSK.
- [3] CSK. *VDM++ Installation Guide*. CSK.
- [4] CSK. *The VDM++ Language*. CSK.
- [5] CSK. *The VDM-SL Language*. CSK.
- [6] CSK. *VDM++ Sorting Algorithms*. CSK.
- [7] CSK. *The VDM++ to C++ Code Generator*. CSK.
- [8] CSK. The vdm++ to java code generator. Tech. rep.
- [9] CSK. *VDM Toolbox API*. CSK.
- [10] DICKINSON, I., AND LINES, K. Typesetting VDM-SL with VDM-SL macros. Tech. rep., National Physical Laboratory, Teddington, Middlessex, TW11 0LW, UK, July 1995.
- [11] FITZGERALD, J., AND LARSEN, P. G. *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.
- [12] FITZGERALD, J., LARSEN, P. G., MUKHERJEE, P., PLAT, N., AND VERHOEF, M. *Validated Designs for Object-oriented Systems*. Springer, New York, 2005.
- [13] 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.

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メソッド でブレイ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。

Java コード生成機能: 仕様書から Java のコードを自動生成する。Toolbox から Java コード生成機能にアクセスするには、別ライセンスが必要である。

L^AT_EX: 一般的な組版システム

清書機能: ファイルを処理して VDM++ の入力ファイルの VDM++ 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++ 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 [13]。

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。VDM++ 言語であっても同様である。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++ と L^AT_EX の結合

このセクションでは、VDM++ の仕様部分を含む L^AT_EX 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 仕様書ファイルのフォーマット

システムを処理するのに L^AT_EX 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++ の ASCII 仕様書であり、もうひとつは原文の混ざった ASCII 仕様書である。後者は仕様書の部分とそうでない部分が “`\begin{vdm_al}`” と “`\end{vdm_al}`” に囲まれているかそうでないかで区別される。仕様書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）。“`\begin{vdm_al}`” は仕様書中の任意の箇所に置くことはできず、`class`, `instance variables`, `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/mergesort.vpp
```

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 L^AT_EX 文書のセットアップ

VDM++ と L^AT_EX を結合するとき、通常の機能は L^AT_EX でも L^AT_EX2_ε でも使うことができるが、テストカバレッジとの結合は L^AT_EX2_ε を使わないとできない。文書の色付け機能など L^AT_EX2_ε でのみ使用できる機能がいくつかあるからだ。

以下の L^AT_EX コードの例は VDM++ の部分を含む一般的な L^AT_EX 文書を生成するためには含まれていなくてはならない L^AT_EX 形式のファイルを示す：

vpp スタイルファイルは Toolbox に含まれている。

```
\documentstyle[vpp]{article}

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

図 34: 旧バージョン \LaTeX 文書の例

```
\documentclass{article}
\usepackage{vpp}

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

図 35: $\text{\LaTeX}2\epsilon$ 文書の例

\LaTeX の見出しは VDM++ 仕様書ファイルのひとつか VDM++ 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書のファイルは Toolbox により \LaTeX 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 `vppde` 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.tex` ファイルに挿入されている。

行番号つけ

生成された \LaTeX 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 [\[10\]](#) 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 \LaTeX 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 \LaTeX 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：クラス、関数、操作、型、インスタンス変数の定義すべてに対するインデックスと関数、型、クラス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物を参照するかを分類するため、自動的に異なる \LaTeX マクロに挿入される。定義に対するマクロは：

- `InstVarDef` インスタンス変数が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。
- `ClassDef` クラス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 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の仕様についてのみインデックスが振られる。
- `ClassOcc` クラスが使用されている箇所を示す。

これらの \LaTeX マクロはインデックス付けを使用するためには \LaTeX 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{\InstVarDef}[1]{\bf #1}
\newcommand{\TypeDef}[1]{\bf #1}
\newcommand{\TypeOcc}[1]{\it #1}
\newcommand{\FuncDef}[1]{\bf #1}
\newcommand{\FuncOcc}[1]{#1}
\newcommand{\ClassDef}[1]{\sf #1}
\newcommand{\ClassOcc}[1]{#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 \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 (\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


サポート外構成物

現状、 \LaTeX の清書機能ではサポートしていない構文の構成物が1つある：コメントを活字に組むことができないのだ。 \LaTeX の清書機能では単に無視されるだけである。VDM++ 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 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プション

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`

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。

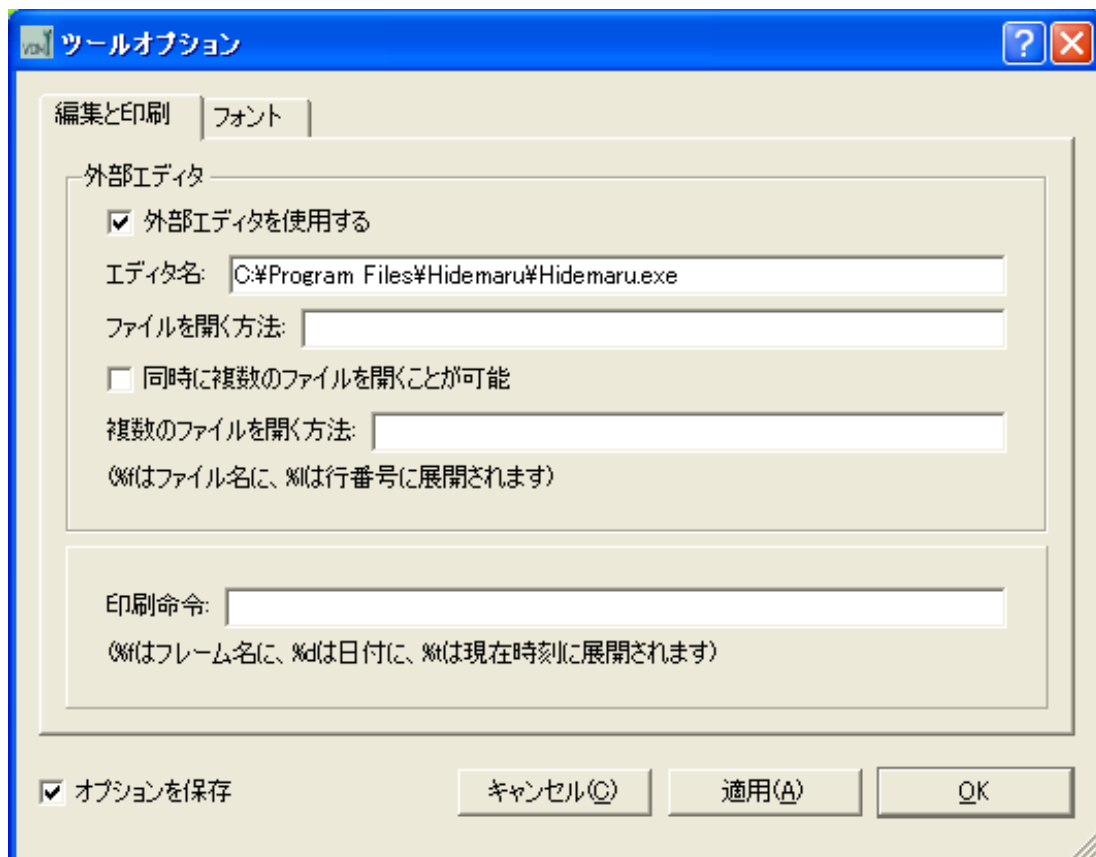


図 36: エディタと印刷オプションの設定

C.2 多言語サポート

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書自体（仕様書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 フォント タブ（図 37 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 (VDM) の項目を選択すると起動する。ツールオプションのフォント 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、フォントタブでは、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。

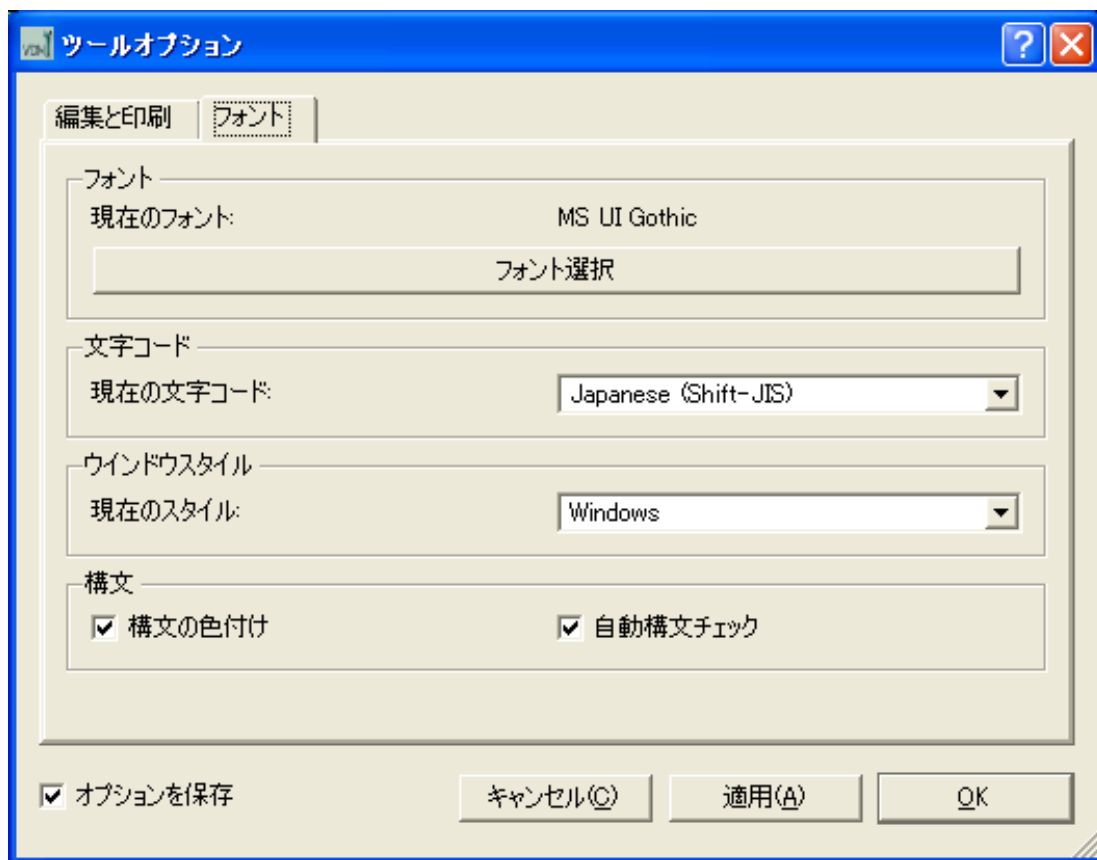


図 37: フォントオプションの設定

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ると限られている。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.3 UML リンクオプション

このタブは、外部の UML ツールとの希望のインターフェースを選択することが可能である。現在は、サポートされている外部の UML ツールは、JUDE :

<http://jude.change-vision.com/jude-web/index.html>

と Enterprise Architect:

<http://www.sparxsystems.com.au/products/ea/downloads.html>

である。

これらへのリンクは両方とも、UML ツール間の XMI インターフェース規格を用いることで実現されている。ここにある、また、テンプレートと同様に VDM レベルにおけるタイプがそのようなファイルに使用したがつている都合のよいファイルを述べるのにおいて可能です。

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 `vppde.el` は Emacs から直接 Toolbox の機能にアクセスすることを可能にする Emacs マクロを含む。セクション 3 からのガイドツアーは `mergesort-init.rtf` ファイルの代わりに `mergesort.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 [3] 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 `vppde` (`vppde` 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++ 開発環境 (`vppde`) 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++ 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。 `mergesort.vpp` とタイプしてリターンキーを押す。

`vppde` 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(以後仕様書ウィンドウと呼ぶ)は `mergesort.vpp` ファイルを表示しており、もう半分は `vppde` 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)。構文エラーは仕様書ウィンドウでは `=>` 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。
- いくつかの 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 vdmloop.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++ test examples for the
rem -- sorting example
set SPEC1=..\DoSort.rtf ..\SortMachine.rtf ..\ExplSort.rtf
set SPEC2=..\ImplSort.rtf ..\Sorter.rtf ..\MergeSort.rtf

vppde -p -R vdm.tc %SPEC1% %SPEC2%
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 SPEC1=..\DoSort.rtf ..\SortMachine.rtf ..\ExplSort.rtf
set SPEC2= ..\ImplSort.rtf ..\Sorter.rtf ..\MergeSort.rtf

vppde -i -R vdm.tc -O %1.res %1 %SPEC1% %SPEC2%

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

:end
```

単純なひとつの変数の変わりに SPEC1、SPEC2 と 2 つの変数が定義されているのは、長い行になるのを避けるためである。

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++ test examples for the sorting example.
SPEC="../dosort.vpp ../implsort.vpp ../sorter.vpp ../explsort.vpp \
      ../mergesort.vpp ../sortmachine.vpp"

## Generate the test coverage file vdm.tc
```

```
vppde -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="../dosort.vpp ../implsort.vpp ../sorter.vpp ../explsort.vpp \
    ../mergesort.vpp ../sortmachine.vpp"
```

```
## Run the specification with argument while collecting
## test coverage information, and write the result to an
## output file.
vppde -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++ Toolbox** のバージョン `v9.0.6` 以降でないとは入っていない。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。

G プライオリティファイルのフォーマット

プライオリティファイルのフォーマットは以下のとおり

priority file = priority entry { ‘,’ , priority entry } ‘,’ ;

priority entry = class name ‘.’ numeral ;

numeral = digit, { digit } ;

digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’ ;

索引

- .vppde ファイル, 48
- .emacs ファイル, 110
- \$\$, 62, 64, 71
- vppde, 110
- vppde
 - コマンドラインオプション, 69
- backtrace コマンド, 71
- break コマンド, 61
- break コマンド, 62, 71
- C++コード生成, 35
- C++ファイル, 84
- classes コマンド, 46
- codegen コマンド, 84
- cont コマンド, 71
- cquit コマンド, 48
- create コマンド, 61, 62, 71
- curthread コマンド, 61, 62, 71, 73
- debug コマンド, 61, 62, 71
- def 型, 53, 54, 56
- delete コマンド, 61, 63, 72
- destroy コマンド, 61, 63, 72
- dir コマンド, 47
- disable コマンド, 63, 72
- Emacs インターフェース, 110
- enable コマンド, 63, 72
- finish コマンド, 72
- first コマンド, 51, 57
- functions コマンド, 46
- GUI, 37
 - スタート, 8, 37
- help コマンド, 47
- info コマンド, 47
- init コマンド, 63, 72
- instvars コマンド, 47
- javacg コマンド, 87
- Java コード生成, 35
- last コマンド, 51, 57
- latex コマンド, 82
- next コマンド, 52, 57
- objects コマンド, 61, 63, 72
- operations コマンド, 46
- Options
 - C++ Code Generator, 84
 - Editor and Print, 107
 - Interpreter, 26
 - Java Code Generator, 87
 - Pretty Printer, 81
 - Type Checker, 55
- Options:Interpreter, 66
- option コマンド, 74
- popd コマンド, 64, 72
- pop コマンド, 64, 72
- pos 型, 53, 54, 83, 86
- previous コマンド, 52, 57
- print コマンド, 23, 61, 63, 72
- priorityfile コマンド, 64, 73
- push コマンド, 64, 73
- pwd コマンド, 34, 47
- quit コマンド, 48

- read コマンド, 51
- remove コマンド, 73
- rtinfo コマンド, 33, 90

- script コマンド, 47
- selthread コマンド, 61, 65
- set full コマンド, 57
- set コマンド, 57
- singlestep コマンド, 74
- stepin コマンド, 75
- step コマンド, 75
- system コマンド, 47

- tcov read コマンド, 65, 75
- tcov reset コマンド, 33, 66, 76
- tcov write コマンド, 33, 65, 75
- tcov コマンド, 65, 75
- threads コマンド, 61, 65, 75
- typecheck コマンド, 57
- types コマンド, 47

- unset full コマンド, 58
- unset コマンド, 76

- values コマンド, 47
- VDM.dot ファイル, 7, 62, 79, 114
- vdmgde コマンド, 8, 37
- VDM 標準, 1
- vpp.sty ファイル, 79

- インタープリタ, 20
 - コマンド, 59
 - 初期化, 21, 59, 61
 - 停止, 63, 64
- インタープリタウィンドウ, 10, 20, 59
- インプット
 - 作成, 7
 - 他言語, 1
 - フォーマット, 1

- エラーリスト, 10, 13, 49, 54

- オンラインヘルプ, 8

- 外部エディタ, 15, 45
- 型エラー, 54
- 型チェック, 15, 53
 - 動的, 26, 69
- 型の正しさ, ⇒
 - pos 型,
 - def 型

- 関数
 - 暗黙, 59
 - 事前条件, 21

- クラスビュー, 10

- 構文エラー, 13, 49, 110
 - 修正, 13
 - フォーマット, 50
- 構文チェック機能, 15
- 構文チェック, 12
- 構文の色付け, 108
- コード生成, ⇒
 - C++コード生成,
 - Java コード生成
- コールスタック, 23, 24, 59
- コマンドラインインターフェース, 46
 - C++コード生成, 83
 - Java コード生成, 86
 - インタープリタ, 69
 - 開始, 46
 - 型チェック機能, 56
 - 構文チェック機能, 50
 - 清書, 80
 - デバッグ, 69
 - ファイルの初期化, 48

- 事後条件チェック, 27, 67, 69

事前条件チェック, 27, 66, 69

実行不可能な構成物, 23

自動構文チェック, 108

証明課題, 27

清書

 L^AT_EX 文書のセットアップ, 102

 インデックス, 104

 コメント, 105

 例, 102

清書機能, 34

前提条件関数, ⇒

 関数,

 事前条件

ソースウィンドウ, 10

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, 88, 91

 環境, 94

 テストスイート, 32, 88

 ファイル, 32, 81, 88

 例, 91

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

不変条件チェック, 27, 69

プライオリティファイル

 フォーマット, 115

ブレイクポイント, 23, 62, 71

Microsoft Word の設定, 62

削除, 24, 59, 61, 63, 73

設定, 23, 61, 62, 71

無効, 25, 59, 63, 72

無視, 23, 61, 70

有効, 25, 59, 63, 72

プロジェクト, 38

 ファイル追加, 11

プロジェクトビュー, 10

ヘルプ, 45

マネージャー, 10, 38

ライセンス, 35, 83, 85

ログウィンドウ, 10