九州大学学術情報リポジトリ
Kyushu University Institutional Repository

形式仕様記述における探索的モデリング

小田, 朋宏

https://doi.org/10.15017/1866375

出版情報：九州大学, 2017, 博士（工学）, 論文博士
バージョン：
権利関係：

形式仕様記述における探索的モデリング

小田朋宏

平成 29 年 7 月

論文梗概

仕様記述の誤りはソフトウェア開発全体の生産性および信頼性に重大な影響を与えることから，ソフトウェアシステムの開発において高品質な仕様記述およびその仕様記述に忠実な実装を作成するための技術として，形式手法が研究され産業界において適用されている．形式仕様記述の適用は，ソフトウェア開発全体のコストが軽減され，また，全体のコストのうち仕様記述工程の占める割合が増加することが適用事例により知られている．これは，ソフトウェア開発全体の中でより多くの意思決定が形式仕様記述において行われることを示しており，形式仕様記述を導入したソフトウェア開発全体の生産性の向上においては，形式仕様記述工程の生産性が重要になることが考えられる．

本論文では，形式仕様記述工程を探索的仕様記述と精緻化の2つの工程に切り分け，探索的仕様記述を支援するための道具立てを提案する．探索的仕様記述は仕様記述に求められる妥当性と機能項目の網羅性と実現可能性を獲得する工程であり，対象ドメインおよびその中でシステムが解決すべき問題を理解し，その問題に対して技術的経済的に実現可能な形で解決策として記述することが探索的仕様記述で求められる作業である．形式仕様記述という数学的な対象を，ドメイン専門家やシステムのオーナーや利用者や実装技術の専門家とのコミュニケーションや，解を記述するために必要な創造性のために生かすことが，探索的仕様記述を支援するツールに求められる．本研究では，探索的仕様記述を支援するツールを設計する上で考慮すべき設計指針を挙げた上で，探索的仕様記述支援環境 ViennaTalk を開発した．ViennaTalk は，VDM-SL 仕様を記述し，プロトタイプとして実行させるためのライブラリを Smalltalk 環境上に実現したものであり，探索的仕様記述を支援するツールとして，Web IDE である VDMPad，UI プロトタイピング環境 Lively Walk-Through，Web API プロトタイピング環境 Webly Walk-Through を提供する．ViennaTalk は，産業界で VDM による成功事例を持つ熟練技術者と学術研究者によって，探索的仕様記述支援環境として妥当であると評価された．また，ViennaTalk のコード生成器の性能と生成されたコードの可読性を評価し，既存ツールに対して性能面および可読性での優位性を示すことで，有用性を示した．

九州大学大学院システム情報科学府情報知能工学専攻

目次

第 1 章　序論 **1**

1.1 背景 1

1.2 解決すべき課題 2

1.3 研究の目的と方法 3

1.4 関連研究 4

1.5 形式仕様記述言語 VDM-SL 6

1.6 本論文の構成 8

第 2 章　探索的仕様記述 **9**

2.1 悪定義問題としてのソフトウェア開発 9

2.1.1 問題解決の視点から見た形式仕様の役割と効果 . . 10

2.2 仕様記述工程の学際性 11

2.2.1 ドメイン知識に起因する学際性 12

2.2.2 技術知識に起因する学際性 12

2.3 探索的仕様記述と精緻化 13

2.3.1 形式仕様の品質特性 13

2.3.2 品質特性の分類による工程の分割 16

2.4 探索的仕様記述における試行錯誤 18

2.5 探索的仕様記述の例：自動販売機 21

2.5.1 要求の概要 21

2.5.2 暫定的な仕様を記述する 23

2.5.3 妥当性，機能項目の網羅性および実現性を獲得する 33

2.5.4 探索的仕様記述のまとめ 37

2.6 探索的仕様記述から精緻化への移行 39

第 3 章　探索的仕様記述への支援 **41**

3.1 探索的仕様記述の失敗要因 41

3.1.1 コミュニケーションを介した試行錯誤での失敗要因 41

3.1.2 個人作業での試行錯誤での失敗要因 43

3.2 支援ツールの設計指針 45
3.3 探索的仕様記述における形式仕様技術 48
3.3.1 構文解析 48
3.3.2 型検査 49
3.3.3 仕様アニメーション 49
3.3.4 仕様スライシング 50
3.3.5 プログラムの自動生成 51
3.3.6 自動単体テスト 51

第 4 章 探索的仕様記述環境 ViennaTalk 53
4.1 アーキテクチャ 53
4.2 設計指針 55
4.3 ViennaTalk ライブラリ 63
4.3.1 Animation パッケージ 65
4.3.2 Browser パッケージ 65
4.3.3 Value パッケージ 69
4.3.4 Type パッケージ 72
4.3.5 Engine パッケージ 73
4.3.6 Parser パッケージ 75
4.3.7 トランスパイラ 77
4.4 VDMPad 86
4.4.1 アニメーションとの対話 88
4.4.2 VDM-SL の値の図的表現 91
4.4.3 実行時検査の設定 92
4.4.4 継続的単体テスト 93
4.5 Lively Walk-Through 94
4.6 Webly Walk-Through 97

第 5 章 ViennaTalk の評価 101
5.1 トランスパイラの評価 101
5.1.1 性能評価：素数列の生成 102
5.1.2 設計指針の観点からの評価 104
5.2 VDMPad の評価 106
5.3 ViennaTalk の評価 109

第 6 章　結論 **113**
6.1 学術的貢献 114
6.1.1 開発手法 114
6.1.2 ツール 115
6.1.3 仕様の編集 116
6.1.4 コミュニケーション 116
6.1.5 導入教育 116
6.2 形式手法 7 つの神話 117
6.3 まとめと今後の展望 119

謝辞 **121**

参考文献 **122**

参考文献 **122**

発表論文一覧 **129**

索 引 **132**

第1章　序論

本章では，本研究の背景として，形式仕様記述手法の概略を説明し，その解決すべき課題として形式仕様記述工程の生産性の向上を提示する．そして，研究の概観を目的と方法として示す．形式仕様記述工程の生産性に関連する既存研究を説明し，本研究の独自な点を示す．本研究で形式仕様記述言語として採用した VDM-SL の概要を紹介し，最後に，本論文の構成を示す．

1.1　背景

産業におけるソフトウェア開発では生産性および成果物としてのソフトウェアの品質がともに不足している．ソフトウェア開発において生産性と品質がしばしばトレードオフとして扱われるが，一方でソフトウェア開発は様々な工程の成果物の積み重ねであり，基礎となる成果物が信頼できるものとなるよう品質を向上させることが後工程の手戻りを抑制し，生産性の向上につながるという側面も持っている．仕様記述工程の成果物である仕様記述は，ソフトウェア開発の基礎となる記述であり，その品質の向上が求められている．

ソフトウェア開発の多くは問題解決の手段として行われる．解決を必要とする問題があり，その問題を解決するための仕組みとして，ソフトウェアシステムの開発が行われる．ソフトウェア開発の目的である問題解決を達成するためには，開発対象であるソフトウェアの仕様を定義し，その仕様を満たすソフトウェアが問題解決に資することを確認する必要がある．ソフトウェアの仕様を定義するためには，元の解くべき問題を理解する必要がある．ソフトウェア開発を開始した時点ではしばしば，元の解くべき問題への理解が不十分であったり，問題定義に曖昧さや抜けや矛盾が含まれていることがある．したがって，ソフトウェアの仕様を定義するためには元の解くべき問題に対して明確化を行う必要がある．

形式手法は数学的な裏付けを持つ道具立てを利用してソフトウェアを開発する手法の総称である．形式手法の一種である形式仕様記述は，ソフトウェアの機能仕様を数学的な裏付けを持つ記法で記述する手法であり，機能仕様から曖昧さを排除することができるとともに，ツールによって分析や検証を行うことができる．形式仕様記述は，航空管制など多くの人命に関わるため高信頼性が求められるソフトウェアの開発や，証券取引などの業務システムをはじめ様々な対象領域で適用されている [AL98, IPA13].

1.2　解決すべき課題

形式仕様は高信頼性システムの構築に多くの事例があるが，システムの信頼性だけでなく経済性にも効果があることが明らかにされている [FLS08, AL98, WLBF09, KN09, IPA13]．ソフトウェア開発への形式仕様の導入が開発プロセスに与える影響の 1 つに，フロントローディング効果がある．

ソフトウェア開発においては，各工程で混入した不具合の発見が遅くなるほど，コストに与える影響が大きいことが知られている [Tas02]．仕様記述も例外ではない．形式仕様の導入によって，開発の早期において顧客の要求に対する明確化と検証を行うことで，仕様記述工程のコストが増大するものの，全体としての開発コストを抑制するフロントローディング効果が報告されている．形式仕様記述工程において顧客が持っている問題とその解決としてのシステムを適切に抽象することで，仕様記述工程単体でのコストが高くなるがそれ以上に開発全体としてのコストが抑制される．また，形式仕様導入によるフロントローディング効果により，開発の工期の短縮や，開発されたソフトウェアの運用コストの低減に効果があったとする事例も報告されている [IPA13].

形式仕様を記述する工程にコストがかかることは，単に仕様記述言語の記号や文法などの技術的な困難だけが原因ではない．問題理解に関する困難が，形式仕様の利点と困難を理解する上で重要な鍵である．形式仕様を記述するためには，記述対象の本質を捉え，適切な抽象度で記述し，多くの利害関係者（ステークホルダー）が理解できるように記述を構成する必要がある．

記述対象の本質を捉えることは，試行錯誤を伴う困難な作業である．問題解決の手段として行われるソフトウェア開発では，仕様記述における記述対象の本質を得るためには，解決しようとしている問題への理解と

深い洞察を要する．しかし，ソフトウェア開発を開始した時点では，問題への理解が不十分であったり，問題定義に曖昧さや抜けや矛盾が含まれている．形式仕様を記述する過程において，問題への理解が不十分な点や，問題定義の曖昧さや抜けや矛盾が表面化する．したがって，形式仕様を記述するためには，記述を進める一方で，同時に問題への理解を深めるための作業が求められる．一度に完成度の高い記述を得ることは現実的ではなく，熟練した形式仕様記述者の作業からも，記述と問題への理解を深めつつ明確化を行い，記述の完成度を漸次的に高めていく様子が観察された．これは，問題解決において悪定義問題 [RW73] と呼ばれる，問題定義が初期段階では明確でない問題を解決する過程と合致する．

一方で，形式仕様を用いない開発ではしばしば，問題への不十分な理解や問題定義の曖昧さや抜けや矛盾が解消されないまま，設計や実装やテストなどの後工程に入り，そのために手戻りが発生し，開発全体のコストが増大する．形式仕様を用いることで，元の問題への理解が不十分な点や曖昧な点が表面化され，手戻り明確な理解を得るための作業が求められる．そのため，形式仕様記述工程そのものが多くの作業量を要するものになり，したがって，多くの工数がかかるようになっている．形式仕様記述を導入したソフトウェア開発のさらなる経済性の向上を工学的に図るためには，形式仕様記述工程での試行錯誤による漸次的なプロセスへの支援が求められる．形式仕様記述工程での作業での困難な問題を分析的に抽出し，困難を克服するための手法や道具立てを確立することによって，形式仕様記述工程の生産性を向上する必要がある．

1.3　研究の目的と方法

本研究は，形式仕様記述を導入することで必要性が強まる仕様記述の生産性の向上を目的として，形式仕様記述を探索的仕様記述とその精緻化に分離し，探索的仕様記述における困難を低減するための道具立ての設計指針を提案する．本研究ではまず，形式仕様記述工程の成果物である仕様記述に求められる品質を，悪定義問題と良定義問題の観点から分類し，悪定義問題となる品質を達成するための段階である探索的仕様記述と，良定義問題となる品質を達成するための段階である精緻化とに分け，特に系統的な解決が困難である悪定義問題である探索的仕様記述の特性を示し，支援するためのツールの設計指針を提案する．さらに，探索的仕様記述という工程とその支援のための設計指針をソフトウェアと

して具体化したものとして，探索的仕様記述を支援するツール群を提供する探索的仕様記述環境 ViennaTalk を開発し，ViennaTalk の探索的仕様記述工程における妥当性を評価する．

1.4　関連研究

形式仕様記述工程については様々な手法が提案されている．Fitzgerald らは VDM-SL による仕様記述として以下の手順を示した [FJ98].

1. 要求を読む
2. データ型や関数を抽出する
3. データ型の表現を決める
4. 関数の型を決める
5. データ型に不変条件を付けて型定義を完成させる
6. 関数定義を完成させる
7. 要求を読み返す

上記の手順は，個々の構成要素について一度に定義を完成させるのではなく，まずは仕様記述に必要な構成要素を列挙し，仕様記述全体について段階的に記述を加えていくことで，最終的に仕様記述を完成させ，要求を読み返すことで仕様記述を改善することを示している．この手順は VDM-SL だけでなく，1 つの記述対象の機能仕様について 1 つの抽象レベルでの記述を行う際の指針として有用である．

段階的詳細化 [ADL+91, Abr07] は，数学的な裏付けのある系統的な道具立てによって形式仕様記述から設計および実装を得る技術であり，高信頼性が要求される開発に多く適用されている．段階的詳細化は Z 記法 [Spi92] による開発や B-method[ADL+91] および event-B[Abr07] を含む形式手法に取り入れられている．抽象度の高い形式仕様記述から抽象度の低い形式仕様記述への変換は詳細化と呼ばれ，各段階の詳細化のために以下の 3 つの記述を行う．

- 抽象度の高い形式仕様記述でのデータと抽象度の低い形式仕様記述でのデータの対応関係を定義する

- 抽象度の高い形式仕様記述において正当な入力は全て抽象度の低い形式仕様記述においても正当な入力であることを証明する
- 入力に対する抽象度の低い仕様記述での出力は全て抽象度の高い仕様記述での出力に求められた性質を満たすことを証明する

段階的詳細化は，1 つの記述対象の機能仕様について，抽象機械 (Abstract Machine) と呼ばれる抽象度の高い形式仕様記述から数段階に分けて全体の抽象度を下げ，各ステップを証明により検証することで，成果物である設計または実装の正当性 を確保する手法である．段階的詳細化での設計または実装の正当性とは元の抽象機械に対する正当性であり，有用なシステムを開発するためには抽象機械を適切に定義し，詳細化の各ステップにおいて適切な構成要素を導入することが求められる．すなわち，段階的詳細化は概念を抽象度によって分類し，それぞれの概念を適切な抽象度の記述で導入することによって，仕様記述という大きな工程を分割して遂行する手法といえる．

仕様記述および開発プロセスの経済性に目を向けたアプローチとして，軽量形式手法 [Rob95] がある．ソフトウェア開発での軽量形式手法とは，ソフトウェア開発の全工程に形式手法を導入するのではなく，部分的な工程に形式手法を取り入れることを指す．例えば，仕様記述を形式仕様記述言語で記述し，そこから技術者が設計および実装を証明なしに行うことは，軽量形式手法の適用といえる．軽量形式手法においてもフロントローディング効果は産業界での事例で報告されている [IPA13, VTSHO12].

アジャイル開発は，ソフトウェア開発を比較的短期間の開発サイクルを回数多く繰り返すことで顧客の要望や開発の進行状況に対して敏捷に対応することを主眼とした開発手法の総称である [DNBM12]．主なアジャイル開発手法として，eXtreme Programming や Scrum が挙げられる．アジャイル開発ではタスクの分割と開発組織のコミュニケーションが重視され，仕様記述などの文書よりも主にプログラムコードを中心として，顧客からの要求の変化に俊敏に答えつつ，逐次的かつ継続的にソフトウェアを成長させる．自然言語による仕様記述から形式仕様記述を得る工程にアジャイル開発の要素を取り入れた XFM [SMSB05] など，アジャイル開発に形式仕様を取り入れる研究が盛んに行なわれている [BBB+09].

以上に示した通り，形式仕様を記述する基本的な手順や，形式仕様から設計および実装の導出や，形式仕様記述工程全体に敏捷性を持たせる手法が確立され産業界で適用されている．しかし，形式仕様記述工程における問題解決の質の違いに注目し，それぞれの問題解決での困難に応

じた手順や指針や支援の仕組みは提案されていない．本研究では形式仕様記述工程を，仕様記述が満たすべき品質特性に基づいて，探索的仕様記述と精緻化の 2 つの工程に分ける．探索的仕様記述とは，仕様記述の妥当性，機能項目の網羅性および実現可能性を確保するための工程であり，試行錯誤により記述対象への理解を深め，問題定義を明確にしつつ，形式仕様記述を進める工程である．精緻化は，機能定義の網羅性，無矛盾性，検証容易性および変更可能性を確保するための工程であり，系統的に作業を進めることが可能な工程である．本論文では，特に探索的仕様記述において行われる試行錯誤を分析し，それらの試行錯誤を支援するためのツールの設計指針を提案する．その上で，探索的仕様記述およびツールの設計指針をソフトウェアとして具体化したものとして，探索的仕様記述環境 ViennaTalk を開発する．そして，ViennaTalk に対して，探索的仕様記述工程における妥当性を評価する．

1.5　形式仕様記述言語 VDM-SL

形式仕様記述手法は形式手法の一種であり，数学的な裏付けを持つ記述言語により形式化された仕様を記述してシステムを開発する手法の総称である．形式仕様記述手法で利用される記述言語は形式仕様記述言語 と呼ばれ，主なものに VDM-SL[FJ98]，VDM++[FLM+05]，VDM-RT[Ver09]，Z 記法 [Spi92]，B-method[ADL+91] および event-B[Abr07]，Larch[Gut91]，OBJ[GT79] 等がある．これらの言語は UML 等の準形式的なモデル記法と比べ，構文および意味論が数学的に定義されており，記述の無矛盾性や正当性や網羅性を始めとする重要な特性を数学的に検証することが可能である点が特徴である．VDM-SL，VDM++，Z 記法はモデル指向型形式仕様記述言語と呼ばれ，記述対象のモデルとして扱うデータ構造や内部状態の変化および入出力を集合論や述語論理に基づいて記述する．Larch および OBJ は代数仕様記述言語と呼ばれ，記述対象の機能を抽象データ型によって代数的に定義する．本研究はモデル指向型形式仕様記述言語を対象とする．

上記のモデル指向型形式仕様記述言語のうち， VDM-SL，VDM++ および VDM-RT は VDM (Vienna Development Method) と呼ばれる手法で用いられる仕様記述言語であり，言語機能の多くが共通している．VDM-SL (VDM Specification Language) は データ型，定数，関数，状態，操作によって記述対象をモデル化する仕様記述言語である．VDM-SL は 1996 年

に ISO 規格化された．ISO 規格ではモジュール機構がなく，システム全体を大域的に記述したが，後にモジュール機構が導入され，大規模なシステムの記述にも適用することが可能になった．同じくモデル指向型形式仕様記述言語である Z 記法との大きな違いは，言語仕様において実行可能な言語機能についてのインタプリタの動作が定義されているなど仕様の実行が重視されている点が挙げられる．VDM-SL の実行可能なサブセットで記述された仕様は，インタプリタによりシミュレーション実行することが可能である．このシミュレーション実行を仕様アニメーションと呼ぶ．VDM++ は VDM-SL に対してクラス等のオブジェクト指向に関連した言語機能やスレッドを追加し，構文を C++ などのプログラミング言語に近付けたものであり，VDM-RT は VDM++ に CPU やデータバスなどのハードウェア構成やリアルタイム処理に関する言語機能を拡張したものである．本論文では形式仕様記述言語として VDM-SL を用いるが，VDM++を始めとする他の実行可能なサブセットを持つ形式的仕様記述言語にも適用可能であり，したがって議論は必ずしも VDM-SL に限定しない．

形式手法には以下の 3 つの適用レベルがあるとされている [BH95, 荒木 08]

レベル 0 : 形式仕様記述

レベル 1 : 形式的開発および検証

レベル 2 : 証明の自動検査

レベル 0 は形式仕様記述言語によって仕様を記述し，その形式仕様記述を用いてソフトウェアを開発する．証明は必ずしも行わない．これは軽量形式手法の一種といえる．レベル 1 は記述された形式仕様から系統的な方法でソフトウェアを導出し，証明による検証を行う．Z 記法や B-method や event-B における段階的詳細化を証明により検証を行う構成による正当性 (Correctness by Construction) はレベル 1 またはそれ以上であるといえる．レベル 2 はレベル 1 での系統的な開発に加え，レベル 1 で行われた証明に対して機械による証明の正しさの検査を行う．これは非常に高い信頼性を求められる開発に適用される．VDM-SL は上記いずれのレベルにおいても適用可能であるが，実行可能なサブセットを持つなど軽量形式手法としての適用に向いていることから，レベル 0 の開発においても多く用いられる．

1.6　本論文の構成

本論文の構成を以下に示す．第 2 章では，問題解決の観点から，形式仕様記述工程での探索的仕様記述工程と精緻化を定義する．まず，仕様記述の困難さの原因として，仕様を記述する作業が定式化困難な悪定義問題であることと，複数の専門領域にまたがった学際性を挙げる．形式仕様記述に求められる品質特性を列挙し，どの品質特性が仕様記述工程開始時に達成条件が明確化可能であるかを示すことで，形式仕様記述工程内での悪定義問題に属する問題解決作業と良定義問題に属する問題解決作業に分離する．また，簡単な仕様記述工程の過程の例を使って，探索的仕様記述では品質項目を達成するために具体的にどのようなタスクが遂行されるのかを説明する．

第 3 章では，探索的仕様記述を支援するためのツールを開発する上での設計指針を示す．まず，探索的仕様記述が失敗する要因を挙げ，それを防ぐためにツールが提供すべき機能を考察し，設計指針として提示する．そして，形式仕様記述に関係する技術について，ツールの設計指針がどのように適用されるかを議論する．

第 4 章では，探索的仕様記述の概念実証である探索的仕様記述支援環境 ViennaTalk について，アーキテクチャを説明した上で，第 3 章で示した設計指針がどのように実現されているかを説明する．ViennaTalk を構成するライブラリやツールについてもそれぞれ説明する．

第 5 章では ViennaTalk の評価として，ViennaTalk の構成要素であるトランスパイラと VDMPad に対する評価および ViennaTalk 全体に対する評価を示す．それらの評価を通して，ViennaTalk が探索的仕様記述を支援するツールの設計指針を正しく実装していることを確認した上で，探索的仕様記述を支援する上での設計指針の妥当性を評価する．

第 6 章では，それまでの議論をまとめた上で，本研究の学術上の貢献を示す．

第2章　探索的仕様記述

本章では，ソフトウェア開発を問題解決の観点で捉え，ソフトウェア開発プロセスにおける仕様記述の役割とその困難さを示す．加えて，仕様記述工程の別の困難さの要因として，仕様記述工程が持つ学際性を示す．これらを前提として，仕様記述の品質基準について悪定義問題の要素を抽出し，その解決には学際性を伴うコミュニケーションが重要であることを示し，形式仕様記述工程を探索的仕様記述と精緻化に分離する．そして，探索的仕様記述で行われる 2 種類の試行錯誤について説明し，探索的仕様記述で行われる具体的な作業のシナリオ例を自動販売機を例にして示す．

2.1　悪定義問題としてのソフトウェア開発

ソフトウェア開発は一種の問題解決である．問題解決は開始状態，ゴール条件，制約により定義され，問題の定式化が可能な良定義問題 (well-defined problem) と，定式化が困難な悪定義問題 (ill-defined problem, wicked problem) に分類される [RW73, Sim96, FR92, AEF97]．良定義問題とは問題定義が問題解決の開始時に明確にされている問題解決を指し，悪定義問題とは問題解決の開始時には問題定義を構成するゴール条件や制約が明確でない問題解決を指す．

例えば，ある駅 A から別の駅 B へ電車で移動するための最短経路を求める問題は，開始状態である「駅 A にいる」こと，ゴール条件である「駅 B に到着している」こと，制約である路線図が事前に定義されていることから，良定義問題である．一方で，週末の旅行を企画する作業は，明確なゴール条件も制約条件も与えられない悪定義問題であることが多い．ゴール条件である「何をすれば楽しめるのか」は明確な定式化は困難であり，また，制約条件として「週末」という一定の時間的制約があるが，具体的な制約条件は企画がある程度具体化されてから気付いたり同行者に相談したりすることで浮かび上がることがある．

一般に悪定義問題は解決のための計画を事前に策定することが困難であり，問題定義と問題解決が並行したプロセスにより解決を図るとされている [RW73]．良定義問題と悪定義問題への分類は，問題の困難さの度合いを区別するものではなく，問題解決に潜在する困難の質の違いであり，問題解決を支援するための方法や道具立ての違いである．

ソフトウェア開発の多くは不明確なゴール条件と制約に基づく悪定義問題である [Fis05]．ソフトウェア開発それ自体が問題解決であると同時に，ソフトウェア開発の多くは顧客やユーザが抱えている問題を解決するための手段として行われる．ソフトウェア開発の目的は，顧客がソフトウェアシステムの導入によって解決しようとしている問題（以後,「顧客が抱えている問題」と呼ぶ）の解決を進めることである．解決を必要とする問題があり，その問題を解決するための手段として，ソフトウェアシステムの開発が行われる．問題解決としてのソフトウェア開発のゴール条件は，ソフトウェアの仕様という形で明確化される．しかし，ソフトウェア開発を開始した時点ではしばしば，顧客が抱えている問題への理解が不十分であったり，問題定義に曖昧さや抜けや矛盾が含まれている．ソフトウェア開発を進めるためには，ソフトウェアの開発プロセスと並行して，顧客が抱えている問題が属する複数の専門的なドメインの知識を獲得するために顧客や各ドメインの専門家の意見や助言を得ながら，要求項目や仕様に潜んでいる曖昧さや抜けや矛盾を発見し，漸次的に理解を深めていく必要がある．すなわち，問題解決としてのソフトウェア開発は多くの場合，顧客が抱えている問題が開発開始時には明確に定義されていない，悪定義問題と見なすことができる．

ソフトウェア開発の多くは悪定義問題だが，ソフトウェア開発で行われる個々の作業は必ずしも悪定義問題ではない．例えば，数値データ列の標準偏差を求める関数を実装する作業について，入力データのフォーマットやデータサイズ，出力のデータ形式，標準偏差の定義式など，関数の仕様が曖昧さや抜けや矛盾なく定義されていれば，その関数を実装する作業は問題解決としてのゴール条件は明確であり，良定義問題と見なすことができる．

2.1.1　問題解決の視点から見た形式仕様の役割と効果

ソフトウェアの仕様記述工程とは，顧客が抱えている問題を理解し，顧客が抱えている問題を解決に導くための手段として求められているソフ

トウェアを定義する工程である．仕様記述中に元の問題に関して不明確な点がないか確認し，仕様が定義するシステムを実現することが元の問題解決に資することを確認し，その仕様がステークホルダーによる共通理解として共有されることが求められる．しかし，日本語などの自然言語によって仕様を記述する場合には，自然言語が持つ曖昧性のために仕様中に残された不明確な点が見逃されたり，読む人によって異なる解釈がされると，仕様の欠陥に起因する手戻りの原因や成果物としてのソフトウェアの欠陥につながる危険がある．

形式仕様とはそのソフトウェア開発の機能面におけるゴール条件を形式言語に基づく厳密な記法で記述したものである．数学的に定義された構文規則と意味論により記述とその解釈から曖昧性を排し，場合分けの網羅性や制約の整合性を系統的に検査することができることから，仕様から機能面での不明確な点を排除し，開発者の共通理解として共有することができる．形式仕様の適用がもたらすフロントローディング効果は，ソフトウェア開発を悪定義問題たらしめているゴールの不明確さに対して，ゴールを形式化し明確にする工程を形式仕様記述工程として分離し，以後の工程を良定義問題として計画的に遂行することを可能にした結果であると解釈することができる．すなわち形式仕様記述工程はソフトウェア開発を悪定義問題としている特性を解消して良定義問題に変換するための工程であり，ソフトウェア開発に系統的な開発手法を導入するための条件を整えるための工程である．形式仕様記述工程自体は悪定義問題として残されており，フロントローディング効果により開発コスト全体に占める仕様記述工程のコストが高まることから，ソフトウェア開発の生産性を高めるためには，形式仕様記述工程の生産性を高めることが重要である．

2.2　仕様記述工程の学際性

仕様記述工程は複数の専門化されたドメインにまたがった学際的な工程である [Fis94]．仕様を記述するにあたっては，仕様記述言語への理解および技量だけでなく，記述対象となるシステムが属するドメインに関する知識が要求される．加えて，仕様には実現可能性が求められることから，記述されたシステムを実現するための技術やその制約条件に関する知識も必要である．すなわち，システムを記述し分析するだけでなく，記述したシステムをどう実現するか，システムがどう使われるか，シス

テムに何が期待されているかを，開発組織が全体として理解する必要がある．このことから，仕様記述工程は仕様記述の技術に関する知識だけでなく対象領域に関する多様な専門的なドメインにまたがった知識が関わる学際的なプロセスである．

2.2.1 ドメイン知識に起因する学際性

記述された仕様は，システムによって解決しようとしている問題の解決に資するものであることが求められる．システムによって解決しようとしている問題を理解するためには，その問題が属する専門化されたドメイン知識が必要であることが多い．例えば証券取引を支援するシステムの開発では，問題の理解には証券取引に関する用語や業務フローや慣習や法規制などの知識が必要である．そして，取引注文を入力する実務の担当者と証券取引の法規制の専門家は，それぞれ別の特化された専門ドメインにまたがっている．それらのドメイン知識はさまざまな専門性を持つドメイン専門家に偏在していることが多く，顧客企業を代表する担当者だけでなく，顧客企業の各部門やソフトウェア開発を行う企業から仕様の妥当性や実現可能性の確認を受けるために，仕様を顧客・ドメイン専門家・プロジェクト管理者など多様な関係者に説明し，仕様に対するフィードバックを受け入れる必要がある．対象ドメイン自体が複数の特化された専門ドメインにまたがっている場合や，組織をまたがってドメイン専門家が分散している場合があり，円滑なコミュニケーションが求められる．

2.2.2 技術知識に起因する学際性

システムの機能仕様を記述する知識と技能は，それ自体が 1 つの専門性を持っている．また，記述された仕様は，設計・実装・UI デザイン・テスト・運用・品質管理など多くの工程およびプロセスにおいて参照されることから，それらの工程およびプロセスを実施する開発者からのフィードバックを受け入れ，仕様の修正を含む適切な対応をする必要がある．形式仕様を導入することで，仕様がソフトウェア開発の多くの工程で参照されることが報告されている [IPA13]．仕様記述とその妥当性や実現可能性の確認は，ステークホルダー間の合意形成プロセスである．開発ライフサイクルにおいて，関係者間のコミュニケーションを円滑にするため

に，仕様への理解が共有されることが求められる．しかしながら，それら専門領域を持つ関係者の多くは形式仕様の専門家ではない．このことからも，仕様記述工程は多くの技術的専門領域からのステークホルダーが関与する学際的なプロセスであり，形式仕様の専門家ではないステークホルダーとのコミュニケーションが重要である．

2.3　探索的仕様記述と精緻化

仕様記述工程は，ソフトウェア開発のゴール条件，すなわち開発の成果物であるソフトウェアが満たすべき条件を定義する工程である．形式仕様の記述という問題解決のゴールは適切な仕様を得ることであるが，どのような仕様が適切であるかを判定する具体的な条件は，仕様記述工程の開始時には明らかではない．仕様記述工程を進めていく過程の中で，顧客やドメイン専門家など多様な関係者とのコミュニケーションを通して，関連する様々なドメインの理解を深めることで，どのような仕様が適切であるかの条件が漸次的に明らかになっていく．したがって，仕様記述工程は悪定義問題であるといえる．ただし，仕様記述工程が全体として悪定義問題であるが，仕様記述工程で行われる全ての作業が必ずしも悪定義問題とはいえない．

本節では，仕様記述工程で行われる作業を，形式仕様に求められる品質特性によって，悪定義問題に属する作業と良定義問題に分類する．2.3.1 節では，IEEE 830-1998 で定義された要求仕様の品質特性に基づいて，形式仕様に求められる品質特性を示す．そして，2.3.2 節で，それぞれの形式仕様に求められる品質特性を獲得する作業が良定義問題であるか悪定義問題であるかを論じ，分類する．品質特性に基づいた良定義問題・悪定義問題の分類によって，形式仕様記述工程の初期段階である探索的仕様記述工程と，それに続く精緻化を定義する．

2.3.1　形式仕様の品質特性

本節では，IEEE 830-1998 で定義された要求仕様の品質特性に基づいて，形式仕様の品質特性を論じる．IEEE 830-1998 で定義された要求仕様の品質特性を 表 2.1に示す．

表 2.1に挙げた品質特性は形式手法の適用を前提とはしていない．ソフトウェアの要求仕様について，自然言語や疑似コードなどによる記述

表 2.1: IEEE 830-1998 によるソフトウェア要求仕様の品質特性

品質特性	説明
妥当性 (correctness)	仕様で記述された特性がソフトウェア開発の目的に資するものであること
非曖昧性 (unambiguity)	仕様記述の解釈が一意に定まること
完全性 (completeness)	機能，性能，設計制約，属性，外部インターフェイスに関する要求が全て記述されていること
	全ての入力に対するソフトウェアの応答が記述されていること
	仕様中の全ての図や表に対するラベル付けと参照，用語の定義と単位の定義がされていること
無矛盾性 (consistency)	仕様が自己矛盾していないこと
順位付け (ranked)	個々の要求に重要度と安定性のランク付けがされていること
検証容易性 (verifiability)	全ての要求が有限で費用効果性のある検証が可能であること
変更可能性 (modifiability)	完全性と無矛盾性を維持したまま容易に変更が可能であること
追跡可能性 (traceability)	各要求と関連する文書が参照できること

表 2.2: 形式仕様記述の品質特性

品質特性	説明
適法性 (well-formedness)	記述が言語の構文と意味論において適法であること
妥当性 (correctness)	仕様で記述された特性がソフトウェア開発の目的に資するものであること
実現可能性 (feasibility)	技術的経済的に実装を作成することが可能であること
機能項目の網羅性 (operational-completeness)	全ての必要な機能項目が定義されていること
機能定義の網羅性 (total-definition)	個々の機能項目が全ての入力や内部状態について 定義されていること
無矛盾性 (consistency)	仕様が自己矛盾していないこと
検証容易性 (verifiability)	全ての要求が有限で費用効果性のある検証が可能であること
変更可能性 (modifiability)	上記の品質特性を維持したまま容易に変更が可能であること

が満たすべき性質を列挙したものである．形式仕様記述言語を用いる場合， 表 2.1に挙げられた品質特性のいくつかは，そのまま適用することは適切ではない．例えば， 表 2.1に挙げられている「完全性」は，形式論理でいうところの完全性とは異なる概念である．また，非曖昧性は形式的な文法定義と意味定義がされている仕様記述言語による記述については自明のものであり，記述の品質として扱うのは適切ではない．

本論文では 表 2.2の品質特性により形式仕様記述を評価するものとする．すなわち，形式仕様記述工程とは， 表 2.2に示した品質基準それぞれについて求められる水準を満たす記述を作成する工程とする．

形式仕様を前提にした場合， 表 2.1のうち，非曖昧性およびラベル付け，用語および単位の定義に関する完全性は，記述言語の文法に則っていることを表す適法性 (well-formedness) と捉えることができる．また， 表 2.1の基準において品質の高い形式仕様を記述しても，それがソフトウェアとして実現可能でなければソフトウェアの仕様として有用ではない．仕様を満たすシステムの費用効果性のある実現が可能であることを，本論文

では実現可能性と呼ぶこととする．さらに， 表 2.1での「完全性」は形式論理でいうところの完全性とは異なる概念であり，網羅性と呼ぶのが適切と考える．本論文では， 表 2.1での機能，属性，外部インターフェイスに関する網羅性を機能項目の網羅性，入力に対する完全性を機能定義の網羅性と呼び，区別することとする．妥当性，無矛盾性，検証容易性および変更可能性は形式仕様においても品質特性として有効である．

表 2.1での完全性のうち性能，設計制約に関する要求は，本論文ではソフトウェアの機能仕様としての形式仕様記述の対象外とする．追跡可能性は記述作業そのものではなく開発ライフサイクル全体で実現するものであるから，本論文では考察の対象外とする．ソフトウェアの機能仕様としての形式仕様の記述においては順位付けは記述前に解決されているものとして，本論文では考察の対象外とする．

2.3.2　品質特性の分類による工程の分割

前節で示した形式仕様の品質特性のうち，適法性は形式仕様ではツールによる自動検査が可能である．すなわち，適法性のゴール条件は形式仕様記述言語の構文定義と意味論により明確に定義されていることから，適法性の獲得は良定義問題である．適法性の獲得のためのコストは，導入教育において十分なトレーニングを行い仕様記述者の能力を養うことで解決される問題である．事例においても数週間から数ヶ月の学習期間で VDM-SL の文法を学び，適法な仕様を記述することが可能であることが示されている [IPA13].

実現可能性および妥当性および機能項目の網羅性は仕様記述および記述言語のみでは決まらない．実現可能性は実現のための後工程の技術に依存し，妥当性は対象ドメインや利用条件，運用条件などに依存する．これらは境界が不明瞭な条件に依存しているため，妥当性の獲得は明確なゴールが定められていない悪定義問題である．

機能定義の網羅性および無矛盾性はその条件が記述言語の構文定義および意味論により定式化されるため良定義問題である．検証可能性および変更可能性は，同じ特性を表現した複数の可能な表現から，抽象度や記述の粒度や名前付け，表現スタイルが適切であるものを選択することで獲得する．適切さの基準が明示されていない場合には悪定義問題であるが，適切な記述規約を定めることによって良定義問題として扱うことができるため，本論文では良定義問題として扱う．

悪定義問題である，妥当性および機能項目の網羅性および実現可能性の獲得は，試行錯誤を通して記述対象への理解を深めることが求められる．妥当性を獲得するためには，顧客やドメイン専門家とのコミュニケーションが必要である．システムの仕様記述が進むにしたがって，仕様記述者だけでなく顧客やドメイン専門家もそのシステム特有の問題を理解し，開発開始時にはなかった視点が生まれ，妥当性に関して新しい条件が加わり，新たな機能項目の必要性を認識し，実現可能性について考察すべき点が発見される．すなわち，多様なステークホルダーとのコミュニケーションを通して問題領域を探索することが，妥当性および機能項目の網羅性および実現可能性の獲得には求められる．この，妥当性および機能項目の網羅性および実現可能性の獲得を探索的仕様記述と呼ぶこととし，その工程を探索的仕様記述工程と呼ぶ．

探索的仕様記述の目的である実現可能性と妥当性はともに仕様記述者以外のステークホルダーの知識を必要とする．そのため，悪定義問題の解決に必要とされる問題定義と問題解決を並行させたプロセスとして，問題定義であるステークホルダーとのコミュニケーションを行うプロセスと，問題解決である仕様を記述するプロセスを並行して進めることが求められる．仕様記述者はその時点での最善の知識に基づいて仕様を記述し，その仕様記述を分析することでドメインに対する理解を深め，さらに仕様記述に対するドメイン専門家や他のステークホルダーからのフィードバックからもドメインについて学び，得られた知見によって仕様記述を改善する．すなわち，構築することによって対象について学習し，それを次の仕様記述に反映させる (reflection in action)[Sch84].

一方，良定義問題である，機能定義の網羅性および無矛盾性および検証容易性および変更可能性は，仕様記述者が形式手法の技術によって獲得することができる．仕様記述をより厳密にし，仕様記述として機能的に構成するための作業であることから，機能定義の網羅性および無矛盾性および検証容易性および変更可能性の獲得を精緻化と呼ぶこととし，その工程を精緻化工程と呼ぶ．精緻化は，仕様記述に対して系統的計画的に進めることができる．

一般に，形式仕様記述工程の初期段階では探索的仕様記述が行われ，妥当性および実現可能性および機能項目の網羅性が確保されたことへの確信が深まるにつれて，精緻化が主な作業となることが望ましい．仮に，逆に精緻化の後で探索的仕様記述を行った場合，妥当でない仕様の 1 つの機能の定義の網羅性を確保しても，その仕様の妥当性を確保するために

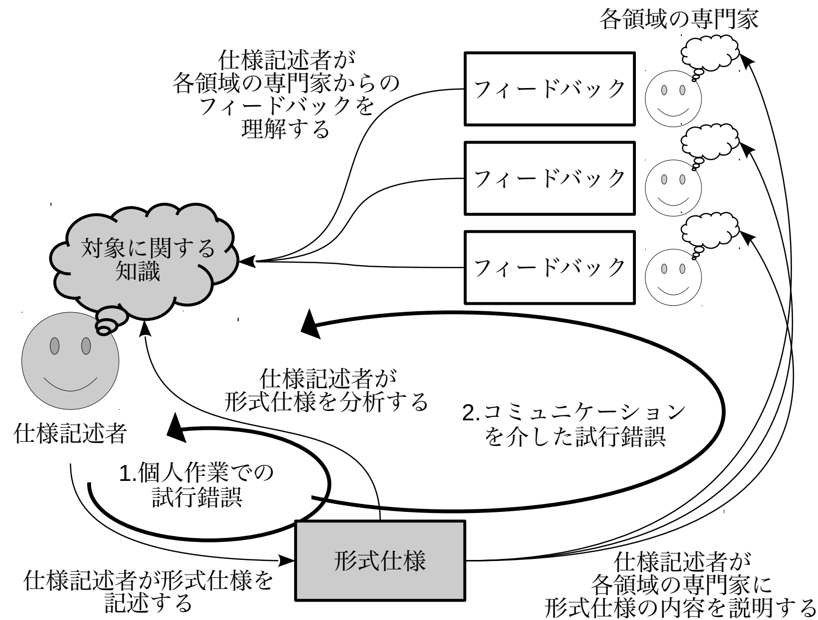

図 2.1: 探索的仕様記述における試行錯誤のプロセス

機能の再定義が行われれば，再び機能定義の網羅性を確保しなければならず，手戻りの原因となる．探索的仕様記述によって妥当性が確保された仕様記述に対して，精緻化で機能定義の網羅性を確保するよう修正した場合には，修正された仕様記述の妥当性は損なわれない．したがって，形式仕様記述工程全体のうち初期段階においては探索的仕様記述を行い，最終段階においては精緻化を主な作業とすることが効率的であると考えられる．

2.4 探索的仕様記述における試行錯誤

探索的仕様記述は，形式仕様記述工程における品質特性のうち，悪定義問題である品質特性を獲得する工程である．一般に悪定義問題の解決には，問題への理解と問題の解決が並行する探索的なプロセスが行われる．したがって，多くの試行錯誤が発生する．

図 2.1に探索的仕様記述における試行錯誤のプロセスを示す．仕様記述者は自分が持っている対象に関する知識を形式仕様として記述する．仕様記述者は自分が記述した形式仕様を読み，またはツールにより分析を

行い，その記述が意味するところを理解することで，対象に関する知識を深める．例えば，仕様に対して静的型検査を行い，記述内容の論理的不整合を発見することで，対象についての自分の誤解に気付き，不整合を解消することで対象に対するより正確な知識を得ることができる．また，仕様をアニメーション実行することで，記述対象の具体的な振る舞いを確認し，対象についてより深く理解することができる．

さらに，記述された形式仕様を各領域の専門家に説明し，フィードバックを得て，そのフィードバックを理解することで，対象に関する知識をより深めることができる．例えば，対象ドメインの専門家により，形式仕様が対象ドメインに適合しているかどうか，適合しない部分がある場合には，ドメインでの正しい知識をフィードバックから得ることで，仕様記述者は対象に関する知識を獲得する．また，UI デザイナに形式仕様が想定する機能セットを説明することで，UI デザイナが想定していたユーザインタラクションと形式仕様が適合するかどうかを確認し，不適合な場合には両者による議論を通して共通理解を構築し，改善のための合意を形成する必要がある．これらの共通理解や合意により，仕様記述者が持つ対象に関する知識が深まっていく．仕様記述者と各領域の専門家の間の共通理解を構築していく上で，形式仕様が中心に位置している．形式仕様が持つ構文的および意味的な厳密さが，異なる専門性を持つステークホルダーの知識を統合するための基盤である．

図 2.1で示されたプロセスには，仕様記述者および形式仕様に関するループが 2 つある．1 つ目のループは仕様記述者が個人で行う作業を表すループである．仕様記述者が形式仕様を記述し，その記述を理解して知識を深めることで，1 つのループを形成している．例えば，仕様記述者が VDM-SL で仕様を記述したとする．その仕様を型検査器で検査し，型検査が型エラーを発見し，仕様記述者は型エラーのメッセージを読むことで，仕様についての不整合を発見する．その不整合を見て，仕様記述者は自分の理解の中に概念的な不整合があったことを発見し，VDM-SL 仕様を修正する．修正された VDM-SL 仕様は再度型検査器にかけられ，型エラーが修正されたことを確認する．型エラーが修正されたことで，仕様記述者は自分の理解の中にあった不整合を解消することができる．これらは個人に閉じた作業であり，試行錯誤を通じて対象への理解を深め，仕様を記述する．この試行錯誤を，個人作業での試行錯誤と呼ぶ．

もう 1 つのループは，様々な領域の専門家とのコミュニケーションを介したループである．仕様記述者が形式仕様を記述し，その記述の内容

表 2.3: 個人作業での試行錯誤とコミュニケーションを介した試行錯誤の特徴

	個人作業での試行錯誤	コミュニケーションを介した試行錯誤
遂行者	記述者個人	記述者とステークホルダー
必要な技術	形式手法の専門的技術	学際的なコミュニケーション
成果物	仕様記述者の理解に則った形式仕様	妥当性や実現可能性について合意された形式仕様

を各領域の専門家に説明する．各領域の専門家は，説明された内容を自分の知識に照らし合わせて，仕様記述者にフィードバックする．仕様記述者はフィードバックを理解することで，開発対象に関する知識を深める．このループは個人に閉じておらず，コミュニケーションを介した試行錯誤を通じて対象への理解を深め，仕様を記述する．この試行錯誤を，コミュニケーションを介した試行錯誤と呼ぶ.

表 2.3に個人作業での試行錯誤とコミュニケーションを介した試行錯誤の概要を示す．個人作業での試行錯誤は，仕様記述者が持っている記述対象に関する理解を仕様記述言語で記述するタスクを遂行する時の試行錯誤である．仕様記述者が記述対象に関する理解を仕様記述言語による表現として記述する時，どの言語機能によってどのような定義をすることが適切かを理解する必要がある．ある概念を仕様記述言語で表現する方法が複数ある場合，その選択が関連する他の概念の表現の選択肢に影響を与える場合がある．仕様記述者の技量にも依るが，一般に言語機能選択の波及効果を予測し適切な記述法を選択することは困難であり，一定の試行錯誤を要する．試行錯誤の成果として，仕様記述者が理解している概念について仕様記述言語で表現した記述が得られる.

一方，コミュニケーションを介した試行錯誤は，仕様記述者が様々な専門領域にまたがる学際的な知見に基づいた深い理解を得るための試行錯誤である．より深い理解に基づく仕様を記述するためには，まずは暫定的な仕様を記述し，ドメイン専門家や技術者など他のステークホルダーにその暫定的な仕様を説明し，妥当性や実現可能性についてのフィードバックを得ることで，対象に関する学際的な知識を獲得する必要がある．暫定的な仕様を説明する時，一般にドメイン専門家や他のステークホルダーは形式仕様記述言語を読解するための知識や技量を持っているとは限らない．相手に形式仕様を理解できるように説明し，また，相手の説

明を理解し，形式仕様に反映させる必要がある．

2.5　探索的仕様記述の例：自動販売機

本節では，自動販売機に付属している金銭管理モジュールの仕様を例に，探索的仕様記述で行われる作業を示す．本節で示す過程は実際の開発事例に基づくものではなく，あくまで過程の中で行われる試行錯誤を説明することを目的とした，仮説的な記述過程である．

以下に本節の構成を説明する．まず，自動販売機の金銭管理モジュールに関する要求の概要を説明する．要求は日本語で記述されている．次に，暫定的な仕様を記述する．要求に現れた概念をどの言語機能で表現するかを試行錯誤で確認しながら，暫定的な仕様案を作成する．そして，暫定的な形式仕様をステークホルダーに示して，フィードバックを得る．ステークホルダーから得られたフィードバックに基づいて仕様を修正することで，妥当かつ実現可能な仕様を作成する．

2.5.1　要求の概要

ある仕様記述者が自動販売機の開発に参加している．仕様記述者は自動販売機の開発に初めて携わる．仕様記述の対象は，金銭管理モジュールと呼ばれる，缶飲料の自動販売機に内蔵されているコインの投入と残高管理と釣り銭の支払いを行うコンポーネントである．自動販売機には，他に販売ボタンモジュールや商品管理モジュールや商品排出モジュールやそれらのモジュールを統括する中央制御モジュールがある．顧客とのやりとりや各モジュールを担当する技術者間の調整は，アーキテクトと呼ばれる自動販売機の開発に熟練した技術者が担当する．図 2.2にアーキテクトから説明された自動販売機の金銭管理モジュールの概要を示す．

簡単のため，金銭管理モジュールが扱うのは百円硬貨，五十円硬貨および十円硬貨のみとする．それぞれの硬貨は投入されると機械が自動的に種別を判定し，割り込み処理を通して本モジュールのルーチンを実行する．アーキテクトからの指示により，割り込みハンドラの実装を簡単にするために，コイン投入で実行される操作として，硬貨の種類ごとに対応する引数なし返り値なしの操作を外部に公開することが求められている．

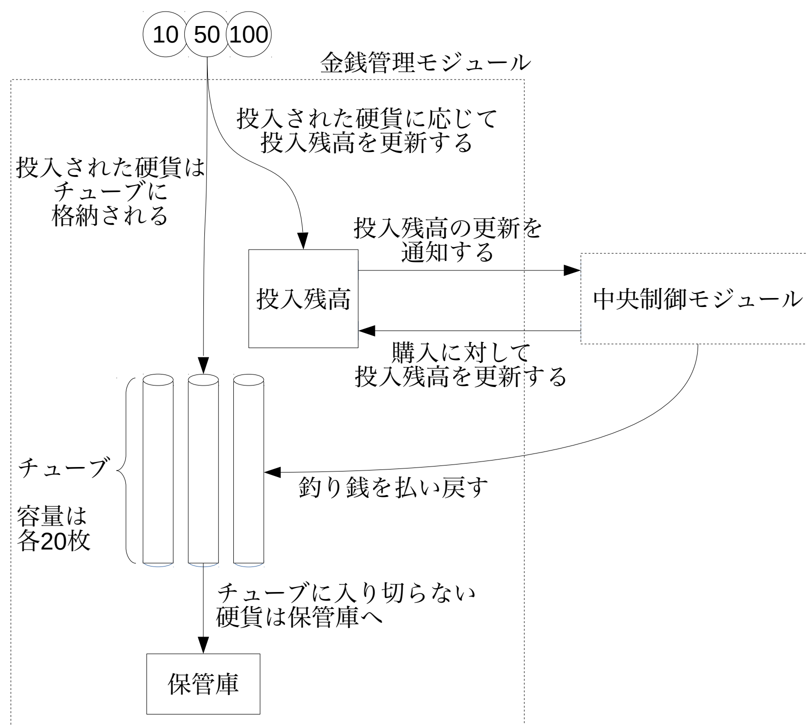

図 2.2: 自動販売機の金銭管理モジュールの概要

投入されたお金の残高の管理も本モジュールの責任で行う．自動販売機の中央制御モジュールから，缶飲料が購入されるごとに決済を行う操作が実行される．缶飲料の値段つまり決済の金額は操作の引数として渡される．金銭管理モジュールが管理する残高が変更されるたびに，販売ボタンの点灯状態の更新をする必要がある．販売ボタンの点灯状態の管理は中央制御モジュールが行う．金銭管理モジュールは，残高が変更されるたびに，更新処理のトリガーとして中央制御モジュールの「残高が変わった」操作を呼び出す必要がある．また，販売ボタンの「点灯」処理のために，現在の残高を取得するための操作を本モジュールが提供する．

釣り銭を払い戻すための精算ボタンも本モジュールに取り付けられている．硬貨の投入と同様，割り込みハンドラを通して処理されるため，精算ボタンが押された時に実行される引数なしの操作を提供するよう，アーキテクトから求められている．また，硬貨の排出も，各硬貨に対応する引数なし返り値なしの排出操作を仮に用意して利用するよう求められている．

釣り銭はそれまでに投入された硬貨から払い戻す．硬貨の種類ごとに 1 本ずつチューブがあり，投入された硬貨はそのチューブに格納される．ただし，それぞれのチューブには容量がある．チューブの型式でさまざまな容量があるが，今回の製品では 20 枚の容量を持つチューブが採用される予定である．容量をオーバーした枚数が投入された場合には，20 枚を超えた分の硬貨を「保管庫」と呼ばれる大きな容器に入れる．「保管庫」に入った硬貨は釣り銭として利用することはできない．

以上が金銭管理モジュールへの要求事項である．

2.5.2　暫定的な仕様を記述する

まずは前節で示された要求を基に探索的仕様記述を行う．要求は仕様記述のゴール条件を示すものではなく，スタート状態である．探索的仕様記述のゴール条件は，妥当性と機能項目の網羅性と実現可能性が確認された VDM-SL 仕様を得ることである．示された要求をそのまま形式仕様記述言語に翻訳しても，必ずしもそれが妥当性を持ち，機能項目が網羅され，実現可能であるとは限らない．多くの場合，要求には記述に曖昧な点が残っていたり，あるいは明示的に記述されていないドメイン知識が潜在している．それらの曖昧な点や明示されていない点を明確にし，適切なステークホルダーと議論することで，明確な定義をしていく必要

```
module 金銭管理
imports from 中央制御部 operations 残高が変わった;
exports all
definitions
```

ソースコード 2.1: 暫定的なモジュール宣言部

がある．具体的なゴール条件は探索的仕様記述を通して試行錯誤を伴いながら徐々に明らかになる．

形式仕様を記述する時に，最初から全てを完備した完全な仕様を記述することは非常に困難である．探索的仕様記述では，まずは不完全ながらも仕様記述言語の文法に適った暫定的な仕様を記述し，記述を進めながら暫定的な仕様に基づいて関係者と議論し，妥当性や機能項目の網羅性や実現可能性が確保された仕様を逐次改善的に記述する．本節では，その議論の基礎となる暫定的な仕様の記述を説明する．

モジュールとその暫定的な外部インターフェイスを宣言する

要求から抽出されるモジュールとして，金銭管理モジュールと中央制御モジュールが挙げられる．また，中央制御モジュールは「残高が変わった」操作を提供し，本モジュールがそれを呼び出すことが決められている．そこで，暫定的にモジュールの宣言部をソースコード 2.1のように記述することができる．

型，定数および状態定義を暫定的に定義する

VDM-SL では，システムを型，定数，関数，状態および操作によって定義する．型はデータ型，定数は特定の値に名前をつけたものである．定数には必要に応じて静的型を宣言することができる．関数は静的に型付けされた副作用のない関数である．関数には必要に応じて事前条件や事後条件を宣言することができる．状態はいくつかの状態変数と初期値の定義および不変条件から成る．操作は，状態を参照すること，および，状態を更新することができる．また，操作は関数と同様に引数および返り値や，事前条件および事後条件を持つことができる．一般に操作は副作用を持つことがあるため，不変条件や事前条件や事後条件内で呼び出すことが禁止されている．

```
types
   硬貨 = <百円硬貨>| <五十円硬貨>| <十円硬貨>;
   枚数 = nat;
   現金 = map 硬貨 to 枚数;
   金額 = nat;

values
   硬貨の金額
     = {<百円硬貨> |-> 100, <五十円硬貨> |-> 50,
       <十円硬貨> |-> 10};
   チューブ容量 = 20;

state State of
  チューブ : 現金
  投入残高 : 金額
inv mk_State(チューブ, -) ==
  forall この硬貨 in set dom チューブ &
    チューブ(この硬貨) <= チューブ容量  -- 物理制約
init s == s = mk_State(
  {<百円硬貨> |-> 0, <五十円硬貨> |-> 0, <十円硬貨> |-> 0},
   0)
end
```

ソースコード 2.2: 型，定数と状態の暫定的な定義

VDM-SL 仕様の各モジュール内はいくつかのセクションから成っている．型は **types** セクションで，定数は **values** セクションで，関数は **functions** セクションで，状態は **state** セクションで，操作は **operations** セクションで定義される．**state** セクション以外のセクションでは，1 つの種類のセクションがモジュール内に複数あっても構わない．1 つのセクション内で複数の定義を記述することができる．

要求に現れる金銭についての概念をまとめると，本モジュールに直接関係する概念として，硬貨，十円硬貨，五十円硬貨，百円硬貨，枚数，残高，チューブ，容量，精算ボタン，釣り銭，排出，が要求に出現している．これらの概念を VDM-SL の言語機能から適切なものを選択して，定義する．VDM-SL による暫定的な定義のうち，**types** セクション，**values** セクションおよび **state** セクションを，ソースコード 2.2に示す．

ソースコード 2.2の **types** セクションでは,「硬貨」型が引用型と直和型によって定義され，**values** セクションでは，定数として「硬貨の金額」が硬貨から金額への写像として定義されている．しかし,「硬貨」型を引用型と直和型によって定義することが適切であるかどうかは自明ではない．例えば「硬貨」と「金額」を型として表現する上で，ソースコード 2.2での定義以外には以下の定義が考えられる．

```
types
  金額 = nat;
  硬貨 :: 金額:nat;
```

仮に「硬貨」をレコード型として定義すると，具体的な硬貨の種類を示すことなく，任意の金額の硬貨を扱うことができる.「硬貨の金額」定数として定義された，硬貨から金額への写像は必要ない．このように，ある概念をどの言語機能によって実現するかによって，他の概念をどのように定義するかが変化する．

レコード型による「硬貨」型の定義では，後に五百円硬貨を扱う必要が発生した場合には，`mk_硬貨(500)` として型定義の変更なしに記述できる．一方で，`mk_硬貨(123)` という実在しない硬貨も記述することが許される．どちらの定義がより適切かは，この時点では自明ではない．その定義を使って，全体の仕様を適切な抽象度で一貫性のある記述ができるように定義を選択する必要があるが，ある定義が適切かどうかは他の概念の記述がされるまでは不確定である．他の概念や機能を記述して，仕様全体について記述として適切であるかを評価する必要がある．この金

銭管理モジュールの記述では，ソースコード 2.2に示したように，引用型と直和型を使って定義した．以上のように，概念ごとの言語機能の選択には，図 2.1での個人作業での試行錯誤が発生する．以下，金銭管理モジュールの暫定的な仕様を記述する具体的な過程を示すが，このような言語機能の選択に関する試行錯誤は省略する．

「現金」型は，どの硬貨が何枚あるかを表現する型である．例えば「釣り銭として五十円硬貨を 1 枚と十円硬貨 3 枚」は,「現金」型の {<五十円硬貨> |-> 1, <十円硬貨> |-> 3} という値で表現され，その金銭価値は「金額」型の 80 という値で表現される．この定義も暫定的な定義であり，適切な言語機能を選択するために個人作業での試行錯誤を行う余地がある．ソースコード 2.2での「現金」型の定義では，{<五十円硬貨> |-> 1, <十円硬貨> |-> 3} だけでなく，{<百円硬貨> |-> 0, <五十円硬貨> |-> 1, <十円硬貨> |-> 3} も現実世界における「五十円硬貨を 1 枚と十円硬貨 3 枚」を指し示す.「枚数」型を nat1 型にするか，または,「現金」型に「inv この現金 == 0 not in set rng この現金」と不変条件を宣言することで，{<五十円硬貨> |-> 1, <十円硬貨> |-> 3} のみが「五十円硬貨を 1 枚と十円硬貨 3 枚」を指し示すように修正することが考えられる．あるいは,「現金」型に「inv この現金 == dom この現金 = {<百円硬貨>, <五十円硬貨>, <十円硬貨>}」と不変条件を宣言することで，{<百円硬貨> |-> 0, <五十円硬貨> |-> 1, <十円硬貨> |-> 3} のみが「五十円硬貨を 1 枚と十円硬貨 3 枚」を指し示すように修正することも考えられる．どの定義が適切かは，この段階ではまだ不明であり，後でドメイン知識としてどれが適切か判明する可能性や，精緻化の段階において仕様全体の整合性から適切な定義を選択する可能性がある．現段階では適切な表現がどれかは決定できないため，ソースコード 2.2での暫定的な定義のままとする．

残高とチューブ内の硬貨は処理により変化するものであるから，暫定的に状態変数として表現する．チューブの容量については，記述上の可能な選択肢として，(1) 枚数を定義する定数，(2) 硬貨ごとの枚数を定義する定数，(3) 枚数を定義する変数，(4) 硬貨ごとの枚数を定義する変数，が考えられる．どれが最も適切な選択であるかは，ハードウェア設計でどの範囲の変更可能性があるかによって決まる．まずは **values** セクションにあるように，暫定的に (1) の枚数を定数「チューブ容量」として定義することを選択し，他の関係者との議論を通して最終的に決定する．他の関係者との議論を介しての試行錯誤も探索的仕様記述で行われる試行

錯誤の 1 つである．

state セクションでは,「現金」型の変数「チューブ」と「金額」型の変数「投入残高」が定義されている．**inv** 節で記述された不変条件は，要求にあるチューブの容量を表現したものである．不変条件をはじめとする表明は，探索的仕様記述でのコミュニケーションを介した試行錯誤において，重要な役割を持っている．不変条件として明示的に表現されていることで，妥当性の確認として，このモジュールへの要求事項の意図と合致しているかどうか仕様記述者と要求者の間で議論することができる．議論の時に説明しやすいように，物理制約である旨がコメントとして明示されている．また，証明課題や仕様アニメーションによって，チューブ容量の物理的制約が守られていることを確認することができる．**init** 節では，チューブはそれぞれの硬貨用のチューブは空であり，投入残高は 0 が，初期状態として定義されている．

操作を暫定的に定義する

最後にこのモジュールが提供する操作を記述する．アーキテクトから硬貨投入と硬貨排出について指定された一連の操作を暫定的に定義した記述をソースコード 2.3に示す.「十円硬貨が投入された」,「五十円硬貨が投入された」,「百円硬貨が投入された」の 3 つの操作を定義するために，硬貨の種類を抽象して，硬貨を引数とする「投入する」操作を定義する．「投入する」操作の定義は後に示す.

「精算ボタンが押された」操作は，投入残高を返金した後で,「中央制御部」モジュールの「残高が変わった」操作を呼び出すようアーキテクトから求められている．ただし，投入残高の金額から，チューブ内の硬貨の枚数に応じて，どの硬貨を何枚返却するかを決定する必要がある．それを「釣り銭」操作としてソースコード 2.4で定義する．

ソースコード 2.4では操作の定義に `pure` が前置されている．`pure` はその操作が副作用を起こさないことを示す宣言である．`pure` が宣言されていない操作は副作用を起こす可能性があるため，事前条件，事後条件，不変条件および関数定義の中から呼びだすことが許されない．`pure` を宣言することで，事前条件，事後条件，不変条件および関数定義の中から呼び出すことができる操作を定義することができる．

与えられた金額が常にチューブ内から返金可能とは限らない.「釣り銭」操作の返り値は現金のオプション型として，チューブ内の硬貨でその金

```
operations
  百円硬貨が投入された : () ==> ()
  百円硬貨が投入された() == 投入する(<百円硬貨>);

  五十円硬貨が投入された : () ==> ()
  五十円硬貨が投入された() == 投入する(<五十円硬貨>);

  十円硬貨が投入された : () ==> ()
  十円硬貨が投入された() == 投入する(<十円硬貨>);

  精算ボタンが押された : () ==> ()
  精算ボタンが押された() ==
    let
      この釣り銭:[現金] = 釣り銭(投入残高)
    in
      if この釣り銭 <> nil
      then
        (返金する(この釣り銭);
        投入残高 := 0;
        中央制御部‘残高が変わった());

  十円硬貨を排出する : () ==> ()
  十円硬貨を排出する() == skip;

  五十円硬貨を排出する : () ==> ()
  五十円硬貨を排出する() == skip;

  百円硬貨を排出する : () ==> ()
  百円硬貨を排出する() == skip;
```

ソースコード 2.3: 硬貨投入の暫定的な操作定義

```
  pure 釣り銭 : 金額 ==> [現金]
  釣り銭(この金額) ==
    (dcl 残り:金額 := この金額,
      釣り銭の硬貨と枚数:map 硬貨 to 枚数 := {|->};
    for この硬貨 in 釣り銭優先順位
    do
      if
        この硬貨 in set dom チューブ
      then
        let
          この硬貨の枚数 : 枚数 =
            min(チューブ(この硬貨),
              残り div 硬貨の金額(この硬貨))
        in
          (釣り銭の硬貨と枚数(この硬貨) := この硬貨の枚数;
          残り :=
            残り - 硬貨の金額(この硬貨)  * この硬貨の枚数);
    return if 残り = 0 then 釣り銭の硬貨と枚数 else nil);
```

ソースコード 2.4: 釣り銭操作の暫定的な定義

額を返金できない場合には **nil** を返り値とする.「釣り銭」操作の返り値が **nil** の場合には「精算ボタンが押された」操作は返金しない.

アーキテクトから本モジュールで提供するよう求められた 2 つの操作の定義をソースコード 2.5に示す.「決済する」操作では事前条件として引数として与えられた金額が残高以内であることを求めている．この事前条件は明示しなくても操作の本体「投入残高 := 投入残高 - この金額」から導出することが可能である．変数「投入残高」の型は「金額」型であり，実体としては nat 型である．したがって,「投入残高 - この金額 >= 0」であることが求められることから,「この金額 <= 投入残高」で

```
operations
  pure 残高 : () ==> 金額
  残高() == return 投入残高;

  決済する : 金額 ==> ()
  決済する(この金額) == 投入残高 := 投入残高 - この金額
  pre この金額 <= 投入残高;
```

ソースコード 2.5: 残高操作と決済操作の暫定的な定義

ある必要がある．機械的に導出可能であっても，この事前条件を明示することは，コミュニケーションを介した試行錯誤に有用である．操作の事前条件を仕様に明示することで，ステークホルダーと議論し妥当性や実現可能性を確認できるとともに，証明責務や仕様アニメーションによってその特性が守られていることを確認することができる．

```
functions
  min : nat * nat -> nat
  min(x, y) == if x <= y then x else y;

operations
  投入する : 硬貨 ==> ()
  投入する(この硬貨) ==
    (チューブ(この硬貨)
      := min(チューブ(この硬貨)+1, チューブ容量);
    投入残高 := 投入残高 + 硬貨の金額(この硬貨);
    中央制御部`残高が変わった());
```

ソースコード 2.6: 投入する操作の暫定的な定義

次に，硬貨を投入するための「投入する」操作の定義をソースコード 2.6に示す．硬貨の投入ではチューブ内の硬貨の枚数がチューブ容量を超えないように，min 関数で上限を設けている．投入残高も硬貨の種類に応じて更新する．上記の状態更新を行った後で，アーキテクトから要求された通り，中央制御モジュールの「残高が変わった」操作を呼び出している．

最後に,「精算ボタンが押された」操作の定義で必要な,「返金する」操作を定義する．これは，硬貨ごとに何枚返却するかが指定され，アーキテクトから求められた硬貨排出の操作の呼び出しを行うとともに，チューブ内の硬貨を更新する．

以上で， 図 2.1の個人作業での試行錯誤の結果，自動販売機の金銭管理モジュールの暫定的な仕様が定義された．この時点では，要求として記述された内容が，VDM-SL の文法に適法な仕様として記述された．個人作業での試行錯誤は，最適ではないながらも，仕様全体として適法な記述が得られたことで，最低限の解決がされている．VDM-SL の文法に適法な記述が得られたことで，次の段階では，その仕様の具体的な記述に基づいて様々なステークホルダーと議論することで， 図 2.1のコミュニケーションを介した試行錯誤に取り組む．

```
values
    釣り銭優先順位 = [<百円硬貨>, <五十円硬貨>, <十円硬貨>];

operations
  返金する : 現金 ==> ()
  返金する(この現金) ==
    (for all この硬貨 in set dom この現金 do
      for i = 1 to  この現金(この硬貨)
      do
        cases この硬貨:
          <十円硬貨> -> 十円硬貨を排出する(),
          <五十円硬貨> -> 五十円硬貨を排出する(),
          <百円硬貨> -> 百円硬貨を排出する()
          end;
    チューブ
      := ({この硬貨 |-> チューブ(この硬貨)-この現金(この硬貨)
        | この硬貨 in set dom この現金}
        munion dom この現金 <-: チューブ))
  pre
    dom この現金 subset dom チューブ
    and (forall この硬貨 in set dom この現金 &
         チューブ(この硬貨) >= この現金(この硬貨));
end 金銭管理
```

ソースコード 2.7: 返金する操作の暫定的定義

```
operations
  チューブを充填する : 硬貨 ==> ()
  チューブを充填する(この硬貨) ==
    (チューブ(この硬貨) := チューブ容量;
    中央制御部`残高が変わった());
```

ソースコード 2.8: チューブを充填する操作の定義

2.5.3　妥当性，機能項目の網羅性および実現性を獲得する

暫定的な仕様は形式仕様記述言語での適法な記述であるに過ぎず，まだ妥当性や機能項目の網羅性や実現性は確保されていない．妥当性や機能項目の網羅性や実現性を高めるためには，要求からは不明確だった点や要求の文面には明示されなかったドメイン知識や実現上の制約事項を明らかにし，明確な定義を示す必要がある．ドメイン知識を獲得するために，問題点や疑問点を指し示すための模型（モデル）として，暫定的な仕様を利用する．ドメイン専門家や他の工程の技術者をはじめとするステークホルダーに暫定的な仕様を示し，問題点や疑問点を示して議論し，それに基づくフィードバックを通して，妥当性，機能項目の網羅性，実現可能性が確認された仕様に修正する．以下に，それらの議論を通して暫定的な仕様から妥当性，機能項目の網羅性，実現可能性が確認された仕様を得る過程を示す．

チューブを充填する操作を追加する

まず，自動販売機の運用に精通した関係者により，この仕様では自動販売機に缶飲料を補充したり売り上げを回収するサービス担当者が釣り銭切れに対応できない点が指摘された．サービス担当者が自販機の売り上げを回収する時に，釣り銭切れでチューブが空だった時には，チューブを容量いっぱいに補充する．暫定的な仕様では，チューブを容量いっぱいに補充した時に，システムに対してそのチューブの内容を更新するための操作が提供されていない．そこで，ソースコード 2.8に示した操作を追加した．

```
精算ボタンが押された : () ==> ()
  精算ボタンが押された() ==
    (返金する(釣り銭(投入残高));
    投入残高 := 0;
    中央制御部`残高が変わった()));
```

ソースコード 2.9: 精算ボタンが押された操作の定義

```
state State of
  チューブ : 現金
  投入残高 : 金額
inv mk_State(チューブ, 投入残高) ==
  (forall この硬貨 in set dom チューブ &
    保管枚数(この硬貨) <= チューブ容量) -- 物理制約
  and 金額分の現金(チューブ, 投入残高) <> nil -- 精算可能
init s == s = mk_State(
  {<百円硬貨 |-> 0, <五十円硬貨> |-> 0, <十円硬貨> |-> 0},
   0)
end
```

ソースコード 2.10: 修正された状態定義

いつでも精算のための釣り銭の用意ができている状態を維持する

また，アーキテクトから精算時の釣り銭の計算について指摘があった．投入残高が 100 円で，チューブ内に十円硬貨が 1 枚，百円硬貨が 1 枚入っていたと仮定する．この時，80 円の缶飲料が販売されると残高が 20 円になり，精算できなくなる．この仕様では精算ボタンが機能しないために，ユーザは釣り銭を受け取ることができない．そこで 2 つの変更が必要である．まず，精算ボタンは常に有効でなければならない．したがって，ソースコード 2.3の「精算ボタンが押された」操作の定義はソースコード 2.9に変更された．さらに，精算ボタンが常に有効である，すなわち式「釣り銭 (投入残高)」が非 **nil** であるような状態を維持する必要がある．これを仕様の中で明示的に不変条件として表現した仕様をソースコード 2.10に示す．

ソースコード 2.10では，不変条件として，チューブ容量に関する物理制約に加えて，常にチューブから投入残高分の釣り銭を出すことができることを宣言している．「金額分の現金」操作の定義をソースコード 2.11に

```
pure 金額分の現金 : 現金 * 金額 ==> [現金]
金額分の現金(この現金, この金額) ==
  (dcl 残り:金額 := この金額,
       釣り銭の硬貨と枚数:map 硬貨 to 枚数 := {|->};
  for この硬貨 in 釣り銭優先順位 do
    if この硬貨 in set dom この現金
    then let
       この硬貨の枚数 : 枚数 =
         min(この現金(この硬貨),
           残り div 硬貨の金額(この硬貨))
       in
       if この硬貨の枚数 > 0
       then
         (釣り銭の硬貨と枚数 := 釣り銭の硬貨と枚数
           munion {この硬貨 |-> この硬貨の枚数};
         残り := 残り
           - 硬貨の金額(この硬貨) * この硬貨の枚数);
  return if 残り = 0 then 釣り銭の硬貨と枚数 else nil);

pure 釣り銭 : 金額 ==> [現金]
釣り銭(この金額) ==
  return 金額分の現金(チューブ, この金額);
```

ソースコード 2.11: 修正された釣り銭計算の操作

示す.「金額分の現金」操作は,「釣り銭」操作でのチューブの状態への参照を仮引数として抽象したものである．ソースコード 2.10での不変条件を記述するため導入された．そして,「釣り銭」操作は「金額分の現金」操作を使って再定義された.

この修正は，釣り銭切れの状態に関するドメイン知識を深めた結果を仕様に反映させるために行った．ソースコード 2.4での「釣り銭」操作の定義のかわりに，最初から釣り銭計算をソースコード 2.11のように定義することもできた．しかし，ソースコード 2.11を記述した段階では，その時点での表現としてより簡潔な定義であるソースコード 2.4での「釣り銭」操作の定義を採用した．アーキテクトからの指摘によって，自動販売機の金銭管理における釣り銭の役割についての理解が深まった結果として，ソースコード 2.11のほうが適切であると判断して，修正が行われた．したがって，この修正は，個人作業での試行錯誤での言語機能の選択が，コミュニケーションを介した試行錯誤の結果として解決された例といえる.

```
operations
  決済する : 金額 ==> ()
  決済する(この金額) == 投入残高 := 投入残高 - この金額
  pre  この金額 <= 投入残高
    and 金額分の現金(チューブ, 投入残高-この金額) <> nil;

  投入する : 硬貨 ==> ()
  投入する(この硬貨) ==
    (dcl
      投入後のチューブ:現金 := チューブ,
      投入後の残高:金額 := 投入残高 + 硬貨の金額(この硬貨);
    投入後のチューブ(この硬貨)
      := (if この硬貨 in set dom チューブ
        then チューブ(この硬貨)
        else 0)
        + 1;
    if
      投入後のチューブ(この硬貨) > チューブ容量(この硬貨)
    then
      投入後のチューブ(この硬貨) := チューブ容量(この硬貨);
    if
      金額分の現金(投入後のチューブ, 投入後の残高) <> nil
    then
      (atomic (
        チューブ := 投入後のチューブ;
        投入残高 := 投入後の残高);
      中央制御部`残高が変わった()));
```

ソースコード 2.12: 状態不変条件を守るよう修正された決済操作と投入する操作の定義

ソースコード 2.10で宣言した釣り銭に関する不変条件を守るためには，チューブや投入残高を更新する操作が不変条件に違反するような状態の更新を行わないように，事前条件などで示す必要がある.「決済する」操作および「投入する」操作が不変条件に違反するような状態の更新を行わないように修正された定義を ソースコード 2.12 に示す.

「決済する」操作には事前条件が加えられた．アーキテクトにこの事前条件を説明したところ，この操作は中央制御モジュールが呼び出すことから，この事前条件を保証するためには中央制御部が「金額分の現金」操作で釣り銭があるか確認して，釣り銭がない場合には販売ボタンの販売可能ランプを消灯してボタンを無効化することになった．また，硬貨の投入を受け付ける「投入する」操作に対して，精算のための釣り銭を

確保できなくなる場合には投入を拒否することで不変状態を守るように修正した.「投入する」操作中に出現している「atomic」は，複数の状態変数への更新を 1 回のシステム状態の更新として行うための言語機能であり，変数「チューブ」と変数「投入残高」への代入が両方完了するまで状態の不変条件の検査が行われない.

投入残高を表示可能な桁数以内に維持する

また，アーキテクトにより，投入残高の表示デバイスが 3 桁の LED 表示器であることから，投入残高が 3 桁を超えないようにするように指摘された．精算のための釣り銭に関する不変条件と同様に，状態に不変条件として桁数の制約を加え,「投入する」操作がその不変条件を守るよう修正した．修正された定義をソースコード 2.13に示す.

以上で， 図 2.1のコミュニケーションを介した試行錯誤が終了した．個人作業での試行錯誤およびコミュニケーションを介した試行錯誤の結果，妥当性および機能項目の網羅性および実現可能性がアーキテクトをはじめとするステークホルダーによって確認された VDM-SL 仕様が記述された．これで探索的仕様記述工程は完了し，この後，精緻化が行われる．精緻化では，型検査による機能定義の網羅性の確認や，証明やテストによる無矛盾性の確認を行い，リファクタリングや記述スタイルに関する規約によって検証容易性や変更可能性を確保する.

2.5.4　探索的仕様記述のまとめ

自動販売機の金銭管理モジュールの探索的仕様記述の過程として，個人作業での試行錯誤やコミュニケーションを介した試行錯誤を経て，妥当性と機能項目の網羅性と実現可能性が確認される過程を示した．個人作業での試行錯誤は，記述言語の文法に則った仕様を記述する作業に伴う言語機能に関する試行錯誤であり，仕様記述者への教育と，記述経験と，仕様記述環境による支援によって，軽減することができる．コミュニケーションを介した試行錯誤は，記述対象が関係する複数のドメインにおける専門的な知識を獲得するための試行錯誤であり，それぞれのドメイン知識を有するステークホルダーとのコミュニケーションによって解決する必要がある.

ドメイン知識に関する試行錯誤は，仕様の変更を伴う場合には，新しい定義を記述するための言語機能に関する試行錯誤が発生する場合がある.

```
values
  最大投入残高 = 999;

state State of
  チューブ : 現金
  投入残高 : 金額
inv mk_State(チューブ, 投入残高) ==
  (forall この硬貨 in set dom チューブ &
    保管枚数(この硬貨) <= チューブ容量) -- 物理制約
    and 金額分の現金(チューブ, 投入残高) <> nil -- 精算可能
    and 投入残高 <= 最大投入残高 -- 表示器からの制約
init s == s = mk_State(
  {<百円硬貨 |-> 0, <五十円硬貨> |-> 0, <十円硬貨> |-> 0},
   0)
end

operations
  投入する : 硬貨 ==> ()
  投入する(この硬貨) ==
    (dcl
      投入後のチューブ:現金 := チューブ,
      投入後の残高:金額 := 投入残高 + 硬貨の金額(この硬貨);
      投入後のチューブ(この硬貨)
        := (if この硬貨 in set dom チューブ
          then チューブ(この硬貨)
          else 0)
          + 1;
    if
      投入後のチューブ(この硬貨) > チューブ容量(この硬貨)
    then
      投入後のチューブ(この硬貨) := チューブ容量(この硬貨);
    if 金額分の現金(投入後のチューブ, 投入後の残高) <> nil
      and 投入後の残高 <= 最大投入残高
    then
      (atomic (
        チューブ := 投入後のチューブ;
        投入残高 := 投入後の残高);
      中央制御部`残高が変わった()));
```

ソースコード 2.13: 投入残高の桁数を守るための修正

また，言語機能の試行錯誤を経て記述した定義が，コミュニケーションを介した試行錯誤の結果，最終的に適切な言語機能の選択が判明する場合もある．例えば，釣り銭計算を定義する上で，状態変数である「チューブ」変数を参照するか，または，仮引数として抽象するかの選択は，言語機能に関する問題であり個人作業での試行錯誤に属する．ソースコード 2.4での「釣り銭」操作の定義では，前者の「チューブ」変数を参照する方法を選択して記述したが，コミュニケーションを介した試行錯誤の結果，常に投入残高が釣り銭として精算可能でなければならないことが判明し，状態の不変条件を定義するために，後者である仮引数として抽象する方法に変更した．

以上の一連の試行錯誤により，妥当性および機能項目の網羅性および実現可能性がステークホルダーによって確認された仕様が得られ，探索的仕様記述工程が完了した．

2.6　探索的仕様記述から精緻化への移行

形式手法はソフトウェア開発を数学の対象として扱うための道具立てであるが，探索的仕様記述において形式仕様は数学の対象であるだけでなく，社会的な対象でもある．形式仕様を記述するためには，人の認知による非形式的な理解を，形式仕様記述言語の言語機能を使って表現すること，および，多くの人に理解されフィードバックを反映することが求められる．そのため，探索的仕様記述では，個人作業での言語機能に関する試行錯誤と，コミュニケーションを介したドメイン知識に関する試行錯誤が行われる．探索的仕様記述を支援する環境は，これらの作業を支援するものであることが求められる．そのための支援ツールの設計指針を 3.2 節で示す．

探索的仕様記述により妥当性，機能項目の網羅性および実現可能性を獲得した後，仕様記述者は仕様の精緻化を行う．仕様の精緻化では，仕様に対するテストや証明によって，仕様の誤りを修正する．例えば，証明または十分な量のテストによって，状態に定義した不変条件が破られないことを確認する．また，後の修正がしやすいように仕様記述に対して語彙の統一や共通定義の抽出によるリファクタリングを行って，仕様記述の変更可能性を確保する．

精緻化においては形式仕様記述は数学的対象であり，VDM-SL を含む多くの形式仕様記述言語の開発環境では，これらの作業を支援するため

の機能が提供されている．場合分けの網羅性や事前条件や事後条件や不変条件や型の健全性が満たされることを確認するために証明すべき課題（証明責務）を自動的に生成する機能や，操作や関数について様々な組み合わせの引数を自動的に生成して評価実行して実行時エラーが発生しないことを確認する組み合わせテストを行う機能が提供されている．これらの機能を使って，仕様記述の機能定義の網羅性や無矛盾性や検証容易性や変更可能性を高めるための精緻化を行う．

形式仕様記述の実務者は，従来もドメイン専門家や関連する実装技術の専門家とのコミュニケーションを通して妥当性や実現可能性を獲得した上で，証明や仕様のテストを通して仕様記述の信頼性を高めてきた．本研究では，明示的に探索的仕様記述と精緻化に工程を分け，探索的な記述やコミュニケーションによる作業が求められる探索的仕様記述に焦点を絞り，探索的仕様記述のために必要な支援としての道具立てを確立する．精緻化で行う具体的な作業の分析やそのための支援ツールの設計指針は，本研究の対象範囲外とする．

第3章　探索的仕様記述への支援

本章では，探索的仕様記述の失敗要因を挙げ，その対応策として支援ツールの設計指針を示す．そして，失敗要因と設計指針をふまえて，形式仕様記述で用いられる技術について探索的仕様記述における役割を示す．

3.1　探索的仕様記述の失敗要因

本節では，探索的仕様記述がどのような原因で失敗するもしくは効率的でなくなるかを考察する．失敗要因が明らかになることで，どのような技術による支援が可能でどのような特性が支援ツールに求められるかの考察につなげる．

探索的仕様記述では，2.4 節において説明した個人作業での試行錯誤とコミュニケーションを介した試行錯誤が行われる．これらの試行錯誤が失敗すると，探索的仕様記述を進めることができない．本節では，2 つの試行錯誤それぞれについて失敗要因を挙げる．

3.1.1　コミュニケーションを介した試行錯誤での失敗要因

探索的仕様記述は，妥当性および実現性および機能項目の網羅性を確保した仕様記述を得る作業である．いずれの品質特性も，適切なドメイン知識を得るためのステークホルダーとのコミュニケーションが必須である．図 3.1に，探索的仕様記述での試行錯誤プロセスにおけるコミュニケーションに関する失敗要因を示す．コミュニケーションを介した試行錯誤の失敗要因は，各領域の専門家からのフィードバックを理解する作業と，形式仕様の意味を各領域の専門家に説明する作業において存在している．以下にそれぞれの箇所で発生する失敗要因を挙げる．

失敗要因 1 関係者からのフィードバックが仕様に反映されない．

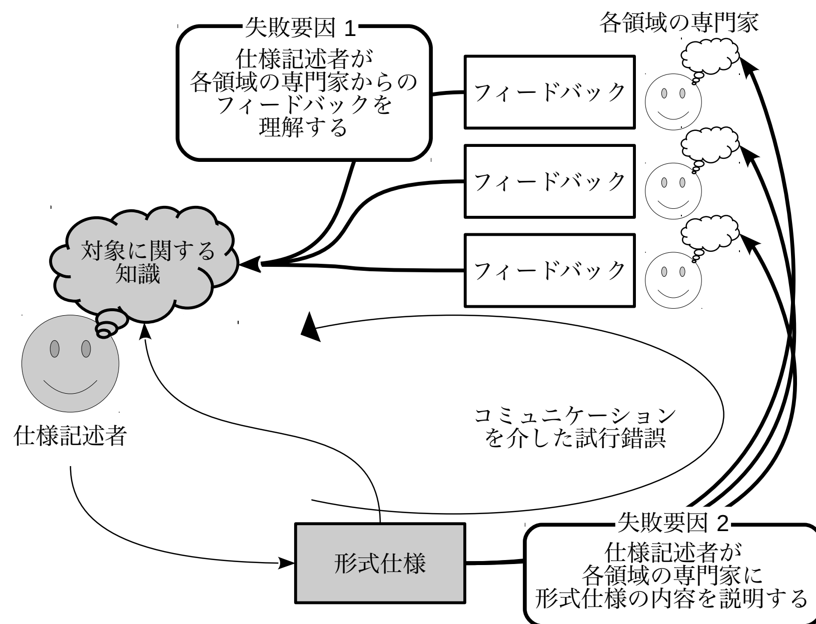

図 3.1: 探索的仕様記述での試行錯誤のプロセスにおけるコミュニケーションでの失敗要因

コミュニケーションの結果としてフィードバックが得られたとしても，そのフィードバックを仕様記述に反映させることができなければ意味がない．コミュニケーションの相手からのフィードバックを理解し，仕様記述全体の品質を損ねることのない形で仕様記述に反映させることが求められる．

失敗要因 2 仕様の意味が関係者に理解されない．

コミュニケーションを行う時点での仕様記述者の理解である仕様記述の内容を伝えることができなければ，品質特性を獲得するために効果的なフィードバックを得ることができない．コミュニケーションの相手である関係者の知識や技能に適合した方法で，仕様記述の内容を説明する必要がある．

3.1.2　個人作業での試行錯誤での失敗要因

探索的仕様記述では，仕様記述の品質特性のうち悪定義問題である実現性および妥当性および機能項目の網羅性の獲得を行う．したがって，一般に探索的仕様記述の対象である仕様記述には品質上の問題が多く残っている．しかし，探索的仕様記述自体を遂行する上で，記述言語の文法に関して適法な完結した仕様記述を効率的に記述する必要がある． 図 3.2に，探索的仕様記述での試行錯誤プロセスにおける記述作業についての失敗要因を示す．個人作業での試行錯誤の失敗要因は，仕様記述者が対象に関する知識に基づいて形式仕様を記述する作業と，記述した形式仕様の意味を理解して対象に関する知識を深める作業において存在している．以下にそれぞれの箇所で発生する失敗要因を挙げる．

失敗要因 3 仕様記述の修正に時間または労力がかかりすぎる．

効率よく試行錯誤を遂行しその結果を反映させるためには，一定の変更可能性が求められる．形式仕様記述工程の最終段階である精緻化では，形式仕様記述工程の成果物に求められる網羅性と無矛盾性を損なわないような変更を行うことが求められる．しかし，探索的仕様記述での試行錯誤の過程においては，仕様記述に対して全ての品質基準を厳格に求めると，試行錯誤に過大な時間と労力を要することとなり，探索の速度を鈍化させてしまう．例えば，各ドメインの専門家との議論の中で，仮に仕様を変更した場合のシステムの振る舞いを確認する場合においては，個人作

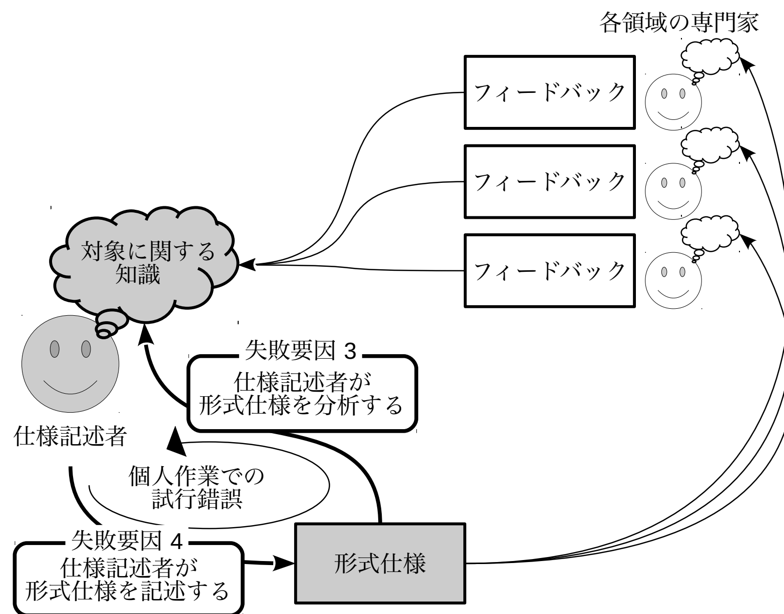

図 3.2: 探索的仕様記述での試行錯誤のプロセスにおける個人作業での失敗要因

業としての仕様記述作業が行われる．この個人作業では，コミュニケーションの成立に支障のない時間内での試行錯誤的な変更が求められる．

失敗要因 4 仕様記述の修正による品質の劣化が管理できない．

前述の**失敗要因 3** で示した通り，探索的仕様記述では試行錯誤による一定の誤りを許容する必要がある．しかし過度に寛容になると，仕様記述を何度修正しても全体として意味のある仕様記述が得られないことになりかねない．すなわち記述した仕様記述者にとって対象に関する自らの知識を深めることができなくなる．仕様記述の修正による劣化を許容する一方で，劣化の状況を把握し管理し，必要に応じて一定の品質を回復することが可能でなければ，探索的仕様記述を有効な形で精緻化に移行することができない．それを防ぐためには適正な品質レベルによる記述が求められることから，この失敗要因は個人作業での試行錯誤での失敗要因として扱う．

3.2　支援ツールの設計指針

現在用いられている開発ツールの多くは形式仕様の精緻化を主な目的として設計されている．実用規模の仕様の適切な抽象度，網羅性，無矛盾性や厳密な表明を求める精緻化に対して，その前段階として比較的小規模で不完全な仕様を扱う探索的仕様記述ではツールを設計する上での前提条件が異なる．ツールの利用シーンに関する前提および目的はツールを設計する上での意思決定に大きな影響を与える．3.1 節に挙げた失敗要因もツールの設計において考慮される必要がある．探索的仕様記述を支援するためのツールを設計する上で共通の指針となるべき方向性を，以下に設計指針として挙げる．

設計指針 1 ツールは比較的小さな仕様を対象として，仕様（モデル）と直接的な対話をするためのユーザインターフェイスを提供するべきである．

この設計指針は失敗要因 1 として挙げたフィードバックの反映，および失敗要因 2 として挙げた仕様の意味への理解に関する原則である．

対話の中での助言をその場で記述する場合，数百行におよぶような大きな仕様を対象とすることは現実的ではない．その助言に関わる限られた

スコープでの仕様を簡便に記述できることが必要である．すなわち，ツールは数行から数十行程度の比較的小さな仕様を対象として想定してユーザインターフェイスを設計することが望ましい．

ユーザインターフェイスとは，ユーザとソフトウェアの界面である．ソフトウェアが持つ情報をユーザに示し，また，ソフトウェアが持つ機能をユーザが操作するための手段を提供する．探索的仕様記述では，ツールはそれ自体がソフトウェアであると同時に，モデルである仕様を理解し操作するためのユーザインターフェイスとしての役割を持つ．したがって，ツールが持つ情報を示し機能を提供するだけでなく，モデルである仕様が持つ情報や機能を提供するユーザインターフェイスが求められる．

探索的仕様記述では，ツールのユーザは仕様記述者に限定されない．ツール上で仕様を記述するだけでなく，仕様の意味するところを顧客やドメイン専門家をはじめとするステークホルダーに説明することが求められる．仕様記述者はモデルを記述し分析するためのツールを操作するが，仕様記述者と対話相手が関心の対象とするのはツールではなく，仕様とその分析結果である．仕様記述者がツールではなくモデルを操作するようなインタラクションモデルを持つユーザインターフェイスを提供することで，ステークホルダーが仕様記述者によるモデルとの対話を観察し理解でき，また，仕様記述者がステークホルダーとの対話の文脈に沿って仕様を修正できることが望ましい．

設計指針 2 ツールは形式仕様記述言語の知識がない者にも理解可能な表示インターフェイスを持つべきである．

この設計指針は失敗要因 2 として挙げた，仕様の意味が関係者に理解されない状況に陥らないための原則である．形式仕様記述は自然言語による仕様記述と比較して，記述言語の読解ができる人が限られていることから，ツールによる支援への必要性が高い．

形式手法の効果を制限する要因として多様な関係者とのコミュニケーションが挙げられる [Fis94]．仕様の精緻化を目的とした従来のツールでは，ツールの利用者は形式仕様を専門とする技術者であり，したがって仕様記述言語および形式手法の知識を持つユーザを想定している．探索的仕様記述では，形式仕様技術者以外の関係者との対話が重要である．対話の中で仕様記述のためのツールを直接操作するのは仕様記述者であるが，顧客やドメイン専門家もツールの表示を観察し仕様の意味するところを理解することで対話をより円滑に進めることができる．

例えば，仕様の精緻化を目的とした従来のアニメーション実行器はテキストベースのユーザインターフェイスが提供されていることが多い．探索的仕様記述を支援するツールでは，単にツール自体がグラフィカルなインターフェイスを持つだけでなく，仕様のアニメーション実行の操作や状態の表示をグラフィカルなものにしたり，自然言語による補足説明を行うなど，仕様記述言語の文法を知らないステークホルダーにも理解できるような工夫が必要である．[Fis94] はそのための効果的なツールとしてプロトタイプやシナリオや利用時のシミュレーションを挙げている．仕様アニメーションをプロトタイプとして利用することは行われているが，そのプロトタイプが前提とする形式手法に関する知識をより少なくし，顧客やドメイン専門家にも容易に理解できるよう工夫することが求められる．

設計指針 3 ツールは仕様記述者により検査の厳格さの度合いを選択することが可能であるべきである．

この設計指針は失敗要因 3 として挙げた，仕様への修正に要するコストに関するものである．

探索的仕様記述では頻繁な仕様の修正が行われる．構文検査および型検査をはじめとする厳密な検査が可能であることは形式仕様記述の重要な利点であるが，試行錯誤を行う上では，厳格さを犠牲にしても実施可能な検査を行うことが有効な場合がある．

例えば，仕様のアニメーション実行において表明違反が発見された場合，主に仕様の精緻化を目的としたアニメーション実行器では，表明違反を発見すると直ちに実行を停止する．仕様の精緻化では，仕様の正当性の確認のためには表明違反は重大であり，仕様および実行シナリオを精査して誤りを発見する必要がある．

しかし，探索的仕様記述では仕様はまだ不完全であることが前提であり，表明の違反が重大な逸脱なのか，それとも表明で記述した条件が誤っているのかは，明白ではない．表明違反を容認して仕様アニメーションを継続することにより，動作を観察し，継続した状態が対象ドメインの知識に照らして正当なものなのかどうかを検討する必要がある．また，対象ドメインやシステムについての理解が不完全である場合は，仕様記述者が対象ドメインやシステムについて理解を深めることが求められる．ドメイン専門家の意見を求めるなど，複数のステークホルダーとの議論を伴う試行錯誤の中で理解を深めるためには，表明違反以後の動作の確認

や分析が必要な場合がある．以上のことから，探索的仕様記述では検査の厳格さを適切に選択することが必要である．

設計指針 4 ツールは継続的な分析が可能であるべきである．

この設計指針は失敗要因 4 として挙げた，仕様への修正による劣化に関するものである．

構文検査および型検査をはじめとする厳密な検査が可能であることは形式仕様記述の重要な利点であり，仕様の修正が行われた時に様々な分析や検査によりその修正の影響を把握することが求められる．

探索的仕様記述における仕様記述者の目的は，妥当な仕様を記述し，ステークホルダーと仕様に関する理解を共有することであり，分析および検査はそのための手段である．ユーザである仕様記述者が，目の前のタスクである仕様との対話，およびステークホルダーとの対話に集中できるよう，ツールは分析および検査を意識させないユーザインターフェイスを提供することが必要である．そのためには，自動的かつ継続的に実行可能な分析をツールが能動的に実行することが望ましい．

3.3　探索的仕様記述における形式仕様技術

本節では，直前の 3.2 節での設計指針を踏まえ，各形式仕様技術の探索的仕様記述への適用可能性と期待される効果および課題を示す．

3.3.1　構文解析

構文解析が可能であることは，形式仕様記述言語の重要な利点である．自然言語で仕様を記述した場合には構文解析による厳密な検査ができない．仕様の最低限の質を保証するために欠かせない技術であり，形式仕様に関する多くの技術の基礎となっている．

探索的仕様記述では頻繁な仕様の修正が行われる．仕様を修正した時には構文解析を行うことで構文誤りがないか検査する．設計指針 3 にあるように，探索的仕様記述では断片的な理解を記述する場合が多く，探索的仕様記述では必ずしも構文的に正しい仕様記述だけを対象とすることは現実的ではない．構文的に誤った仕様のうち構文的に正しい部分に適用可能な技術が望ましい．また，設計指針 4 に挙げた継続的な分析の

一種として，構文的に誤った仕様から構文的に正しい仕様を得るための作業を支援する技術が探索的仕様記述での生産性向上につながる．

3.3.2　型検査

一般に静的に型付けされたプログラミング言語における型検査には，プログラムを実行せずに行われる静的型検査と，プログラムの実行中に行われる動的型検査がある．形式仕様記述言語の多くは静的に型付けされており，静的型検査が行われる．静的型検査は記述の整合性に関する証明であり，形式仕様記述言語によって厳密な仕様を記述するための技術的な支援として有効である．設計指針 4 にあるように型検査が継続的に適用されるとより有効であると考えられる．

探索的仕様記述においても型検査は有効な技術である．記述された仕様の型に関する矛盾を発見することで，仕様記述者による対象ドメインやシステムに関する理解の矛盾を発見することができる．型検査によって仕様記述者の理解の矛盾を発見し解決していくことで，仕様記述者の理解をより厳密で妥当なものにしていくことができる．形式仕様記述工程の成果物としての厳密な仕様を記述するために静的型検査は非常に有効な技術であるが，設計指針 3 にあるように，型に関する矛盾を解消していく上で静的な型付けなしで適用可能な技術を利用することも有効な場合があることから，探索的仕様記述を支援するための開発環境は静的型検査を強制しないことが望ましい．

3.3.3　仕様アニメーション

仕様アニメーションは，形式仕様記述言語の実行可能なサブセットにより記述された仕様を実行する技術である．実行機構としてインタプリタを利用することが多く，VDM-SL では言語仕様でインタプリタに関する仕様が定められている．仕様アニメーションにより，形式仕様記述をプロトタイプとして利用することができる．仕様アニメーションは，与えられた仕様に基づいてソフトウェアシステムを実装した場合に完成するであろうシステムの動作をシミュレーションする技術といえる．

探索的仕様記述を通して妥当な仕様を記述するためには，顧客やドメイン専門家の知識や経験や洞察力が必要だが，顧客やドメイン専門家の多くは形式仕様記述言語に精通していない．この問題に対する解として

は，顧客やドメイン専門家のうち何人かに形式仕様記述言語の初歩を教えることで，形式仕様の内容を理解するために求められる最低限の読解スキルを身につけてもらうことが挙げられる．しかしこの方法は，顧客やドメイン専門家に対して形式仕様記述言語を習得するための時間や手間を求めることになり，必ずしも実施可能とは限らない．また，形式仕様技術者が形式仕様の内容について自然言語で顧客やドメイン専門家に説明することも可能であるが，この場合には顧客やドメイン専門家は形式言語を用いたことによる厳密さを享受することができない．UI デザイナやマニュアル作成者なども，形式仕様による明確で厳格な仕様記述を理解できることが望ましいが，必ずしも形式仕様記述言語の読解スキルを持っているとは限らない．探索的仕様記述では，仕様記述者と様々なステークホルダーの対話によってシステムに関する共通理解を醸成することが重要であるが，形式仕様記述言語に関するスキルの差が大きな障壁となる．設計指針 2 にあるように，形式仕様の専門知識を持たない関係者にも理解可能な表現によるアニメーションが望ましい．形式仕様記述言語を直接読解することができないステークホルダーは，仕様アニメーションによるプロトタイプを使ってそのシステムの動作を理解することで，形式仕様の内容を知ることができる．探索的仕様記述において，仕様アニメーションは，多様なステークホルダーと仕様に関する共通理解を築くために有望な技術である．

仕様アニメーションは仕様のテストにおいても利用される．プログラミング言語と同様に，VDM には vdmUnit [LLJ+13] と呼ばれる単体テストフレームワークがあり，また，組み合わせテスト [LLB10] の機能も実現されている．仕様を基に開発されたプログラムのテストを行う時のテストオラクルとしても利用される．探索的仕様記述では仕様の修正が高い頻度で行われることから，仕様の誤りを早期に発見するための仕様の組み合わせテストを含む単体テストが仕様の品質と生産性を高めるために有効である．

3.3.4　仕様スライシング

仕様スライシングは形式仕様からある特定の特性や式の値に影響を与える部分を抽出する技術である [OA93]．仕様に対する修正の影響範囲を把握するためや，仕様を説明する時に特定の動作に関係ある部分を示すために利用することができる．設計指針 1 に挙げた小規模な仕様との直

接的なインタラクションを有効に利用する上で有用であると考えられる．

3.3.5　プログラムの自動生成

プログラムの自動生成とは，形式仕様記述言語で記述された仕様からツールによって機械的にソースコードを生成する技術である．自然言語で記述された仕様では非常に困難であるが，構文および意味論が厳密に定義された仕様記述言語には，B-method や VDM-SL などプログラムを自動生成可能なものがある．プログラムを自動生成するツールは，コード生成器 (code generator) またはトランスパイラ (transpiler) と呼ばれる．

プロダクションコードを自動的に生成することで実装工程の作業を大幅に減らすことができるため，プログラムの自動生成は形式仕様によるフロントローディング効果を高めることができる．また，プログラムの自動生成は，ソフトウェア開発の最終的な成果物であるプロダクションコードを生成するだけでなく，仕様アニメーションの実行機構としても利用される．プログラムの自動生成は，インタプリタによる実行と比較して実行速度において有利であることが多い．

探索的仕様記述では仕様のプロトタイプ実行や仕様のテストで仕様アニメーションが利用される．最終成果物であるプロダクションコードではなく，仕様アニメーションとしてプログラムを自動生成する場合には，設計指針 3 に挙げたように，仕様記述の完成度に応じて，コード生成器による検査や生成されたプログラム内の実行時検査についての厳格さの度合いを指定できることが望ましい．設計指針 2 に挙げた形式仕様の知識を持たない関係者にも理解できるよう，生成されたプログラムの表現としてグラフィカルなユーザインターフェイスと結合するなどの応用も有効であると考えられる．

3.3.6　自動単体テスト

プログラムの正当性の証明や，仕様からのプログラム自動生成は，仕様に対するプログラムの品質を確保するための技術である一方，テストはテスト対象の動的な特性を確認するための技術である．一般には実行プログラムがテスト対象となるが，実行可能な形式仕様を取り入れた開発では，形式仕様をテスト対象とすることができる．

自動単体テストは eXtreme Programming をはじめとするアジャイル開発を中心に広く利用されている技術であり，実行可能な形式仕様記述言語においても誤りを発見する上で有効な技術である．形式仕様記述の手法である VDM においても vdmUnit [LLJ+13] と呼ばれる単体テストフレームワークが開発され，産業界においても実用されている．これらのテストフレームワークでは，技術者がテストケースとして一連の動作と検査すべき特性を記述し，テストフレームワークがそれらのテストケースを実行し，記述された通りの特性を示したテストケース，記述した特性を満たさなかったテストケース，実行に失敗したテストケースに分類集計して，技術者に示す．すなわち，これらのテストフレームワークは，記述対象が持つべき特性を明示的に記述し，その特性からの逸脱を検知するためのツールである．

探索的仕様記述ではアジャイル開発と同様に修正の頻度が高いため，回帰テストとしての自動単体テストは修正によって発生した誤りの早期発見に有効である．設計指針 3 に挙げたように仕様記述の完成度に応じたテストの厳密さの指示ができることが望ましい．また，テストは設計指針 4 に挙げたように継続的に行うことができることが望ましい．

第4章　探索的仕様記述環境 ViennaTalk

探索的仕様記述とそのためのツールの設計指針を具体的なツールとして示すために，探索的仕様記述を支援するための開発環境 ViennaTalk を開発した．本章では，ViennaTalk について，アーキテクチャ，設計指針，およびライブラリを説明することによって，探索的仕様記述を支援するためのツールに共通して必要となる構成要素を示す．

4.1　アーキテクチャ

ViennaTalk のアーキテクチャ構成を 図 4.1に示す．ViennaTalk は Pharo Smalltalk [BDN+09] 上に実装されている．また，ViennaTalk はアニメーション実行エンジンとして VDMJ インタプリタ [Bat09] を利用するためのインターフェイスを提供している．VDMJ インタプリタは VDM-SL, VDM++ および VDM-RT を解釈実行する Java プログラムであり， JRE (Java Runtime Environment) 上で動作する．さらに，ViennaTalk にはウェブサーバ機能が実装されており，ウェブブラウザ上で実行される JavaScript プログラムを含んでいる．

ViennaTalk を実装するための言語環境として，Pharo Smalltalk を採用した．Smalltalk 環境 は，ダイナブック構想 [Kay72] の実現に向けて開発され，マルチウィンドウシステムを持つグラフィカルなユーザインターフェイスやオブジェクト指向プログラミングといった現代のプログラミング環境の重要な基礎を確立したプログラミング環境である．Kay らは，ユーザがプログラミングを通じて数学や物理などの学習や，ビジネスでのデータ分析などを行うための，あらゆる年齢層を対象にした個人媒体として，ダイナブックを提案し，その暫定的な実現として Smalltalk 環境を開発した．

Smalltalk の特徴は個人向けのダイナミックメディア [KG77] であり，ユー

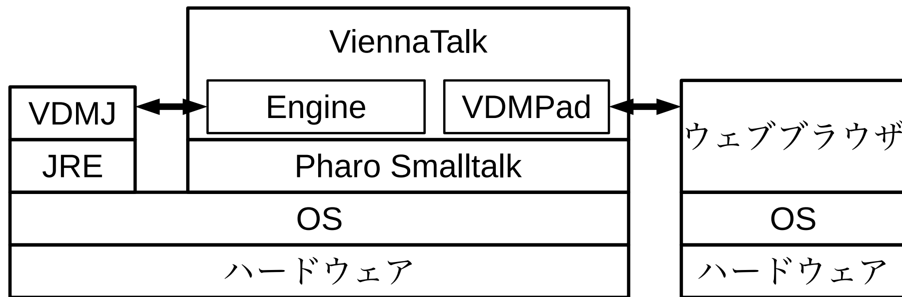

図 4.1: ViennaTalk のアーキテクチャ構成

ザと計算機の間の柔軟なインタラクションを重視して開発された．ユーザであるプログラマが実行中のプログラムを修正し，元のプログラムを終了させせることなくプログラムの修正を反映させ，継続実行させることができることを，プログラミング環境のライブ性と呼び，ライブ性のある環境の上でのライブ性を利用したプログラミングをライブプログラミングと呼ぶ．Smalltalk 環境は，ライブ性を持つプログラミング環境である．また，Smalltalk 環境は全て Smalltalk 自身で記述され，修正することが可能であることから，Smalltalk 以外の言語の開発環境も含め，開発ツールおよび開発環境を構築するための基盤として利用することに適している．Smalltalk 環境で構築された開発ツールの例として，sUnit が挙げられる．sUnit は Kent Beck により開発された自動単体テストフレームワークである．sUnit は一般に普及した自動単体テストフレームワークである xUnit の最初の実装であり， 1.4 節で関連研究として挙げたアジャイル開発の一つである eXtreme Programming 手法の基礎となった [Ola03].

Pharo Smalltalk は オープンソース形式で活発な開発が行なわれている Smalltalk 環境の一つである [BDN+09]．Smalltalk 環境として柔軟なインタラクションの実現やライブプログラミングが可能であることに加え，現代的なユーザインターフェイスを構築するための Morphic [MS95, IKM+97] と呼ばれるフレームワークや，パーサライブラリなどの補助ライブラリが充実している．

3.2 節で示したデザイン指針に従ったツールを開発するためには，以下の要件を満たす基盤環境が必要である．

- デザイン指針 1 として挙げた，仕様との直接的な対話をするためのインターフェイスを構築するための，柔軟なユーザインタラクショ

ンの構築が容易であること．

- デザイン指針 2 として挙げた，形式仕様記述言語の知識がない者にも理解可能な表示インターフェイスを構築するための，新しいウィジェットの作成が容易であること．
- デザイン指針 3 として挙げた，仕様記述者により検査の厳格さの度合いを選択することが可能なツールを実装するための，柔軟な処理系の構築が可能であること．
- デザイン指針 4 として挙げた，継続的な分析を実装し開発環境の一部として実行することが容易であること．

Pharo Smalltalk は，ユーザからの入力イベントの処理からウィジェットの描画まで，ユーザインターフェイスやユーザインタラクションの柔軟な構築が可能であることから，デザイン指針 1 およびデザイン指針 2 を満たすツールを開発するために適していると考えられる．また，柔軟な処理系の構築に必要なパーサライブラリや DSL 処理系を実装するためのフレームワークおよび sUnit から発展したテストフレームワークが Pharo Smalltalk 上で提供されており，デザイン指針 3 およびデザイン指針 4 を満たすツールを開発する基盤として適している．以上の理由から，ViennaTalk を実装する基盤として，Pharo Smalltalk を採用した．

4.2　設計指針

本節では，3.2 節で示したデザイン指針のそれぞれについて，ViennaTalk における実現を示す．

設計指針 1 ツールは比較的小さな仕様を対象として，仕様（モデル）と直接的な対話をするためのユーザインターフェイスを提供するべきである．

ViennaTalk は，仕様との対話を実現する核となるオブジェクトとしてアニメーションオブジェクトを定義している．アニメーションオブジェクトは，仕様アニメーションの対象となる仕様とアニメーション状態を組としたものである．ViennaTalk のアニメーションオブジェクトは仕様記述者に直接的な対話を可能にするためにライブ性とイメージベース環境を持つ．

ライブ性とはプログラミング環境の特性であり，プログラミング環境の上で実行中のプログラムを修正し，修正されたプログラムに従って実行を継続することが可能である時に，そのプログラミング環境はライブ性を持つとされる．ライブ性は形式仕様のアニメーション実行にも拡張して適用することができる．すなわち，仕様アニメーション環境の上でアニメーション実行中の仕様を修正し，修正された仕様に従ってアニメーション実行を継続することが可能である時に，その仕様アニメーション環境はライブ性を持つとする．ライブ性を持つことで，ツール利用者はアニメーションとの柔軟で直接的な対話を行うことができる．

図 4.2に非ライブなアニメーションとライブなアニメーションの比較を示す．仕様アニメーションは仕様を読み込み，初期状態である「State1」から操作「Op1」を実行することで，状態を「State2」に変化させる．ここで，仕様の問題点に気付き，仕様を編集する時，非ライブな仕様アニメーションでは，修正された仕様に対して新しい仕様アニメーションを開始し，初期状態「State1'」を作り出す．元の仕様での「State2」を再現するためには，状態「State1'」から操作「Op1」を再度実行し，状態「State2'」に変化させる．その後，さらに続く操作「Op2」を実行することで，状態「State3'」に変化させる．一方，ライブな仕様アニメーションでは,「State1」に操作「Op1」を実行し，状態が「State2」となった時に仕様を修正しても，状態「State2」を可能な範囲で維持して「State2"」とし，続く操作「Op2」を実行することで状態「State3"」に変化させる．

ライブな仕様アニメーションは，開発対象となるシステムの利用シナリオに沿って仕様をアニメーション実行させながらその動作を観察し，問題点を発見し次第修正しながら利用シナリオを続行していくことを可能にする．これはドメイン専門家と仕様アニメーションをしながら議論し，ドメイン知識を獲得しつつ，仕様に新しいドメイン知識を反映させていく際に有用である．一方，元の仕様での操作による状態変化と修正された仕様での操作による状態変化が混在することになり，厳格な検証には向かない．

探索的仕様記述では仕様の修正が頻繁であり，アニメーション実行中にアニメーション状態を観察することで得られた知見に基づいて仕様を修正し，その修正の効果を直ちに確認するためにライブ性は効果的である．また，ViennaTalk では，状態の変更もライブ性の一部として扱う．すなわち，ViennaTalk の仕様アニメーション実行機構は，アニメーション実行中の仕様で定義された操作以外に，ユーザが直接状態を書き替える

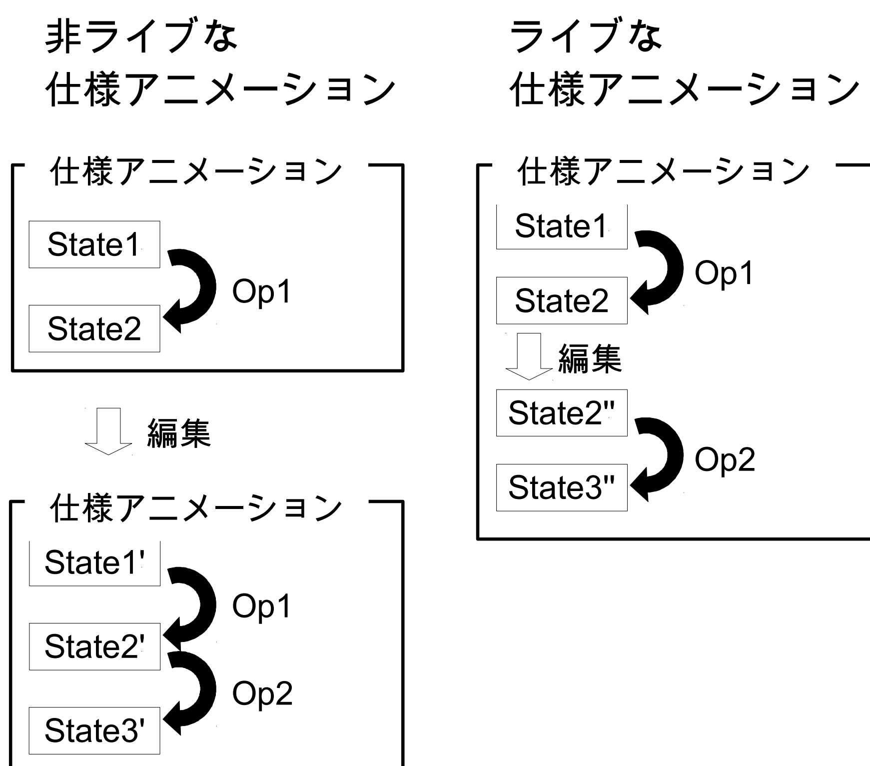

図 4.2: 非ライブなアニメーションとライブなアニメーションの違い

ことによる変更を許す．探索的仕様記述においては，状態の変更は様々な仮説的な状態を設定して操作をアニメーション実行することで記述された仕様の動作を理解するための有効な手段である．

ライブ性は探索的仕様記述と精緻化では，探索的仕様記述でより効果的である．精緻化においては仕様の修正は頻繁ではなく，また，修正前の仕様でのアニメーション状態をそのまま修正後の仕様のアニメーションに継続させることは，修正前の仕様の特性と修正後の仕様の特性を組み合わせることになり，厳格な検証を損ねることから，ライブ性は精緻化においては有用な特性とはいえない．仕様の精緻化を目的とした従来のVDM-SL 開発支援ツールは，仕様を変更するとアニメーション状態が初期化されるため，ライブ性を持たない．また，仕様の精緻化においては記述対象となるシステムの取り得る状態を厳格に扱うために，初期化命令を除き，使用中で定義された操作の実行を通さなければアニメーション状態を変更することを禁止する．

ViennaTalk のもう 1 つの大きな特徴は，イメージベース環境である．イメージベース環境とは，ファイルベース環境との対照で用いられる用語である． 図 4.3にファイルベース環境とイメージベース環境の概念的な違いを示す．ユーザがソースプログラムや仕様等の記述をファイルに書き込み，エディタやインタプリタやコンパイラ等のツールにファイルへの参照を与えることでツールの機能を提供する開発環境がファイルベース環境であり，多くのツールキットや統合開発環境はファイルベース環境として実現されている．ファイルベース環境では，ユーザである開発者は常に自分が操作の対象としているのはどのファイルであるかを意識し，その内容をエディタで編集したり，あるいはインタプリタ等のツールでファイルに対する分析や操作を行う．一方，イメージベース環境とは，ファイルを介さずにソースプログラムや仕様記述を開発環境内部のメモリイメージ内にオブジェクトとして保持し，各ツールはそのメモリイメージ中のオブジェクトに直接アクセスする環境であり，代表的な例として Smalltalk 環境がある．イメージベース環境では，ユーザはスクリーン上に表示されているオブジェクトを対象として操作を行う．探索的使用記述においては，操作の対象は仕様アニメーションの状態や仕様記述であり，ファイルを意識する必要はない．作業状態の保存や復元は，環境であるオブジェクトシステム全体をイメージファイルと呼ばれるファイルに格納するが，ユーザは必ずしもそれを意識する必要はない．

探索的仕様記述において仕様と直接的な対話を実現するためには，ファ

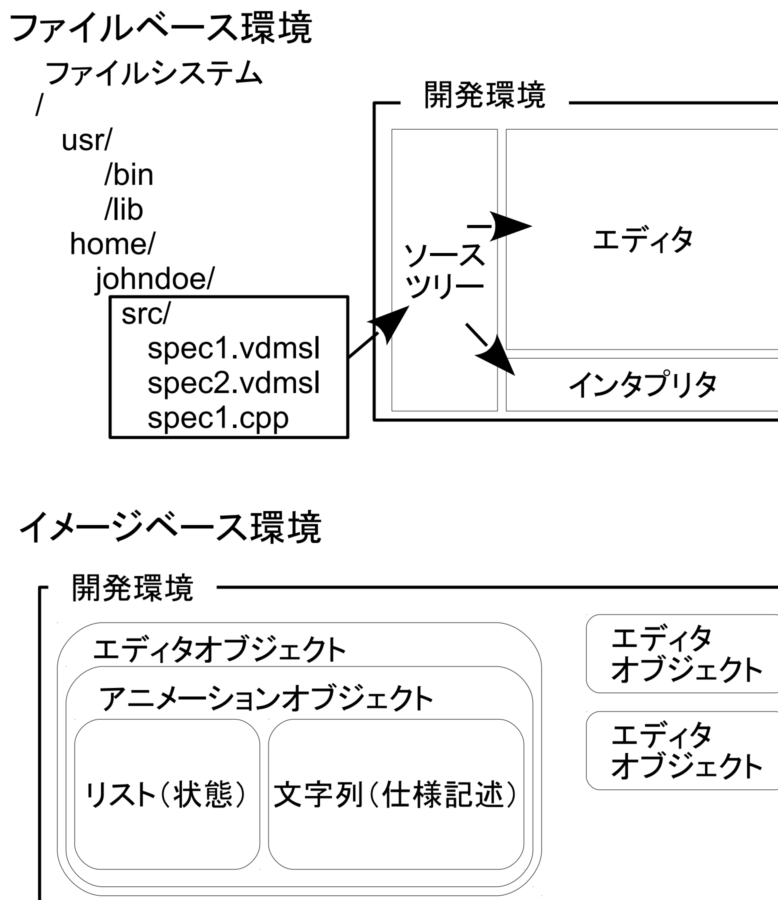

図 4.3: ファイルベース環境とイメージベース環境の違い

イルのような中間に介在するものを極力排し，各ツールのユーザインターフェイスを介してアニメーションオブジェクトに直接アクセスすることが可能であることが望ましい．ViennaTalk が基盤としている Pharo Smalltalk はそれ自体がイメージベース環境であり，ViennaTalk もイメージベース環境として設計された．ただし，精緻化以降において利用される開発環境との連携のために仕様記述をファイルに入出力する機能は実装されている．

設計指針 2 ツールは形式仕様記述言語の知識がない者にも理解可能な表示インターフェイスを持つべきである．

ViennaTalk では，VDM-SL の値をグラフィカルに表示するための 2 つの方法を提供する．1 つは HTML による図的表示であり，もう 1 つは Smalltalk によるユーザインターフェイス構築である．

VDM-SL の値の表記は形式仕様技術者やプログラミングの知識を持つ人には難しいものではないが，それ以外のステークホルダーにとっては読解の障壁となりうる．ViennaTalk には VDM-SL の値を HTML での図的表現に変換する機能が実装されている．集合や列，レコード，写像等の構造を持つ値を図的に表現することで，技術的背景の少ないステークホルダーにとっての障壁を下げることができる．また，トークン型やクォート型の値のように構造を持たない値であっても，型の種類が視覚的に判別できるようにすることで，読解の負荷を減じることができる．

ViennaTalk は，VDM-SL 仕様の内容を VDM-SL の文法知識を持たないステークホルダーに説明するためのもう 1 つの手段として，VDM-SL 仕様に GUI を付けたプロトタイプの作成を支援する． 4.5 節で説明する Lively Walk-Through は，VDM-SL 仕様に簡便な GUI を結合して実行可能にする UI プロトタイピング環境である．また，ViennaTalk の基盤となっている Pharo Smalltalk では，Morphic [MS95, IKM+97] と呼ばれる GUI (Graphical User Interface) フレームワークが提供されている．Lively Walk-Through が提供する汎用な GUI 部品では不足がある場合には，Morphic フレームワークを利用して独自の GUI 部品を作成することができる．

さらに，Lively Walk-Through を使わず，VDM-SL 仕様のアニメーションを Smalltalk オブジェクトとしてパッケージングすることで，VDM-SL 仕様を Smalltalk アプリケーションの一部として組み込み，プロトタイプを作成することができる．ViennaTalk は，VDM-SL 仕様のアニメーションを Smalltalk オブジェクトとしてパッケージングするとして，アニメー

ションオブジェクトによる方法とトランスパイラによる方法の2つを提供している．本節の設計指針1で説明した通り，ViennaTalk は VDM-SL 仕様のアニメーション実行をアニメーションオブジェクトとしてモデル化している．アニメーションオブジェクトは1つの VDM-SL 仕様の実行状態をオブジェクトとして表現したものであり，一般の Smalltalk のコンテナオブジェクトに共通するインターフェイスである `at:` や `at:put:` メッセージによって内部状態の入出力が可能である．従って，Smalltalk で実装されたコンテナオブジェクトを扱う Smalltalk プログラムは，VDM-SL 仕様を通常の Smalltalk プログラムの一部として動作させることができる．トランスパイラによる方法は，VDM-SL 仕様から Smalltalk ライブラリを自動生成し，そのライブラリを利用したアプリケーションをプロトタイプとして作成する方法である．詳細は 4.3.7 節で説明するが，ViennaTalk が提供するトランスパイラは実行可能な VDM-SL 仕様中のモジュールをクラスに，各モジュールの状態変数を Smalltalk クラスのインスタンス変数に，関数や操作をメソッドとして生成する．これらのインターフェイスを利用する GUI アプリケーションを作成し，VDM-SL 仕様が定義する機能を理解するためのプロトタイプとして利用することができる．

まとめると，ViennaTalk では，モデルとしての仕様について，グラフィカルな表現を与えることで，VDM-SL の文法知識を持たないステークホルダーに VDM-SL 仕様が定義する状態や機能を説明する仕組みを提供する．グラフィカルな表現を与える仕組みとして，値を HTML 上で図的に表現する機能，Lively Walk-Through により簡便な GUI 部品により GUI 付きプロトタイプを作成するプロトタイピング環境，および，アニメーションオブジェクトまたはトランスパイラによって VDM-SL 仕様を Smalltalk オブジェクトとしてパッケージングすることで Smalltalk のグラフィカルなアプリケーションの一部として動作させることでプロトタイプを作成する方法を提供している．

設計指針 3 ツールは仕様記述者により検査の厳格さの度合いを選択することが可能であるべきである．

この設計指針は探索的仕様記述においては厳格さよりも試行を優先させることがあることから，精緻化においては許容されないエラーに対して，許容することがユーザによって選択可能であるべきことを表している．ViennaTalk では，表 4.1に示す検査機能がインタプリタおよびトランスパイラにより提供されている．

表 4.1: ViennaTalk が提供する検査機能の適用選択肢

検査種別	検査名	インタプリタ	トランスパイラ
コンパイル時	構文検査	必須	必須
	静的型検査	必須	選択可
実行時	動的型検査	選択可	選択可
	表明検査	選択可	選択可

ViennaTalk が提供する検査機能には，コンパイル時に行われる構文検査と静的型検査，および，実行時に行われる動的型検査と表明検査がある．既存の VDM-SL 開発環境である VDMTools および Overture tool では，これに加えて，証明責務の自動生成がある．証明責務の自動生成は，定義された機能が事前条件および事後条件，不変条件，静的型に違反しないことや定義の網羅性を確認するために証明すべき特性を列挙する機能で，精緻化において有用な検査機能であり，探索的仕様記述を支援する ViennaTalk では提供していない．

コンパイル時に行われる構文検査については常に厳格に適用される．すなわち言語仕様で定義された構文規則に違反した場合，インタプリタ実行もトランスパイラによるプログラム生成も行われない．ただし，探索的仕様記述の段階ではまだ仕様記述が不完全であるという前提があることから，疑わしい記述に対する警告メッセージは表示しない．静的型検査については，インタプリタは常に静的型検査を行い，型に関する誤りが発見された場合にはインタプリタ実行を行わない．これは，ViennaTalk が利用している VDM インタプリタである VDMJ からくる制約である．トランスパイラは静的型検査を行うか行わないか，また，型に関する誤りが発見された時にプログラム生成を停止するか警告に留めるかを設定により選択することができる．

実行時に行われる動的型検査および表明検査については，インタプリタおよびトランスパイラのいずれも実行時型検査を行うかどうかを設定により選択することができる．

設計指針 4 ツールは継続的な分析が可能であるべきである．

ViennaTalk では継続的な分析として，アニメーションによる継続的テストを提供する．継続的テストはテスト対象の動作について自動的に実

行可能なテストを開発プロセスに組み入れることで，誤りの迅速な発見を支援する．

ViennaTalk が実装しているウェブ IDE である VDMPad では VDM-SL 仕様に対する継続的なテストが可能である．VDMPad 上で VDM-SL 仕様をアニメーション実行すると，その実行状況と結果をテストケースとして定義することができる．定義されたテストケースは，以後のアニメーション実行ごとに再検査される．すなわち，ある機能の振る舞いを後の仕様修正によって変えたくない場合に，その振る舞いをテストケースとして定義しておくことで，その振る舞いが変わっていないかどうかを，アニメーション実行のたびに自動的に確認することができる．仕様への修正がテストケースとして保存された振る舞いから逸脱した場合には，直ちに把握することができる．

ViennaTalk の基盤となっている Pharo Smalltalk には，継続的テストのためのテストフレームワークが標準で提供されている．Pharo Smalltalk のテストフレームワークを利用して，VDM-SL 仕様に対するテストをアニメーションオブジェクトに対するテストとして記述することができる．VDM-SL 仕様から生成された Smalltalk ライブラリに対しても Pharo Smalltalk のテストフレームワークを利用してテストすることができる．

4.3　ViennaTalk ライブラリ

本節では，探索的仕様記述環境 ViennaTalk を構成するライブラリを説明する．ViennaTalk は複数のパッケージと呼ばれるモジュールから構成されている．パッケージは，クラス群および既存クラスへの拡張メソッド群を持ち，依存関係を宣言することで構成を管理することができる．

ViennaTalk のパッケージ構成を 図 4.4に示す． 図 4.4中の細線による矩形はパッケージを表わし，中のラベルはパッケージ名を表わす．各パッケージには 1 つまたは複数のクラスおよびメソッドが格納されている．あるパッケージで定義されたクラスの個別のメソッドを別のパッケージに格納することも可能である．パッケージ間の矢印は依存関係を表す．例えば Engine パッケージから Zinc パッケージに矢印が引かれているが，これは Engine パッケージは Zinc パッケージに依存していることを表わす．太い実線による矩形内のパッケージが ViennaTalk が提供するパッケージであり，矩形外のパッケージは外部オープンソースソフトウェアパッケージである．太い点線による矩形はパッケージグループを表わす．パッケー

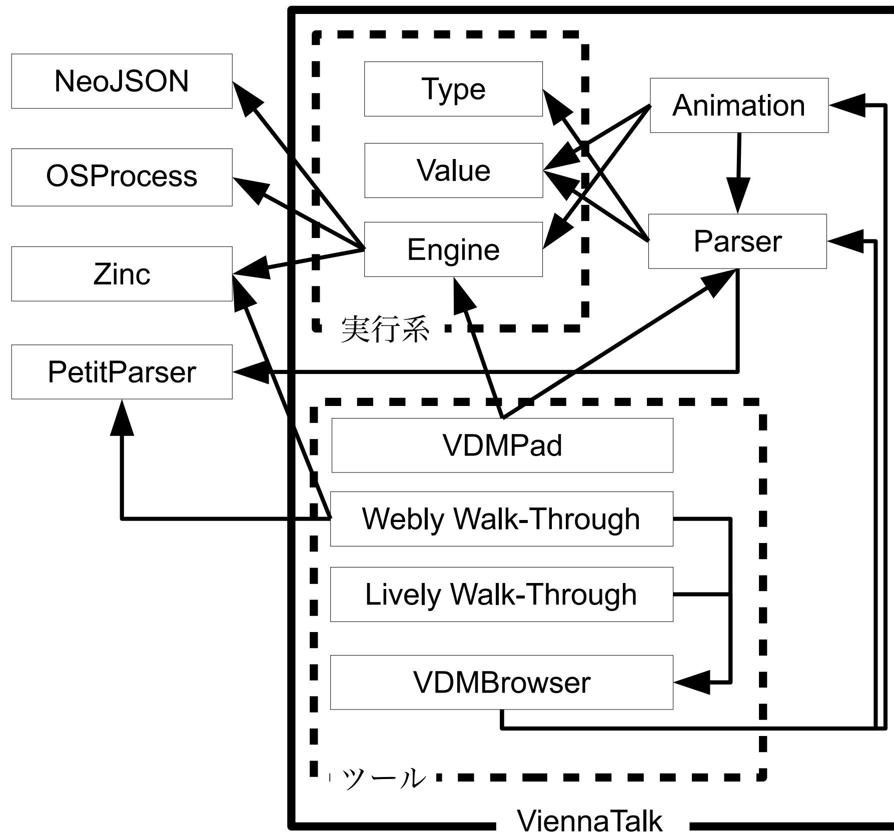

図 4.4: ViennaTalk を構成する Smalltalk パッケージ群

ジグループにプログラム上の意味はなく，構成を把握し易くするために便宜的につけられたパッケージの分類名を表わす．例えば`Type`パッケージ，`Value`パッケージおよび`Engine`パッケージが実行系グループとしてまとめられている．これらのパッケージ構成は Pharo Smalltalk の標準構成管理ツールである Metacello によって自動管理されており，ViennaTalk 上から自己更新することができる．

以下，ViennaTalk を構成する各パッケージについて説明する．

4.3.1　Animation パッケージ

Animation パッケージは仕様アニメーション実行中の VDM-SL モデルを表すアニメーションオブジェクトである ViennaAnimation クラスを提供する．アニメーションオブジェクトは，VDM-SL 仕様と状態を保持し，VDM-SL 仕様の参照と変更，各状態変数の参照と変更，および，VDM-SL 表現式の実行を行う．

探索的仕様記述では，仕様アニメーションによる妥当性の確認とその結果行われる仕様の修正が頻繁に行われる．アニメーションオブジェクトは単に VDM-SL 仕様を保持するコンテナではなく，探索的仕様記述で仕様記述者が行う作業の中心に位置するものであり，ViennaTalk においても後述の VDMBrowser や Lively Walk-Through を始めとする各ツールの実装に用いられている．

4.3.2　Browser パッケージ

Browser パッケージは探索的仕様記述を目的とした VDM-SL 用のブラウザ VDMBrowser を実装する `ViennaBrowser` クラスを提供する．VDMBrowser の特徴として，4.2 節 の設計指針 1 で説明した通り，ライブなアニメーションオブジェクトが中心であることが挙げられる．従来の IDE でのコードブラウザがソースファイルを編集することを目的としているのに対して，VDMBrowser は Animation パッケージが提供するアニメーションオブジェクトを編集するためのインターフェイスであり，すなわち，アニメーション実行中のモデルを操作するためのツールである．モデルに対する操作として，ユーザは VDMBrowser のユーザインターフェイスを通して VDM-SL 仕様の編集，状態変数の値の編集，および VDM-SL で記述された式の評価を行うことができる．

図 4.5に VDMBrowser の画面例を示す．VDMBrowser の UI は大きく上下に分かれており，上部には左から順にモジュールリスト，状態変数リスト，値テキストエリアが配置されている．下部はタブにより 2 つの UI を切り替えることができる．Specification タブを選択すると，下部のテキストエリアは VDM-SL の仕様を編集するための仕様テキストエリアが表示される．Workspace タブを選択すると，下部のテキストエリアはワークスペースと呼ばれるテキストエリアが表示される．図 4.6に Workspace タブが選択された場合の画面例を示す．

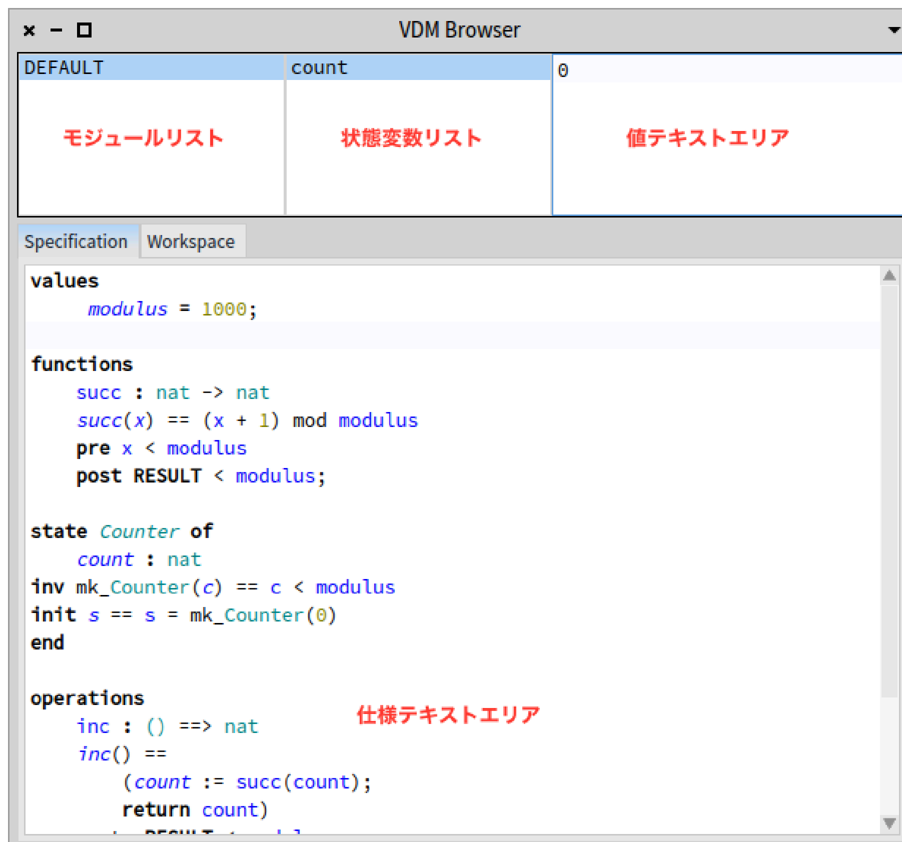

図 4.5: VDMBrowser での仕様編集の画面例

図 4.6: VDMBrowser の Workspace の画面例

ブラウザ上部のモジュールリストには，VDM-SL 仕様で定義されているモジュールが列挙され，ユーザがそのうちの 1 つのモジュールを選択することで，VDMBrowser 上のモデルに対する操作の対象となるモジュールを選択することができる．状態変数リストと値テキストエリアはアニメーションオブジェクトが保持する状態をユーザが把握し必要に応じて変更するためのインターフェイスである．状態変数リストには，選択されたモジュールで定義されている状態変数が列挙され，ユーザがそのうち 1 つの状態変数を選択することで，値テキストエリアに当該状態変数の値が表示される．値テキストエリア中に VDM-SL の表現式を入力することで，当該状態変数の値を変更することができる．

ブラウザ下部の仕様テキストエリアには，モジュールリストで選択されたモジュールの VDM-SL 仕様が表示され，ユーザが編集することができる．ワークスペースは，ユーザがアニメーション実行を行うためのインターフェイスである．ワークスペース上には，VDM-SL 仕様が想定している利用シナリオとしての VDM-SL 表現式やその説明となる自然言語による記述を自由形式で記述する．ユーザは，ワークスペース上でアニメーション実行する対象となる VDM-SL 表現式を選択し，アニメーション実行を指示することで，アニメーションオブジェクトに対して VDM-SL 表現式の評価実行を指示することができる．VDMBrowser は，アニメーション実行機構としてインタプリタまたは 4.3.7 節 で説明するトランスパイラ (Transpiler) のいずれかを選択し利用することができる．

モジュールリスト，状態変数リスト，および仕様テキストエリアは精緻化を目的とする既存の VDM 開発環境でも共通して採用されているユーザインターフェイスである．値テキストエリアおよびワークスペースは探索的仕様記述を目的とする ViennaTalk に独特なものであり，ライブなプログラミング環境である Smalltalk 環境から取り入れたものである．

ViennaTalk が取り入れた元である，Smalltalk 環境のインスペクタおよびワークスペース，クラスブラウザについて説明する．Smalltalk 環境にはオブジェクトが持つインスタンス変数のリストとテキストエリアを組み合わせたインスペクタと呼ばれるユーザインターフェイスが提供されている．また，Smalltalk 環境には任意のテキストを自由形式で記述することができ，そのテキストから Smalltalk 表現式を選択して評価実行を行うワークスペースと呼ばれるユーザインターフェイスも提供されている．Smalltalk 開発者はクラスブラウザ上でクラス定義やメソッドを編集しながら，ワークスペース上に様々な利用シナリオを Smalltalk 表現式や自然

言語によるコメントを記述し，評価実行した結果として返されるオブジェクトをインスペクタを使って精査することで，クラス定義やメソッドの変更の影響を把握しながら効率的にプログラミングを行う．

ViennaTalk の VDMBrowser は Smalltalk 環境のインスペクタ，ワークスペース，および，クラスブラウザのユーザインターフェイスを VDM-SL 向けのブラウザとして統合したものである．探索的仕様記述における試行錯誤を支援するために，プログラミングにおける試行錯誤に適したライブ環境である Smalltalk 環境のツールの設計およびユーザインターフェイスを VDM-SL 記述環境に取り入れた．

4.3.3　Value パッケージ

ViennaTalk では，VDM-SL の値を表現するために可能な限り Smalltalk の標準クラスライブラリで提供されているオブジェクトを利用する．Value パッケージは，VDM-SL の値のうち Smalltalk の標準クラスライブラリでは表現できない値を表現するためのクラスや，VDM-SL の言語機能のうち Smalltalk の言語機能および標準ライブラリでは実装されていない機能を提供するクラス，および実行機構が検知した実行時の例外を表現するクラスを提供する．

表 4.2に VDM-SL の値とそれに対応する Smalltalk クラス， 表 4.3に VDM-SL の値を表現するために追加したクラス， 表 4.4に Smalltalk の標準クラスライブラリで提供されているクラスへの拡張として追加したメソッドをそれぞれ示す．

VDM-SL の複合型およびトークン型は，Smalltalk の標準クラスライブラリに直接対応するクラスがなかったために，それぞれの型の値を表現するためのクラスを定義した．VDM-SL の言語機能である関数合成や繰り返し関数合成については，既存のクラスの拡張として一般化した定義が困難であるため，合成結果を表現するクラスを定義した．

VDM-SLの言語仕様において実行可能とされる言語機能のうち，Smalltalk の言語機能では実現されていない機能として，パターンマッチング，実行時型検査，実行時表明検査が挙げられる．パターンマッチングは VDM-SL で簡潔な記述を可能にする重要な言語機能である．ViennaTalk では，`ViennaRuntimeUtil` クラスがパターンマッチングを行う機能を提供する．実行時型検査は 4.3.4 節で説明する型実装クラスが行う．実行時表明検査については，状態に対する不変条件に相当する機能が Smalltalk の標

表 4.2: VDM-SL の型とその値に対応するオブジェクトのクラス

VDM-SL の型	**Smalltalk のクラス**
nat	Integer
nat1	Integer
int	Integer
real	Float
bool	Boolean
<quote> *t1* * *t2*	Symbol Array
set of *t*	Set
seq of *t*	OrderedCollection
map *t1* to *t2*	Dictionary
inmap *t1* to *t2*	Dictionary
t1 −> *t2*	BlockClosure
token	ViennaToken
compose *t* of *f1* : *t1* *f2* :- *t2* *t3* end	ViennaComposite

表 4.3: VDM-SL の演算子を実装するための主な新規クラス

追加クラス	VDM-SL の値
ViennaComposite	複合型の値
ViennaToken	トークン型の値
ViennaComposition	関数合成された関数
ViennaIteration	繰り返し関数合成された関数
ViennaException	例外

表 4.4: VDM-SL の言語機能を実現するための既存クラスへの主な拡張

既存クラス名	メソッド名	機能
Context	viennaReturn: *value*	操作のコンテキストから return する
Collection	onlyOneSatisfy: *block* power powerDo: *block*	*block* の評価結果が真になる要素を 1 つだけ持つかを判定する 冪集合を得る 全ての部分集合を列挙してそれぞれを引数として *block* を評価する
OrderedCollection, Dictionary, BlockClosure	applyTo: *value*	*value* 番目の要素を得る
Dictionary, BlockClosure	comp: *map-or-func* $\star\star$ *value*	関数合成を得る *value* 回繰り返し関数合成を行った結果を得る

準機能には欠けている．ViennaTalk では，Pharo Smalltalk が提供するスロット機構 [VBLN11] を利用した `ViennaStateSlot` クラスによって，状態に対する実行時の不変条件の検査を実現している．クラス定義時にインスタンス変数の実装として `ViennaStateSlot` クラスを指定することで，インスタンス変数への書き込み時に自動的に不変条件が評価され，不変条件が満たされない場合には `ViennaStateInvariantViolation` 例外が発生する．

4.3.4　Type パッケージ

Type パッケージは VDM-SL の型を表現したクラス群を提供する．

Smalltalk は動的型付き言語であるため，言語要素としての型システムを持たない．一般にクラスが静的型付き言語の型に相当するが，Smalltalk のクラス階層が単一継承による木構造であるのに対して VDM-SL の型システムは直和型や不変表明があるため木構造ではない．そのため，VDM-SL の型システムを扱うために，ViennaTalk では型をオブジェクトとして表現する．VDM-SL の言語仕様において型は第 1 級オブジェクトではないが，ViennaTalk では型オブジェクトは第 1 級オブジェクトである．

`ViennaType` クラスは VDM-SL の型を表現するオブジェクトの抽象クラスであり，`ViennaNat` クラスなどの具象クラスが VDM-SL の個々の型および型コンストラクタを実装する． 表 4.5に VDM-SL の型と ViennaTalk での型オブジェクトの対応関係を示す．ViennaTalk では，値と型とクラスの関係に注意が必要である． 表 4.2には値とクラスの対応関係が， 表 4.5には型とクラスの対応関係が示されている．例えば，VDM-SL の整数値 `1` は ViennaTalk では `SmallInteger` クラスのインスタンスであるが，`nat` 型は ViennaTalk では `ViennaType nat` という表現式で生成される `ViennaNat` クラスのインスタンスである．`ViennaType nat incliudes: 1` という Smalltalk の表現式は，VDM-SL の整数値 `1` が VDM-SL の `nat` 型の値であるかどうかを判定するが，`ViennaNat` クラスのインスタンス `ViennaType nat` が，`SmallInteger` クラスのインスタンスである `1` について，`includes:` という関係が成り立つかどうかを判定している．`SmallInteger` クラスと `ViennaNat` クラスには直接的な関係はない．つまり，ViennaTalk では，Smalltalk のクラスとインスタンスの関係と，VDM-SL の型と値の関係は，完全に独立して定義されている．

ViennaType クラスは，型オブジェクトが提供すべき API として，与えられた値が当該型の値であるかどうかを判定する include:メッセージ，与えられた型が当該型の部分型であるかどうかを判定する<=メッセージ，有限要素を持つ型の値を列挙する do:メッセージ，型の不変条件を追加する inv:メッセージを宣言している．さらに，表 4.5の中列の Smalltalk 式に示された optional，*，|，set 等の，型オブジェクトを組み合わせて新しい型オブジェクトを生成する API を提供している．ViennaNat 等の VDM-SL の個々の型を実装するクラスは，これらの API を実装している．

4.3.5　Engine パッケージ

Engine パッケージは VDM-SL のインタプリタ実行機構を提供する．ただし，ViennaTalk 自体は VDM-SL のインタプリタを実装せず，Java による VDM インタプリタである VDMJ [Bat09] またはネットワーク上の VDM インタプリタサーバを利用して VDM-SL のインタプリタ実行を行う．Engine パッケージは，ViennaEngine，ViennaVDMJ，ViennaClient，ViennaBankEngine および ViennaServer の 5 つのクラスを提供する．

ViennaEngine は，VDM インタプリタの抽象クラスである．公開された API として，表現式の評価実行を行う evaluate: specification: states: module: vdm10: rtc: メッセージを宣言している．表 4.6に API の引数および返り値を示す．ViennaEngine の各具象クラスは，それぞれの実行機構によってこの API に引数として与えられた条件に従ってインタプリタ実行を行い，評価値と変更後の状態変数の値とエラーメッセージを返す．ただし，評価実行に成功した場合にはエラーメッセージは nil になり，評価実行に失敗した場合には評価値は nil になる．

ViennaVDMJ は，外部プロセスとして VDMJ を起動してプロセス間通信を通して VDM-SL のインタプリタ実行を行うクラスである．MacOS 環境および Linux 環境ではデフォルトとして利用される実行機構である．Pharo Smalltalk 環境の制約により，Windows 環境上では利用することができない．

ViennaClient は，ネットワーク上の VDM インタプリタサーバに HTTP を介してインタプリタ実行をリクエストし結果を得るクラスである．ViennaClient クラスに VDM インタプリタサーバの URL を引数と

表 4.5: VDM-SL の型を表現するクラス

VDM-SL の型	型を表すオブジェクトを得る **Smalltalk** 式	型を実装したクラス
nat	ViennaType nat	ViennaNatType
nat1	ViennaType nat1	ViennaNat1Type
int	ViennaType int	ViennaIntType
real	ViennaType real	ViennaRealType
bool	ViennaType bool	ViennaBoolType
<quote>	ViennaType quote: #*quote*	ViennaQuoteType
[*t*]	*t* optional	ViennaOptionType
t1 * *t2*	*t1* * *t2*	ViennaProductType
$t1 \mid t2$	$t1 \mid t2$	ViennaUnionType
set of *t*	*t* set	ViennaSetType
set1 of *t*	*t* set1	ViennaSet1Type
seq of *t*	*t* seq	ViennaSeqType
seq1 of *t*	*t* seq1	ViennaSeq1Type
map *t1* to *t2*	*t1* mapTo: *t2*	ViennaMapType
inmap *t1* to *t2*	*t1* inmapTo: *t2*	ViennaInmapType
t1 −> *t2*	*t1* −> *t2*	ViennaPartialFunctionType
t1 +> *t2*	*t1* +> *t2*	ViennaTotalFunctionType
token	ViennaType token	ViennaTokenType
compose *t* of *f1* : *t1* *f2* :- *t2* *t3* end	ViennaType compose: '*t*' of: {*f1* . false . *t1*. {*f2* . true . *t2*}. {nil . false . *t3*}}	ViennaCompositeType
t inv *pattern* ==*expr*	t inv: [:$v \mid expr$]	ViennaConstrainedType

表 4.6: ViennaEngine の評価実行 API

	名前	説明	クラス
引数	expression	評価式	String
	specification	仕様	String
	states	評価実行前の状態変数の値	Dictionary
	module	評価式の名前空間を提供するモジュール名	String
	vdm10	VDM の言語仕様のバージョン	Boolean
	rtc	実行時検査を行うかどうか	Boolean
返り値	result	評価式を評価実行した返り値	Object または nil
	states	評価実行後の状態変数の値	Dictionary
	message	エラーが発生した場合にはエラーメッセージ	String または nil

して渡すことで，その URL を利用するインタプリタを生成する．Windows 環境上ではデフォルトとして利用される実行機構である．

`ViennaBankEngine` は，複数のインタプリタ実行機構を束ねてラウンドロビンによって並行して複数のインタプリタ実行を行うクラスである．複数の `ViennaVDMJ` のインスタンスを保持することで複数の VDM 仕様を並行して処理する能力を向上させることができるとともに，複数の `ViennaClient` のインスタンスを保持することで複数の VDM インタプリタサーバに負荷を分散させることができる．

`ViennaServer` は，VDM インタプリタサーバをサービスとして提供するクラスである．`ViennaServer` に `ViennaEngine` のインスタンスを引数として渡すことで，そのインスタンスをインタプリタとして HTTP サーバ上でサービスを提供する．デフォルトでは，16 個の `ViennaVDMJ` のインスタンスを実行機構としてインタプリタ実行サービスを提供する．

4.3.6 Parser パッケージ

Parser パッケージは，VDM-SL のパーサである `ViennaVDMParser` クラス，および，その出力である抽象構文木 (AST) を表わす `ViennaNode` クラスを中心に構成されている．パーサは Pharo Smalltalk で実装された

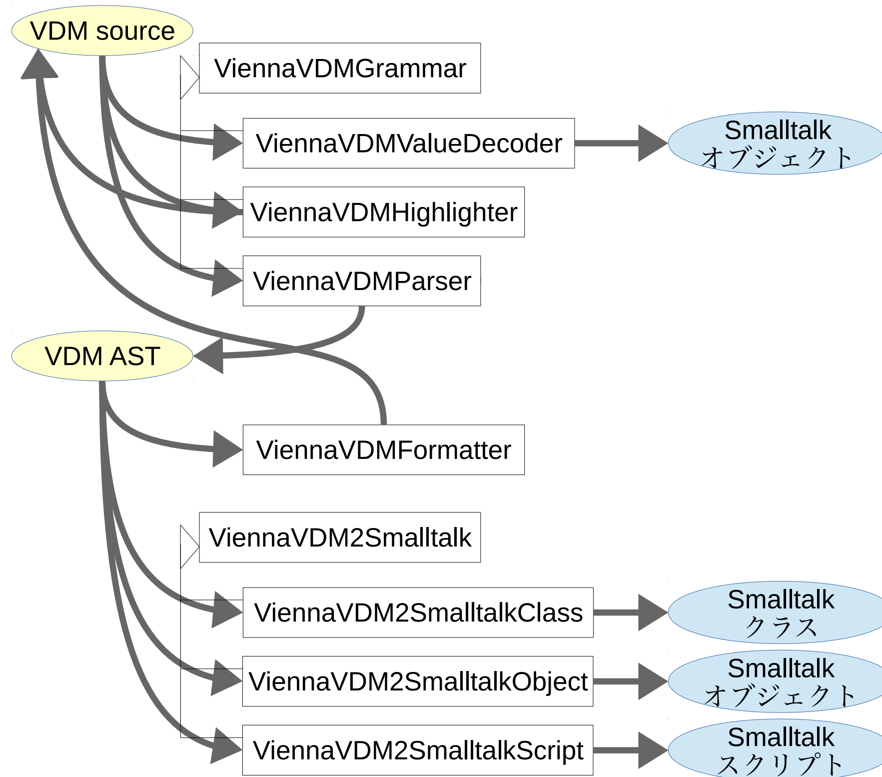

図 4.7: パーサ，抽象構文木およびソース生成に関連するクラス群

PEG (Parser Expression Grammar) パーサライブラリ `PetitParser` を利用して実装されている．図 4.7にパーサ，抽象構文木およびソース生成に関連したクラスの関係を示す．

`ViennaVDMGrammar` は VDM-SL の構文定義として各具象構文要素を受け付けるメソッドを定義している．それらのメソッドは構文規則に従って，与えられたソース文字列を受理する．`ViennaVDMParser` クラスはそのサブクラスとして実装され，各メソッドをオーバーライドして，`ViennaVDMGrammar` クラスのメソッドがソース文字列を受理した結果から抽象構文木を生成する．こうして文法定義と抽象構文木生成を別のクラスに分離することで，抽象構文木以外の出力を行うパーサを比較的容易に実装することができる．また，`ViennaVDMHighlighter` は，VDM-SL の予約語や型と式などの構文要素を色分けすることで可読性を向上させる構文ハイライトを行うクラスである．`ViennaVDMHighlighter` クラスは `ViennaVDMGrammar` クラスのサブクラスとして定義され，各メソッ

ドは抽象構文木を生成する代わりに色や字体で修飾されたソーステキストを生成する．VDM-SL のリテラル等の値の表現を読み込んで，その値に対応する Smalltalk オブジェクトを生成する ViennaVDMValueDecoder クラスも ViennaVDMGrammar クラスのサブクラスとして実装されている．

生成された抽象構文木の利用として，VDM-SL 仕様の自動整形とプログラム自動生成がある．ViennaVDMFormatter クラスは自動整形器である．ViennaVDMParser クラスによって生成された抽象構文木から整形された VDM-SL 仕様を出力する．前述の ViennaVDMFormatter クラスとを併せて，ViennaTalk 上のエディタで VDM-SL 仕様の可読性を向上させるために利用することができる．

プログラムの自動生成としては，VDM-SL から Smalltalk へのトランスパイラを 3 つ提供している．ViennaVDM2SmalltalkScript クラスは，VDM-SL の抽象構文木から Smalltalk の評価可能なプログラム片を生成する．ただし，対象となる VDM-SL 仕様はモジュール化されていないものに限る．ViennaVDM2SmalltalkClass クラスは，VDM-SL の抽象構文木から Smalltalk クラス群を生成する．対象となる VDM-SL 仕様がモジュール化されている場合には各モジュールについてそれぞれクラスが生成され，また，仕様全体を表すクラスも生成される．モジュール化されていない場合には，仕様全体を表すクラスのみが生成される．ViennaVDM2SmalltalkObject クラスは，VDM-SL の抽象構文木から Smalltalk オブジェクト群を生成する．ViennaVDM2SmalltalkObject クラスと同様に Smalltalk クラス群の生成と同じくクラス群を生成するが，生成されたクラスは匿名クラスであり，それら匿名クラス群のインスタンスが生成される点が異なる．すなわち，対象となる VDM-SL 仕様がモジュール化されている場合には各モジュールについてそれぞれオブジェクトが生成され，また，仕様全体を表すオブジェクトも生成される．モジュール化されていない場合には，仕様全体を表すオブジェクトのみが生成される．

4.3.7　トランスパイラ

本節では，前節で示したトランスパイラについて詳細を説明する．トランスパイラとは，トランスレータとコンパイラを組み合わせた造語で，ある 1 つの言語で記述されたソースから別の言語で記述されたソースを生成するプログラムを指す．コード生成器はトランスパイラの一種とみ

なすことができる．一般に形式仕様記述の文脈では，仕様とプログラムを明確に区別し，プログラムを生成するという意味で，プログラム自動生成またはソースコード自動生成という用語が用いられることが多い．一方，トランスパイラはあるプログラミング言語のソースコードから別のプログラミング言語への変換を指すことが多い．ViennaTalk では，生成されたプログラムは成果物としてのプログラムコードとしてではなく，あくまで開発対象となるシステムを理解するためのモデルであるという立場から，ある言語によるモデル表現から別の言語によるモデル表現への変換という意味で，トランスパイラという用語を使う．

VDM で記述された形式仕様記述からのプログラム自動生成は既に産業界において実用されている技術である．既存のツールとしては，VDMTools が提供する C++自動生成と Java 自動生成，および，Overture tool が提供する Java 自動生成がある．これら既存のプログラム自動生成を使って実行プログラムを作成し実行するためには，VDM 開発環境とは別に自動生成対象となる言語の開発ツールを使ってプログラムを修正しコンパイルし，リンクし，実行しなければならない．これらのツールは，形式仕様記述工程の最終段階として，成果物としてのプログラム全体または一部を作成するツールであり，仕様記述の初期段階の試行錯誤を行っている時に使うことを想定したものではない．

一方，プログラムの自動生成は探索的仕様記述においても有用な技術である．開発に必要な様々な技術領域の技術者にとって，プログラムの自動生成は成果物としてのプログラムの特性を理解する上で，コンソール上でのインタプリタによる実行よりもより正確な予測を可能にする．一般にトランスパイラで生成されたプログラムの実行はインタプリタによる実行と比較して効率的であることから，仕様アニメーションを効率的に行うための技術として用いることもできる．したがって，プログラムの自動生成は探索的仕様記述において妥当性および実現可能性を評価するために有用な技術であるといえる．

トランスパイラの設計指針

3.2 節で示した設計指針をトランスパイラに適用して，以下の設計指針によりトランスパイラの設計を行った．設計指針 1-a から設計指針 1-d は，3.2 節の設計指針 1 を ViennaTalk のトランスパイラ向けに具体化したものである．

設計指針 1 ツールは比較的小さな仕様を対象として，仕様（モデル）と直接的な対話をするためのユーザインターフェイスを提供するべきである．

探索的仕様記述においてモデルと仕様記述者の間の対話性は非常に重要であり，その実現のための 4 つの設計指針を以下に示す．

設計指針 1-a 生成されたソースコードは自動的にコンパイルされ実行することが可能であるべきである．

ユーザである仕様記述者のタスクはモデリングでありプログラミングではない．生成されたプログラムを実行するためにソースコードを改変する必要を課すことは避けるべきである．ViennaTalk のトランスパイラは生成した Smalltalk プログラムを自動的にコンパイルし，ViennaTalk 上で実行可能な状態にする．

設計指針 1-b 生成されたソースコードのコンパイルおよび実行は，ViennaTalk 内で行われるべきである．

ViennaTalk は Smalltalk が提供する統合開発環境の上で動作する．コンパイルおよび実行のために ViennaTalk 以外の開発環境に切り替えることを要求することは，統合開発環境であることの利点を阻害する．ViennaTalk では，生成されたソースコードを ViennaTalk 内で完結してコンパイルし実行する．

設計指針 1-c トランスパイラによる VDM-SL の言語仕様への制限は最小限に留めるべきである．

探索的仕様記述は仕様記述工程の初期に行われることから，仕様記述は高い抽象度で問題の本質を直接的に記述したものであることが求められる．トランスパイラによる言語仕様の制限を避けるためのミクロな工夫を課すことは単に仕様記述者への重荷であるだけでなく，可読性の低下などモデルとしての品質を劣化させることにつながる．ViennaTalk のトランスパイラはインタプリタ実行が可能な VDM-SL 仕様は全て Smalltalk プログラムに変換することができる．

設計指針 1-d 生成されたソースコードのデバッグが可能であるべきである．

探索的仕様記述での仕様記述は暫定的なものであり，不正確で，誤りを含んでいる可能性がある．したがって生成されたソースコードにも誤りが含まれ，生成されたソースコードのデバッグが必要になる可能性がある．そのため，生成されたソースコードはプログラマが記述したソースコードと同様の方法でデバッグが可能でなければならない．ViennaTalkのトランスパイラは，プログラマが記述したソースコードに近いソースコードを生成する．Smalltalk に習熟したプログラマは，トランスパイラに生成されたプログラムについて，Smalltalk プログラムとして誤りの箇所を同定し，修正することができる．

設計指針 2 ツールは形式仕様記述言語の知識がない者にも理解可能な表示インターフェイスを持つべきである．

探索的仕様記述では，ドメイン専門家や様々な技術領域の技術者からのフィードバックが求められる．トランスパイラが生成したプログラムもそれらのステークホルダーとのコミュニケーションに利用される．ViennaTalkのトランスパイラが生成したプログラムは，Smalltalk 上の GUI フレームワークを利用してユーザインターフェイスを構築したり可視化技術を適用することができる．

設計指針 3 ツールは仕様記述者により検査の厳格さの度合いを選択することが可能であるべきである．

探索的仕様記述を支援するツールは暫定的で，不正確で，誤りを含んでいる仕様記述を対象として受け入れる必要がある．トランスパイラが疑わしい仕様記述を拒絶すると，探索的仕様記述ではトランスパイラの利用が実質的に困難になる．ViennaTalk のトランスパイラは，ユーザによる設定に応じて，生成する Smalltalk プログラム中に静的型検査や実行時の諸検査を行うコード片を生成するかどうかを選択することができる．

設計指針 4 ツールは継続的な分析が可能であるべきである．

一般に，トランスパイラが生成したプログラムは，インタプリタよりも効率的に実行できる．仕様の単体テストの実行も効率的に行うために，トランスパイラで生成したプログラムをテストフレームワークで自動テスト可能であることが望ましい．ViennaTalk のトランスパイラが生成した Smalltalk プログラムは，Pharo Smalltalk が提供するテストフレームワークおよび Jenkins を利用して継続的に自動単体テストを行うことができる．

```
module ExampleCounter
exports all
definitions
values
  modulus = 1000;
types
  Count = nat inv c == c < modulus;
functions
  succ : Count -> Count
  succ(x) == (x + 1) mod modulus
  pre x < modulus
  post RESULT < modulus;
state Counter of
  count : Count
  inv mk_Counter(c) == c < modulus
  init s == s = mk_Counter(0)
end
operations
  get : () ==> Count
  get() == return count
  post RESULT < modulus;
  inc : () ==> ()
  inc() == count := succ(count);
  reset : () ==> ()
  reset() == count := 0;
end ExampleCounter
```

図 4.8: Smalltalk プログラムを自動生成する元となる VDM-SL 仕様例

トランスパイラの設計

図 4.8に本節でコード生成の対象とする VDM-SL 仕様例を示す.

この仕様は，1000 を法 (modulus) とする合同算術 (modular algebra) によるカウンタの仕様である．関数 `succ` が，`modulus` を法とする合同算術での 1 を加算する関数として定義されている．システムの状態として，カウント値 `count` が定義されていて，初期値は 0 である．3 つの操作が定義されており，`get` は現在のカウント値を答える．`inc` はカウント値を合同算術での 1 を加算して更新する．`resett` はカウント値を 0 にする.

以下に，この仕様の各要素がどのように Smalltalk プログラムに変換されるかを説明する.

```
ViennaTranspiledObject subclass: #ExampleCounter
        slots: { #post_get. #inc. #Count. #Counter.
        #init_Counter. #inv_Counter. #reset. #post_succ.
        #state. #modulus. #get. #succ. #pre_succ.
        #count => ViennaStateSlot }
        classVariables: {  }
        category: 'Auto␣Generated␣from␣VDM'
```

図 4.9: Smalltalk プログラムを自動生成する元となる VDM-SL 仕様例

モジュール

モジュールは `ViennaTranspiledObject` クラスのサブクラスとして定義される．以下，モジュールに対応する生成されたクラスをモジュールクラスと呼ぶ．モジュールクラスは対応する VDM-SL モジュールが宣言する個々の型，定数，関数，状態変数，操作をインスタンス変数として保持するとともに，型，定数，関数，および操作へのアクセスを提供する． 図 4.9に生成されたクラス定義を示す．

型定義

VDM-SL 仕様で定義された型は， 4.3.4 節に示された ViennaType クラスのオブジェクトとして実装される． 図 4.8中の `Count` 型に対応して生成された `Count` メソッドを 図 4.10に示す．`Count` 型は VDM-SL 仕様では型不変条件が宣言されている．Smalltalk には不変条件に相当する言語機能は提供されていないが，ViennaTalk では VDM-SL の型をオブジェクトとして実装することで，型不変条件を実装している．`Count` 型は `nat inv c == c < modulo` と定義されているが，Smalltalk では自然数型オブジェクト `ViennaType nat` に不変条件を定義する `inv: [:c | c < self modules]` メッセージ送信することで定義される．VDM-SL での定義と Smalltalk による実装は表現が似ており，対応する定義を容易に認識することができる．

定数

VDM-SL 仕様中の値は， 4.3.3 節の 表 4.2に示された対応関係によってオブジェクトとして実装される． 図 4.8中の定数 `modulus` に対応して生

```
Count
  ^ Count
    ifNil: [ Count := ViennaTypeHolder new.
      Count
        type: (ViennaType nat
           inv: [ :c | c < self modulus ]).
      Count ]
```

図 4.10: 生成された型定義の例

```
modulus
  ^ modulus
    ifNil: [ modulus := 1000.
      modulus ]
```

図 4.11: 生成された定数定義の例

成された modulus メソッドを 図 4.11に示す.

関数定義

VDM-SL では関数は第一級のオブジェクトである．すなわち，値として状態変数に保持すること，関数や操作への引数として渡すこと，また，関数や操作の返り値とすることができる．一般に Smalltalk プログラムでは関数に相当する処理はメソッドとして定義する．しかし，関数を第一級のオブジェクトとして扱うためには，クロージャオブジェクトとして実装するのが Smalltalk プログラムとして自然である．以上の理由から，トランスパイラは関数はクロージャオブジェクトとして実装し，そのクロージャオブジェクトにアクセスするためのメソッドを定義する．

また，VDM-SL では，関数定義に事前条件や事後条件が宣言されている場合には，それらの事前条件や事後条件を引用した関数が定義される．ViennaTalk ではそれらも同時にクロージャとして実装する．図 4.12に生成された succ メソッドを示す．succ メソッド内で，pre_succ，post_succ および
succ の 3 つのクロージャオブジェクトが順に定義され，そのうち succ が返り値として返される．pre_succ および post_succ はそれぞれ事前条件および事後条件を引用した関数である．

```
succ
  ^ succ
    ifNil: [ pre_succ := [ :x |
      (self Count includes: x)
        ifFalse: [ ViennaRuntimeTypeError signal ].
      x < self modulus ].
      post_succ := [ :x :RESULT |
      (self Count includes: x)
        ifFalse: [ ViennaRuntimeTypeError signal ].
      (self Count includes: RESULT)
        ifFalse: [ ViennaRuntimeTypeError signal ].
      RESULT < self modulus ].
      succ := [ :x |
      (self Count includes: x)
        ifFalse: [ ViennaRuntimeTypeError signal ].
      [ | RESULT |
      RESULT := [ x < self modulus
        ifFalse: [ ViennaPreconditionViolation signal ].
      (x + 1) \\ self modulus ] value.
      (self Count includes: RESULT)
        ifFalse: [ ViennaRuntimeTypeError signal ].
      RESULT < self modulus
        ifFalse: [ ViennaPostconditionViolation signal ].
      RESULT ] value ].
      succ ]
```

図 4.12: 生成された関数定義の例（実行時検査を行う場合）

```
succ
  ^ succ
    ifNil: [ pre_succ := [ :x | x < self modulus ].
      post_succ :=
        [ :x :RESULT | RESULT < self modulus ].
      succ := [ :x | (x + 1) \\ self modulus ].
      succ ]
```

図 4.13: 生成された関数定義の例（実行時型検査を行わない場合）

```
inv_Counter
  ^ inv_Counter ifNil: [
  inv_Counter := [ :_inv |
  | c |
  (((ViennaRuntimeUtil
    matchTuple:
      {(ViennaRuntimeUtil
        matchRecord: 'Counter'
        args: {(ViennaRuntimeUtil matchIdentifier: 'c')})})
          value: {_inv})
    ifEmpty: [ false ]
    ifNotEmpty: [ :binds |
      c := binds first at: 'c'.
      true ])
    ifFalse: [ ViennaNoMatch signal ].
  c < self modulus ]]
```

図 4.14: 生成された不変条件関数の例

また，ViennaTalk のトランスパイラは実行時型検査および実行時表明検査を行わないプログラムを生成することもできる．実行時型検査および実行時表明検査を行わない場合の succ メソッドを 図 4.13に示す．

状態定義

VDM-SL 仕様の状態変数は Smalltalk のインスタンス変数として実装される． 図 4.9にあるように，VDM-SL の状態変数 `count` は，Smalltalk のインスタンス変数 `count` として実装される．図 4.8において状態 `Counter` には状態不変条件が宣言されている．ViennaTalk のトランスパイラは不変条件を引用したクロージャ`inv_Counter` を生成するとともに，不変条件検査と実行時型検査を行う `inv` メソッドを生成する． 図 4.14に引用された `inv_Counter` メソッドの定義を， 図 4.15に `inv` メソッドの定義を示す．

生成されたコードが状態変数 `count` を更新する度に `inv` メソッドを実行することで，実行時型検査と不変条件の検査が行われる．ViennaTalk では，Pharo Smalltalk のスロット機構 [VBLN11] を利用して，インスタンス変数への書き込みに対する実行時検査が実現されている． 図 4.9では，インスタンス変数 `count` が `count=>ViennaStateSlot` と定義されてい

```
inv
  (self inv_Counter  value: self state)
    ifFalse: [ self stateInvariantViolation ].
  (self Count includes: count)
    ifFalse: [ ViennaRuntimeTypeError signal ]
```

図 4.15: 生成された不変条件の例

る．ViennaStateSlot は，Smalltalk コンパイラに対して，インスタンス変数 count への代入に対して常に inv メソッドを実行するバイトコードを生成するよう指示する．Smalltalk コンパイラは，ExampleCounter クラスのメソッドをコンパイルする時に，インスタンス変数 count への代入文に対して，当該変数への書き込みに加えて inv メソッドの呼び出しを行うバイトコードを生成する．これによって，インスタンス変数 count への代入のたびに，状態不変条件および実行時型検査が検査される．

操作定義

VDM-SL の操作は第一級オブジェクトではないことから，Smalltalk ではメソッドとして実装される．ただし，呼び出し方法について関数と共通するインターフェイスを提供するために，クロージャとしても扱うことができるインターフェイスを提供する． 図 4.16に get 操作を実装する get:メソッド，操作の実行部を実装する_get メソッド，および，クロージャを提供する get メソッドを示す．

また， 図 4.9において，操作 get は事後条件が宣言されている．事後条件を引用した関数 post_get を実装した post_get メソッドを 図 4.17に示す．引用された事後条件は関数であることから，クロージャとして実装されている．

4.4 VDMPad

本節では，ViennaTalk 上に実装されたツールの 1 つである VDMPad の目的，設計指針，実装，および評価を示す．VDMPad は VDM-SL による探索的仕様記述を支援する統合開発環境 (IDE) である [OA13, OAL15]. VDMPad はウェブ上で IDE を提供する Web IDE であり，利用者はウェ

```
get: _op
  | _oldState RESULT |
  _oldState := self state.
  RESULT := self _get.
  (self Count includes: RESULT)
    ifFalse: [ ViennaRuntimeTypeError signal ].
  RESULT < self modulus
    ifFalse: [ ViennaPostconditionViolation signal ].
  ^ RESULT

_get
  ^ count
get
  ^ get
    ifNil: [ get := [ self get: nil ].
      get ]
```

図 4.16: 生成された操作の例

```
post_get
  ^ post_get
    ifNil: [ post_get := [ :_State :_oldState :RESULT |
      (self Count includes: RESULT)
        ifFalse: [ ViennaRuntimeTypeError signal ].
      RESULT < self modulus ].
      post_get ]
```

図 4.17: 生成された操作事後条件の例

ブブラウザから VDMPad サーバにアクセスすることで，手軽に VDM-SL の仕様記述環境を利用することができる．

VDMPad は基本的にコマンドライン用 VDM-SL インタプリタに柔軟で使いやすいユーザインターフェイスをウェブ上で提供するツールである．VDMPad の主な機能は，VDM-SL 仕様の編集と，仕様アニメーションである．顧客との対話でのスケッチ的な利用に対応するために，必要な時にすぐに利用できるようログイン操作なしで利用可能であり，保存のための明示的な操作をすることなしに前回利用した時の仕様とアニメーション状態を復元して利用することができる．VDMPad のユーザインターフェイスは， 3.2 節で示した設計指針 1「ツールは比較的小さな仕様を対象として，仕様（モデル）と直接的な対話をするためのユーザインターフェイスを提供すべきである」に基いて設計された．ユーザインターフェイスは直感的な操作を可能にするために単純化され，仕様記述やアニメーション操作は，各テキスト領域への記述と「evaluate」ボタンで可能である．仕様記述の負荷を軽減するための機能として構文ハイライトやソースの自動整形が提供されるとともに，VDM の値を直感的に理解できるように値を図的な表現で表示することができる． 図 4.18に VDMPad の画面例を示す．

4.4.1　アニメーションとの対話

VDMPad のアニメーションの実行機構として既存の VDM インタプリタを利用しているが，探索的仕様記述に適合するようインタラクションのモデルが大きく変更されている．ここでは VDM インタプリタのインタラクションモデルを比較対象として，VDMPad のインタラクションモデルを説明する．

従来の VDMJ インタプリタのインタラクションは Read-Eval-Print Loop (REPL) と呼ばれるインタラクションモデルに基づいている．REPL とは，ユーザからの文字列入力を読み（Read），読み込んだ文字列に従って評価実行を行い（Eval），評価実行の結果を表示する（Print）ことを繰り返す（Loop）ことで，ユーザとインタプリタの間のインタラクションを行うモデルである．

REPL インタラクションモデルに基づいた VDM インタプリタでは，ユーザである仕様記述者が対話を行う対象はツールである．仕様をテキストエディタ等で記述しファイルに保存し，VDM インタプリタ上でファイ

VDMPad

powered by Squeak Smalltalk with VDMJ
about VDMPad

```
 1 module 金銭管理
 2 imports from 中央制御部 operations 残高が変わった;
 3 exports all
 4 definitions
 5 types
 6     硬貨 = <百円硬貨>| <五十円硬貨>| <十円硬貨>;
 7     枚数 = nat;
 8     現金 = map 硬貨 to 枚数;
 9     金額 = nat;
10
11 values
12     硬貨の金額 =
13       {<百円硬貨> |-> 100, <五十円硬貨> |-> 50, <十円硬貨> |-> 10};
14     チューブ容量 = 20;
15
16 state State of
17     チューブ : 現金
18     投入残高 : 金額
19 inv mk_State(チューブ, -) ==
20     forall この硬貨 in set dom チューブ &
21       チューブ(この硬貨) <= チューブ容量 -- 物理制約
22 init s ==
23     s = mk_State({<百円硬貨> |-> 0, <五十円硬貨> |-> 0, <十円硬貨> |-> 0}, 0)
```

Format

仕様エリア

金銭管理

チューブ　{<五十円硬貨> |-> 18, <十円硬貨> |-> 11, <百円硬貨> |-> 20}

五十円硬貨	十円硬貨	百円硬貨
18	11	20

投入残高　0

0

Initialize

状態エリア

```
金銭管理`十円硬貨が投入された()
金銭管理`五十円硬貨が投入された()
金銭管理`百円硬貨が投入された()
金銭管理`精算ボタンが押された()
```

evaluate

make it a testcase

ワークスペース

()

()

結果エリア

メッセージエリア

図 4.18: VDMPad の画面例

ルを読み込む命令を実行することでアニメーションを開始する．そしてVDM-SL表現式を評価実行する命令を実行することでアニメーション実行を行う．このインタラクションモデルでは，仕様はファイルの中に格納された文字列であり，インタプリタ上での直接的な操作対象ではない．また，アニメーション実行するためのVDM-SL表現式はインタプリタへの命令の一部として入力されるため，ユーザにとってツールに与える命令の一部にすぎず，直接的な操作対象とは言い難い．

VDMPadでは，ユーザである仕様記述者がアニメーションと対話することを意図したインタラクションモデルを採用した．インタプリタに対して命令を与えるREPLと異なり，VDMPadのユーザインターフェイスは，アニメーションの構成要素であるVDM-SL仕様と各状態変数の値と評価式をユーザに常に表示し，また，ユーザからの編集を許すことで，ユーザがVDM-SL仕様や状態変数や評価式と対話する手段として設計されている．

VDM-SL仕様は常に仕様エリア上に表示され編集可能である．また，アニメーション状態も状態エリア上に各状態変数ごとに値が表示され編集可能である．変数の値は図的な表現によって直感的に把握することが可能なように表示される．評価実行するVDM-SL表現式は，ワークスペース上に表示され編集可能である．図4.19にワークスペースの使用例を示す．ワークスペースには複数のVDM-SL表現式を予め記入してから，各表現式を選択して評価実行することができる．すなわち，ワークスペースは対象システムが想定する利用シーンのシナリオを記述したものといえる．ユーザである仕様記述者は，評価実行したいVDM-SL表現式をワークスペース上で選択し,「evaluate」ボタンを押すことで，指定されたVDM-SL表現式がインタプリタ実行され，その結果が表示される．状態エリアがVDM-SL表現式を評価実行した後の状態に更新され，表現式の評価値が結果エリアに表示される．この評価値も図的な表現で表示される．アニメーション実行機構はライブ性が確保されているため，アニメーションを初期化することなしに仕様とアニメーション状態を常に変更可能である．アニメーション実行機構の内部状態を意識する必要はなく，アニメーション実行に関する全ての情報は一覧性を確保した形で表示され，常に編集が可能であり，ユーザである仕様記述者は目の前のVDMPadのユーザインターフェイスをアニメーション実行中のモデルそのものであると見做して，モデルと対話することができる．ツールに対する対話は「evaluate」ボタンを押す操作のみで，残りはモデルである仕様とアニメーション状

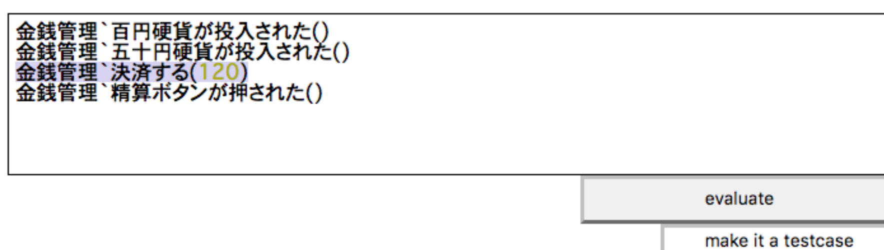

図 4.19: ワークスペース上に記述されたアニメーションのシナリオの例

態と実行シナリオとの対話である．このユーザインタラクションにおいて，VDMPad はスケッチ的な仕様記述を支援するために，アニメーションとの直接的な対話のためのユーザインターフェイスを提供していると言える．そして，構文ハイライトや自動整形や図的表現によって，仕様記述者とアニメーションの間の対話を支援している．

また，VDM インタプリタはデバッグ用のステップ実行や実行カバレッジの表示，証明責務の生成など，仕様の精緻化において有用な機能が豊富に実装されている．しかし VDMPad では探索的仕様記述でのスケッチ的な仕様記述に特化したインタラクションを実現するために，それらの機能の多くにはユーザインターフェイスを提供していない．

4.4.2 VDM-SL の値の図的表現

VDM-SL の値の表記は形式仕様技術者やプログラミングの知識を持つ人には難しいものではないが，それ以外のステークホルダーにとっては読解の障壁となりうる．集合や列，レコード，写像等の構造を持つ値を図的に表現することで，技術的背景の少ないステークホルダーにとっての障壁を下げることができる．またトークン型やクォート型の値のように構造を持たない値であっても，型の種類が視覚的に判別できるようにすることで，読解の負荷を減じることができる．例えば，図 4.18では VDMPad 上に示された状態変数 `stock` の値は {`<BEER> |-> 10, <WINE> |-> 3`} という写像であるが，図的表現では写像が表形式で示され，クォート型の値`<BEER>` および `<WINE>` は灰色を背景色とした矩形枠の中に表示されることで，文字列などの類似した値と区別される．図的表現は形式仕様記述に慣れないステークホルダーにも直感的な理解を与えることができる．また，VDM-SL に熟達した技術者にとっても認知負荷の低減に役

型	値	図的表現
real	1.0	1
quote	<quote>	quote
seq of char	"abc"	"abc"
seq	[1, 2, 3, 4]	1 2 3 4
set	{1, 2, 3, 4}	1 2 3 4
map	{<one>|->1, <two>|->2 }	one two 1 2
product	mk_tuple(1, "abc")	1 "abc"
composite	mk_Record(1, "abc")	Record 1 "abc"
token	mk_token(0)	token 0

図 4.20: VDMPad における VDM-SL の値の図的表現の一覧

立つ．このことから，図的表現は 3.2 節で示した設計指針 2「ツールは形式仕様記述言語の知識がない者にも理解可能な表示インターフェイスを持つべきである」に適合している．図的表現の表示例を 図 4.20 に示す．

4.4.3　実行時検査の設定

VDMPad では，ユーザ設定により実行時型検査および実行時表明検査を無効化することができる．実行時検査を無効にすることによって，ユーザである仕様記述者は表明や型による保護から逸脱した場合の潜在的な振る舞いを見ることで，記述対象の特性についてより広い理解を持つことができる．実行時検査の無効化は注意をもって扱われる必要があり，通常時は実行時検査が有効に設定されていることが推奨される．この設定機能により，VDMPad は 3.2 節で示した検査の厳格さに関する設計指針 3 を満たす．

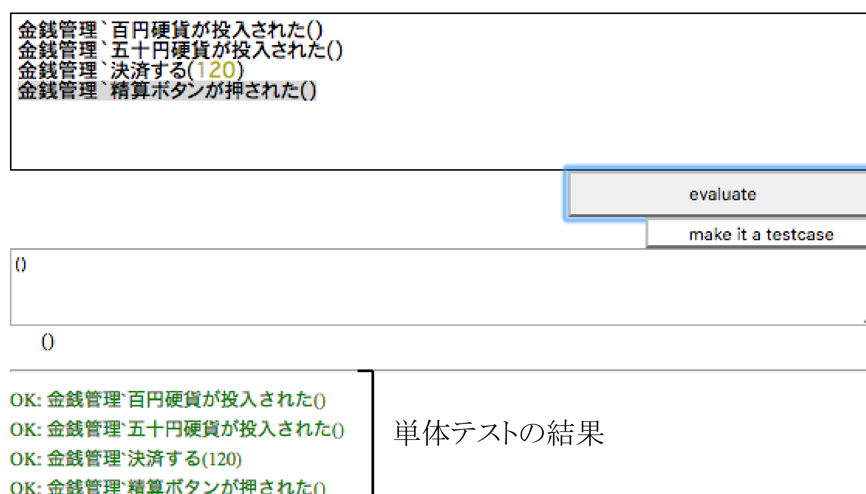

図 4.21: VDMPad での継続的単体テストの画面例

4.4.4　継続的単体テスト

3.3.6 節で示されたように，単体テストはプログラミングにおいて xUnit 等のテストフレームワークにより支援され，多くの開発で実施されている．従来の単体テストフレームワークでは，テストケースの実行は定期的に自動実行される場合や技術者が明示的に実行の指示を与える場合がある．

VDMPad での単体テストは前述のテストフレームワークと異なり，仕様記述者とステークホルダーの対話の中で合意した仕様の振る舞いからの逸脱を検知するためのツールである．テストによる品質向上を目的とした工程で行われるテストとは異なり，スケッチ的な仕様記述での単体テストではモデリング以外の作業負荷を最小限に抑えることが望ましい．そのため，VDMPad では仕様記述者に明示的なテストケースの記述を要求しない．

VDMPad ではテストケースは仕様アニメーション実行の履歴から作成される．VDMPad の設定で単体テストを ON にした状態でアニメーション実行をすると，「evaluate」ボタンの下に「make it a testcase」ボタンが表示される（ 図 4.21参照）．仕様記述者とステークホルダーがその動作で良いと合意をしたら「make it a testcase」を押す．すると，そのアニメーション実行前の状態，アニメーション実行した表現式，アニメーション実

```
FAILED: 金銭管理`精算ボタンが押された() => Runtime: Error 4131: State invariant violated: inv_State in '金
銭管理' (console) at line 1:1

prestates : {"金銭管理`チューブ":"{<五十円硬貨> |-> 19, <十円硬貨> |-> 0, <百円硬貨> |-> 20}","金銭管理`投入残高":"30"}
expression : 金銭管理`精算ボタンが押された()
value : ()
poststates : {"金銭管理`チューブ":"{<五十円硬貨> |-> 19, <十円硬貨> |-> 0, <百円硬貨> |-> 20}","金銭管理`投入残高":"30"}
delete
```

図 4.22: 示された動作特性からの逸脱が見つかったテストケースの表示例

行の返り値，アニメーション実行後の状態，の 4 つ組がテストケースとして保存される．次回以降，アニメーション実行が行われるたびに，保存されたテストケース群が順次実行され，それぞれの結果が表示される．合意した振る舞いと異なる結果が得られたテストケースは 図 4.22に示されたように赤く表示される．これにより，探索的仕様記述で仕様を頻繁に修正しても，合意した振る舞いからの逸脱があった時点で，仕様記述者はその逸脱を把握することができる．

このインタラクションでは，単体テストのためにユーザに追加された作業は「make it a testcase」ボタンを押すことのみであり，モデリング以外の作業負荷を最低限に抑えている．また，VDMPad の継続的単体テストは 3.2 節の継続的な分析に関する設計指針 4 を満たしている．従来の単体テストフレームワークでの定期的実行やユーザによる明示的な実行と異なり，仕様の変更がありアニメーション実行が行われるたびにテストが実行されるため，ユーザである仕様記述者は合意された特性からの逸脱を早期に発見し，仕様の再修正や合意の変更により対処する機会が与えられる．

4.5　Lively Walk-Through

Lively Walk-Through は探索的仕様記述において形式手法の技術者と UI デザイナが協力して UI プロトタイプを構築する環境である．UI プロトタイプを構築し試用し議論することを通して，形式手法の技術者と UI デザイナがそれぞれの成果物についての共通理解や合意を形成することが目的である．

ユーザインターフェイス設計はインタラクティブなシステムを構築する上で重要な工程であり，形式手法の技術者と UI デザイナの協力関係はインタラクティブなシステムの利用性を高める上で重要である [NY13].

ユーザはユーザインターフェイスを通してシステムの機能を理解し操作を行う．ユーザインターフェイスの設計は単に意匠的なデザインだけでなく，システムの機能とユーザの認知を結びつけるための重要な設計項目である．UI デザイナはシステムの機能項目だけでなく，表示デバイスの物理的な大きさ，解像度，色調や，入力デバイスの種類や特性に応じて，かつドメイン固有の問題を考慮した上で，ユーザとシステムの間の対話モデルを構築し，実現可能かつ利用性の高いユーザインターフェイスを設計する役割を持っている．ユーザインターフェイスが働きかけるユーザの認知とシステムの機能が乖離すると，ユーザにとってたとえシステムの機能が仕様通りに実装されていたとしても，ユーザの意図と反する結果となる．

一方，システムの機能仕様を策定する上で，適切な機能項目を定義するためには，ユーザが要求する情報や操作の粒度を理解することが必要である．ユーザとシステムの間で発生する対話をモデリングし設計することはUI デザイナの重要な専門技術である．従って，システムの機能を定義する形式仕様を記述するためには，UI デザイナが実現可能で利用性が高いと判断したユーザインターフェイス設計を理解し，そのユーザインターフェイスの動作を実現可能な機能や情報の粒度で機能を定義する必要がある．

システムの機能を定義する形式仕様とユーザインターフェイス設計は相互に依存しており，どちらか一方を先に定義して他方を後で合わせることは，利用性の高いインタラクティブなシステムを構築する上で適切ではない．形式仕様技術者とUI デザイナがお互いに他方の成果物を理解し，共通理解を形成して，合意する部分と修正すべき部分を議論し，擦り合せることが望ましい．しかし，形式仕様とユーザインターフェイス設計は専門分野として異なる知識や技能を要求することから，お互いに他方の成果物を直接理解することが困難である．

Lively Walk-Through は，専門知識や技能が異なる形式仕様技術者と UI デザイナが協同して UI プロトタイピングを行い，UI プロトタイプの振る舞いを理解することで，共通理解や合意を形成するための環境である．形式仕様技術者の視点からは，対象ドメインに適合したユーザインターフェイスを構築するための妥当性の確認であり，また，現実的なユーザインターフェイスが構築可能という意味での実現可能性の確認と捉えることができる．また，UI デザイナが設計したユーザインターフェイスを備えることで，形式仕様の技術的背景を持たないステークホルダーやエ

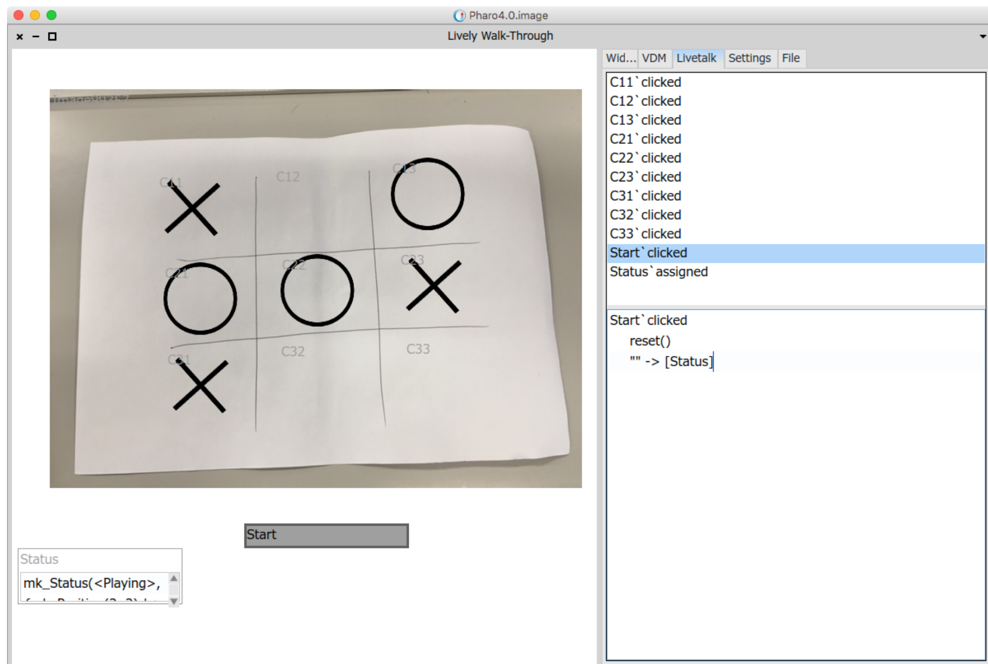

図 4.23: Lively Walk-Through の画面例

ンドユーザ代表が形式仕様の意味するところを理解するための仕様アニメーションとして利用することができる．

探索的仕様記述を支援するためのツールとしては，Lively Walk-Through は設計指針 1 の小規模なモデルに対する直接的な対話をするためのインターフェイスを備えていると言える．Lively Walk-Through は仕様を編集する環境として 4.3.2 節で示した VDMBrowser を埋め込んでいることから，ライブなアニメーションが可能であることも，設計指針 1 を満たしている．また，設計指針 2 の形式仕様記述言語の知識がないステークホルダーにも利用可能な表示インターフェイスを提供するツールであると言える．VDMBrowser を通してアニメーション実行機構としてトランスパイラが利用可能であることから，設計指針 3 の検査の厳密さの選択可能性および設計指針 4 の継続的な分析については 4.3.7 節で示したトランスパイラと同様に満たしている．

図 4.23 に Lively Walk-Through の画面例を示す．形式仕様技術者が VDM-SL で記述された実行可能な形式仕様を画面右側に埋め込まれた VDM-Browser 上に提供し，UI デザイナが画面左側に GUI 部品を使ったユーザインターフェイスを提供する．画面右側で LiveTalk と呼ばれる専用言語

で，GUI 部品のイベントや入力内容と VDM-SL 仕様を結合することで，VDM-SL 仕様を GUI 部品を通して実行することができる．

4.6　Webly Walk-Through

Webly Walk-Through は実行可能な VDM-SL 仕様を Web API として利用可能にするためのプロトタイプ用サーバである．ウェブアプリケーションの開発では工期の短縮が重要なファクターとなることが多い．Web API はウェブアプリケーションのクライアント側がサーバ側の機能を利用するためのインターフェイスであり，ウェブアプリケーションの開発では適切な Web API の設計が必須である．Webly Walk-Through は Web API の仕様を VDM-SL で記述する形で形式仕様を導入することで，Web API のサーバ側の実装の生産性および信頼性の向上を図るとともに，クライアント側の開発にも Web API のプロトタイプ実装として利用可能にするものである．

また，ウェブアプリケーション以外にも Web API によってコンポーネント間を疎結合するために利用することができる．軽量形式手法では，開発対象の重要な一部のみに形式仕様を適用し，残りの部分は従来の自然言語による仕様記述によって開発することがある．形式仕様で記述した部分を Web API としてプロトタイプを提供することで，残りの部分を実装言語を問わず Web API を通して形式仕様部分と結合することができる．Webly Walk-Through は Web API のパラメータや返り値のデータフォーマットとして JSON を採用している．JSON は一般に広く普及したテキスト形式の構造を持ったデータフォーマットであり，多くの実装言語でライブラリとして提供されている．Webly Walk-Through によって形式仕様部分を結合可能にすることによって，形式仕様を導入しない部分の開発にも，形式仕様による非曖昧性の利益を享受することができる．

図 4.24に Webly Walk-Through の画面例を示す．画面例では，マイクロ SNS サーバの機能仕様を VDM-SL で記述し，Web API として提供している．クライアント側は HTML および JavaScript により実装されている． 図 4.25にクライアントの画面例を示す．マイクロ SNS クライアントとしてウェブブラウザ上で HTML ファイルを開いている．ウェブ画面上段には SNS コンテンツを書き込むためのテキストウィジェットと「Blah!」と書かれた送信ボタンが配置されている．送信ボタンをクリックすると，JavaScript によってサーバ側である Webly Walk-Through の Web API にア

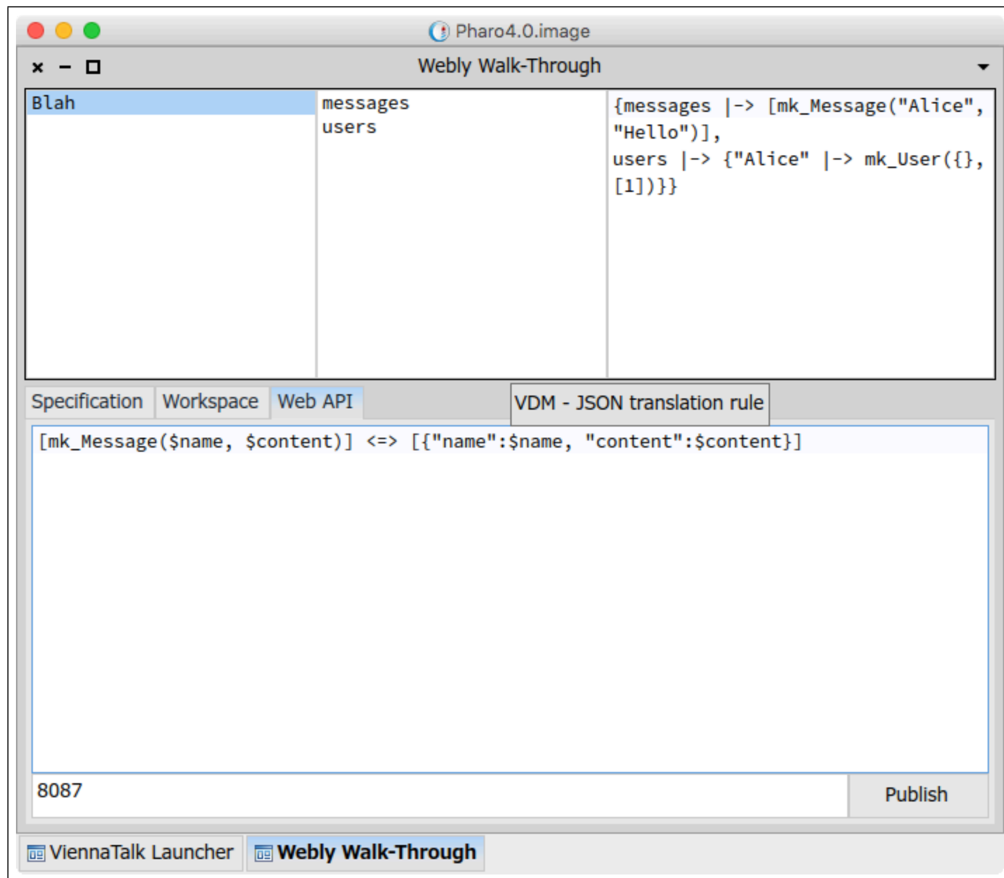

図 4.24: Webly Walk-Through の画面例

クセスし，新規コンテンツとしてサーバに登録される．ウェブ画面下段には，他のユーザが書き込んだコンテンツが表示されている．これらのコンテンツも，JavaScript によって Webly Walk-Through の Web API を通してサーバから取得されたものである．このようにして，VDM-SL で記述されたサーバ側の Web API の仕様をプロトタイプとしてウェブブラウザ上のクライアントに提供することができる．

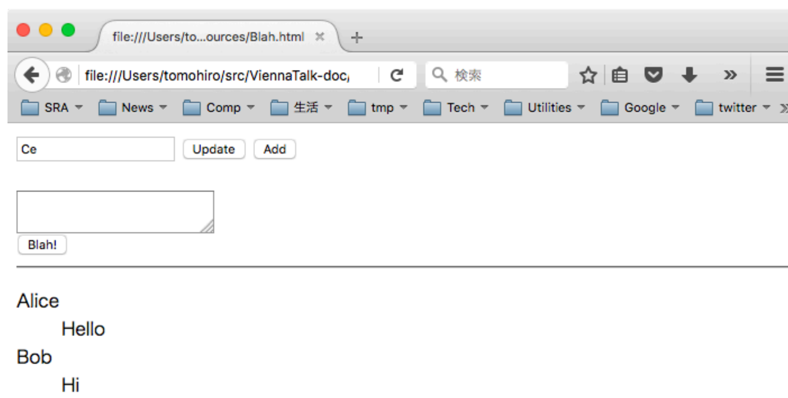

図 4.25: Webly Walk-Through の Web API を利用するウェブクライアントの画面例

第5章　ViennaTalk の評価

本章では，ViennaTalk の評価として，ViennaTalk を構成する重要なコンポーネントであるトランスパイラと，探索的仕様記述を直接的に支援する VDMPad に対する評価を示した上で，ViennaTalk 全体に対する VDM の有識者による評価を示す．ViennaTalk の特徴の 1 つとして，探索的仕様記述での試行錯誤における仕様アニメーションの実行機構としてトランスパイラが利用可能である点が挙げられる．インタプリタと比較してトランスパイラを積極的に利用する理由としては，生成されたプログラムの性能と，GUI などのユーザインターフェイスや外部コンポーネントとの結合の自由度が挙げられる．5.1 節に，ViennaTalk のトランスパイラが出力したプログラムに対する性能評価と可読性評価を示す．

また VDMPad は探索的仕様記述を支援するための Web IDE であり，VDMPad 単体について 3.2 節で示した設計指針が正しく実現されたか，また，VDMPad が探索的仕様記述に向いているかを評価することで，設計指針および VDMPad の実装の妥当性を評価する．妥当性を評価する上で，ドメイン専門家である VDM 技術者からの評価が重要であると考え，産業界で長年の間 VDM を使った開発を実践してきた熟練技術者複数名に，いくつかの質問項目を提示することで，VDMPad の探索的仕様記述における妥当性を評価した．

ViennaTalk 全体に対する評価も，VDMPad 同様にドメイン専門家による妥当性の評価を行った．ViennaTalk 全体については，産業界で VDM を実践してきた熟練技術者に加えて，長年 VDM に取り組んできた学術研究者をドメイン専門家として，いくつかの質問項目への回答を得ることで，妥当性を評価した．

5.1　トランスパイラの評価

探索的仕様記述において，トランスパイラは効率的なアニメーション技術として利用されることがある．また，データベースやネットワーク

表 5.1: 処理系ごとのベンチマークの結果

Tool	インタプリタ/コード生成器	生成言語	処理時間 (ms)	処理時間 **(Overture=1)**
VDMTools	Interpreter	C++	22,044	79.6
VDMJ	Interpreter	Java	5,281	19.1
VDMTools	Code generator	C++	337	1.22
Overture tool	Code generator	Java	277	1.00
ViennaTalk	Code generator	Smalltalk	193	0.700

サービスなど外部のコンポーネントやサービスを利用したシステムのプロトタイプとしてトランスパイラによるプログラムの自動生成を行う場合にも，性能上の実現性をより正確に見積もるためにはトランスパイラが生成したプログラムの性能は重要である．

本節では，探索的仕様記述のためのトランスパイラとして ViennaTalk のトランスパイラの性能評価，および 4.3.7 節で挙げたデザイン指針を満たすかどうか論じる．

5.1.1　性能評価：素数列の生成

性能評価のためのベンチマークとして，2 から 10000 までの自然数から素数列を取り出す VDM-SL 仕様について，既存のインタプリタやコード生成器によってプログラムを自動生成し，性能を比較した．用いた VDM-SL 仕様を 図 5.1に示す．

このベンチマークは VDMTools および VDMJ によるインタプリタ実行，VDMTools により生成された C++コード，Overture tool により生成された Java コードと，ViennaTalk によって生成された Smalltalk プログラムにより実行され，それぞれの実行時間を計測した．実行環境は，i5-3210M CPU，クロック周波数 2.50GHz，2 コア，4G バイトの主記憶上の仮想計算機，OS は Ubuntu 16.04 上である．処理系は，C++コンパイラは gcc version 5.4.0，Java 処理系は Java HotSpot(TM) 64-Bit Server VM(build 25.101-b13)，Smalltalk 処理系は Pharo 4 with 32-bit Cog VM を使った． 表 5.1に結果を示す．

一般に期待されているように，コード生成による実行はインタプリタ実行と比較してより効率よく処理を実行した．計測対象となったインタ

```
state Eratosthenes of
    space : seq of nat1
    primes : seq of nat1
init s == s = mk_Eratosthenes([], [])
end
operations
    setup : nat1 ==> ()
    setup(x) ==
        (space := [k | k in set {2, ..., x}];
        primes := []);
    next : () ==> [nat1]
    next() ==
        if space = [] then
          return nil
        else let x = hd space in
          (primes := primes ^ [x];
           sieve(x);
           return x);
    sieve : nat1 ==> ()
    sieve(x) ==
        space := [space(i)
                  | i in set inds space
                  & space(i) mod x <> 0];
    prime10000 : () ==> seq of nat1
    prime10000() ==
        (setup(10000);
        while next() <> nil do skip;
        return primes);
```

図 5.1: ベンチマークとして仕様した VDM-SL 仕様

プリタおよびコード生成器の中では ViennaTalk のトランスパイラが最も効率よく処理を実行した．したがって，効率の良いアニメーション実行器として有用であるといえる．

一般に，C++ソースからコンパイルされたバイナリプログラムは，Java VM 上で動作する Java プログラムよりも実行効率が良く，また，一般に Java VM 上で動作する Java プログラムは Smalltalk VM 上で動作する Smalltalk プログラムよりも実行効率が良いことが知られている．表 5.1に示したベンチマーク結果は逆の順位となっている．1 つのベンチマーク結果から原因を導くことは困難だが，考えられる説明の 1 つとして ViennaTalk のトランスパイラは Smalltalk の標準ライブラリが提供する機能を多く利用していることが挙げられる．図 5.2に Overture tool により生成された `sieve` 操作の Java ソースコードを示す．標準ライブラリにはない VDMSeq クラスや VDMSet クラスを使い，数値を Number クラスにボックス化して，計算には `.longValue()` によりプリミティブ型の `long` 型に変換して計算している．図 5.3に同じ `sieve` 操作から ViennaTalk により生成された Smalltalk ソースコードを示す．Smalltalk 標準ライブラリの機能である `select:thenCollect:` メッセージによって制御構造を作り，数値は Smalltalk 標準の Integer クラスのオブジェクトにより計算を行っている．これにより，80 年代から継続して改良されてきた Smalltalk のクラスライブラリの性能を生かしている．Overture tool が生成した Java ソースコードが VDMSet 等の専用のクラスを定義しているのは，VDM-SL が求める機能の中には Java の標準ライブラリでは直接提供されていないものがあるため，その実装のために新規クラスを定義して利用しているからである．一方，Smalltalk では標準クラスライブラリに対してもメソッドを追加するなどをして機能を拡張することができる．ViennaTalk では，表 4.4に挙げた拡張をしており，Smalltalk 標準クラスライブラリの長年にわたる性能向上の成果を利用することができる．これが ViennaTalk のトランスパイラが生成したプログラムの実行性能が既存のソースコード生成器のものよりも優れた結果を出した要因の 1 つであると考えられる．

5.1.2　設計指針の観点からの評価

ベンチマークは各ツールでターゲット言語のソースコードに変換され，ターゲット言語の処理系によりコンパイルされ実行された．VDMTools や Overture tool が生成した C++および Java プログラムはコンパイルし実行

```
public static void sieve(final Number x) {
  VDMSeq seqCompResult_2 = SeqUtil.seq();
  VDMSet set_2 = SeqUtil.inds(Eratosthenes.space);
  for (Iterator iterator_2 = set_2.iterator();
        iterator_2.hasNext();) {
    Number i = ((Number) iterator_2.next());
    if (!(Utils.equals(Utils.mod(
      ((Number) Utils.get(Eratosthenes.space, i)).longValue(),
      x.longValue()), 0L))) {
        seqCompResult_2.add(
          ((Number) Utils.get(Eratosthenes.space, i)));
    }
  }
  Eratosthenes.space = Utils.copy(seqCompResult_2);
}
```

図 5.2: Overture tool により生成された Java ソースコード

```
_sieve_: x
    space := ((1 to: space size)
            select: [ :i | (space value: i) \\ x ~= 0 ]
            thenCollect: [ :i | space value: i ])
        asOrderedCollection
```

図 5.3: ViennaTalk により生成された Smalltalk ソースコード

するために，それぞれの開発環境外にターゲット言語の開発環境を構築し，その上でコンパイルされ実行する必要があった．ViennaTalk 上でトランスパイラを利用するためには，VDMBrowser 上のメニューを選択するだけで，自動的にライブラリが生成された．したがって，生成されたコードの自動コンパイルに関する設計指針 1-a および ViennaTalk 内での処理に関する設計指針 1-b は満たされている．

生成対象となる VDM-SL 仕様への制限事項としては，VDMTools および Overture tool にはパターンマッチの利用など自動生成ではサポートされない言語機能があった．ViennaTalk のトランスパイラはインタプリタ実行可能な言語機能は全てコード生成の対象としている．したがって，VDM-SL の言語仕様への制限に関する設計指針 1-c は満たされている．

可読性の点においては，図 5.3に示したソースコードはプログラマが直

接書いたものと遜色ない．デバッグにおいても，Smalltalk 環境に標準のデバッガを利用して，プログラマが書いたプログラムと同様にデバッグすることが可能である．したがって，設計指針 1-d に挙げたデバッグ可能性は満たされている．

また，ViennaTalk のトランスパイラは文字，文字列，数値，真偽値，集合，写像，組などの値を Smalltalk 標準ライブラリが提供するクラスを使って表現するため，Smalltalk 環境の既存の UI フレームワークと親和性が高い．レコードやトークンなど ViennaTalk が独自に提供するクラスについても API を Smalltalk での標準的な API に準拠して設計したため，Smalltalk の既存のプログラムと容易に結合することができる．したがって，設計指針 2 に挙げた VDM-SL の知識がない者にも理解可能なインターフェイスを持つことは満たされている．

ViennaTalk のトランスパイラは 図 4.12および 図 4.13に示した通り，それぞれの実行時検査について選択的に有効化/無効化することができる．したがって，設計指針 3 にあげた検査の厳格さの度合いを選択可能であることは満たされている．

ViennaTalk のトランスパイラから生成されたプログラムは Smalltalk のテストフレームワークで自動単体テストの対象とすることができる．したがって，設計指針 4 にあげた継続的な分析が可能であることは満たされている．

以上により，ViennaTalk が提供するトランスパイラは探索的仕様記述を支援するためのツールの設計指針に適っていると言える．

5.2　VDMPad の評価

探索的仕様記述を支援するツールとして VDMPad に対するユーザ評価を行った．VDMPad は探索的仕様記述を総合的に支援するツールであり，かつ，ウェブブラウザ上で利用するツールであることから ViennaTalk 上で操作する他のツールとは異なる操作性を持っていることから，VDMPad を単体として評価した．

VDMPad の妥当性を評価するためには,「VDM による開発」という対象ドメインの専門家の知識および経験が重要である．そこで，VDM の既存の開発環境である VDMTools および Overture tool の利用経験を 3 年以上持つ熟練 VDM 技術者を評価者として選定し，VDMPad を 2 年以上利用した上での評価を依頼した．評価者はそれぞれ実際の開発現場におい

表 5.2: VDMPad 評価アンケートの各設問に対する回答の選択肢

評価	スコア
1	既存 IDE が優れている
2	どちらかというと既存 IDE が優れている
3	どちらともいえない
4	どちらかというと VDMPad が優れている
5	VDMPad が優れている

て長年 VDM に取り組んできた技術者であり，仕様記述の実務やコンサルテーションや教育を通して VDM のいくつかの成功事例に貢献した経験を持っている．評価者による VDMPad への妥当性評価は，彼等の VDM による開発の経験に裏付けられた深い理解に基づいている．また，VDMPad を評価するための基準として， 3.2 節で示した設計指針に関連した評価項目を設定し，既存の開発環境である VDMTools および Overture tool との比較による評価を行った．評価者はそれぞれの設問に対して， 表 5.2に示した選択肢から選択する形式で VDMPad を評価した．

各設問とそれに対するスコアの平均値，および関係する設計指針を 表 5.3に示す．

b-1 と b-2 では小規模な仕様記述が高いスコアを得ている．c-1 から c-8 までの仕様記述の初期段階と最終段階の比較では，初期段階で高いスコア，最終段階で低いスコアとなった． 表 5.4に探索的仕様記述に関する平均スコアと精緻化に関する平均スコアをまとめる．探索的仕様記述に関するスコアが詳細化に関するスコアを大きく上回っていることから，探索的仕様記述に特化された開発環境であるという評価を得たと言える．

各設問のスコアから，設計指針ごとに平均を求めた結果を 表 5.5に示す．全てのデザイン指針について，VDMPad は既存の開発環境と比較して高く評価された．

4.4 節において示した通り，VDMPad は設計指針 1 から 4 に基いて設計された．ユーザ評価によって，VDMPad が設計指針を正しく実装したことが確認されたとともに，大規模と小規模の比較および初期段階と最終段階の比較による評価結果から，探索的仕様記述に適したツールであることが確認された．総合すると，VDMPad は探索的仕様記述を支援するツールとして VDM の専門技術者から肯定的に受け入れられたと言える．

表 5.3: VDMPad 評価アンケートの結果

設問	平均スコア	設計指針
a-1: 全体として使いやすいか	3.3	1, 2, 3, 4
a-2: UI はわかりやすいか	3.3	1
a-3: アニメーションしながらの仕様の修正がしやすいか	4.0	1
a-4: 非技術者への説明に使いやすいか	4.3	2
a-5: 妥当性の検証に向いているか	3.0	2, 3, 4
b-1: 小規模な仕様に向いているか	4.3	1
b-2: 大規模な仕様に向いているか	0.7	
c-1: 仕様記述初期の試行錯誤に向いているか	4.3	1, 2, 3, 4
c-2: 仕様記述最終段階の品質向上に向いているか	2.3	
c-3: アニメーション実行が仕様記述初期の試行錯誤に向いているか	4.7	1
c-4: アニメーション実行が仕様記述最終段階の品質向上に向いているか	1.0	
c-5: 静的検査機能が仕様記述初期の試行錯誤に向いているか	4.7	3
c-6: 静的検査機能が仕様記述最終段階の品質向上に向いているか	1.0	
c-7: テストフレームワークが仕様記述初期の試行錯誤に向いているか	4.0	4
c-8: テストフレームワークが仕様記述最終段階の品質向上に向いているか	1.0	

表 5.4: VDMPad の探索的仕様記述と詳細化に関する評価結果

	設問	平均スコア
探索的仕様記述	b-1, c-1, c-3, c-5, c-7	4.4
詳細化	b-2, c-2, c-4, c-6, c-8	1.2

表 5.5: VDMPad に関するデザイン指針ごとの評価結果

デザイン指針	設問	平均スコア
1	a-1, a-2, a-3, b-1, c-1, c-3	4.5
2	a-1, a-4, a-5, c-1	3.7
3	a-1, a-5, c-1, c-5	3.8
4	a-1, a-5, c-1, c-7	3.7

5.3　ViennaTalk の評価

探索的仕様記述における ViennaTalk の妥当性の評価を行った．ViennaTalk の妥当性を評価するドメイン専門家として，産業界で VDM を複数の実開発プロジェクトに適用し成功させた経験を持つ熟練技術者 2 名と，ソフトウェア工学の研究を通して VDM に関する深い理解を持つ大学の研究者 1 名に評価を依頼した．評価結果は，産業界の技術者 2 名は VDM による開発の経験に裏付けられた深い理解に，また，大学の研究者はソフトウェア工学および形式手法に関する高度な学術的知識に裏付けられている．3 名の評価者は，ViennaTalk 上でいくつかの比較的小規模 (モジュール数で 2 から 5 程度）な仕様記述を行なった上で，ViennaTalk と既存 IDE を比較し評価した．

仕様記述において行われる 6 種のタスクについて，ViennaTalk と既存 IDE を比較し，それぞれについて，表 1 に従ってスコアを回答するよう求めた．結果を表 2 に示す．これらのタスクのうち，タスク 1 およびタスク 2 は探索的仕様記述で主に行われ，タスク 3 およびタスク 4 は部分的に探索的仕様記述においても行われる．ViennaTalk はスコア平均値ではタスク 1 からタスク 4 において従来 IDE よりも適していると判断された．

主に探索的仕様記述で行われるタスク 1 およびタスク 2 が比較的スコアが高く，主に精緻化で行われるタスク 5 およびタスク 6 は低いスコアとなり，探索的仕様記述と精緻化の両方で行われるタスク 3 およびタスク 4 はその中間的なスコアであったことから，ViennaTalk が探索的仕様記述に適した開発環境であることを支持していると考えられる．

個々のスコアに着目すると，回答者 A はタスク 1 にスコア 1 をつけたが，ViennaTalk での編集操作に慣れていないというコメントが回答者 A から挙げられたことから，編集操作の改善によりスコアが向上する可能

表 5.6: ViennaTalk と既存 IDE の比較基準

既存 IDE のほうが適している	1
どちらかというと既存 IDE のほうが適している	2
どちらとも言えない	3
どちらかというと ViennaTalk のほうが適している	4
ViennaTalk のほうが適している	5

性がある．同じタスク 1 に回答者 C は,「スケッチとして使う際の気軽さは ViennaTalk が一番」として，スコア 5 を与えている．回答者 B は Smalltalk 環境に習熟しプロトタイプの作成に長けており，タスク 2 に対してスコア 5 を与えている．回答者 B はタスク 6 に対してはスコア 1 を与えているが,「実装言語が Smalltalk ならば 5」とコメントしており，回答者 C も同様に「Smalltalk で実装するなら勿論 5. 様々なコード生成をこの先活用できそうなら 4〜5 になるかも」とコメントしていることから，タスク 6 に対してはコード生成を重視し実装言語により評価が変化している．

表 5.7: タスクごとの ViennaTalk と既存 IDE の比較結果

タスク	スコア			
	A	B	C	平均
1. 仕様記述がない状態から仕様の草案を記述する	1	4	5	3.3
2. 仕様の妥当性を確認する	4	5	3	4.0
3. 仕様記述中の機能項目の詳細な定義の誤りを発見する	4	4	2	3.3
4. 仕様記述の論理的不整合を発見する	4	3	3	3.3
5. 大きな記述量の仕様記述をメンテナンスする	2	2	2	2.0
6. 仕様記述に基づいて設計および実装を行う	2	1	3	2.0

第6章　結論

本研究は形式仕様記述の生産性を向上させることを目的としている．ソフトウェア開発には仕様記述，設計，実装言語，オペレーティングシステム，データベース，ユーザインターフェイス，テスト，運用，プロダクトラインをはじめとする様々な工学的な技術が用いられており，その全てがソフトウェア開発の生産性や成果物としてのソフトウェアシステムの品質に影響する．加えて，ソフトウェア開発にはソフトウェア技術者だけでなく，ドメイン専門家，プロダクトオーナー，エンドユーザ等の様々な関係者が関与する，学際的で複雑な協力関係によって進められる．その中で形式仕様記述が果たす役割とは，ソフトウェア開発に創造的な問題解決としてのゴール条件を定義することである．開発するソフトウェアシステムが何であるのかを形式言語によって明確に記述することで，様々なステークホルダーが共通理解を持ってそれぞれの専門領域で開発に貢献し，また，ソフトウェア開発がゴール条件を満たしたかどうかを確認する技術が形式仕様記述といえる．

ソフトウェア開発に形式仕様を導入することで，仕様記述工程のコストが増大するものの全体としての開発コストを抑制することが知られている．問題解決の視点からは，形式仕様記述工程は本質的に悪定義問題であるソフトウェア開発について，明確な定義を与えることで後工程を良定義問題に変えていると見なすことができる．しかし形式仕様記述工程自体は悪定義問題であることから，系統的な問題解決が困難なまま残っている．本研究は，形式仕様記述工程を悪定義問題である探索的仕様記述と良定義問題である精緻化に分離し，探索的仕様記述が持つ困難と必要とされる支援を示すことによって，形式仕様記述の生産性を向上させるものである．

本研究の最も重要な貢献は，形式仕様記述工程を探索的仕様記述と精緻化の 2 つの工程に分離した点にある．困難の質が異なる 2 つの工程に分離することで，従来から研究が進んでいる型理論や証明責務生成および証明支援による精緻化での系統的な問題解決の他に，形式仕様の学際性

やコミュニケーションの問題を探索的仕様記述での困難として特定し，その解決を支援するための環境が満たすべき性質を明らかにした．そして，探索的仕様記述とその支援ツールの設計指針の実証として，探索的仕様記述支援環境 ViennaTalk を開発した．技術的評価として，ViennaTalk に搭載されたコード生成器の性能および生成コードの可読性を評価し，既存ツールに対し優位であることが示された．また，ViennaTalk の探索的仕様記述での妥当性が，VDM での経験が豊かな熟練技術者および学術研究者により確認され，有用性が示された．

本章では，本研究の学術的貢献を示すとともに，Hall が挙げた形式手法 7 つの神話 [Hal90] について探索的モデリングの観点から考察する．

6.1　学術的貢献

形式手法は数学的な裏付けのある道具立てを使ってシステムを開発する手法の総称である．形式手法をソフトウェア開発へ適用することはソフトウェア工学の一領域として捉えることができる．ソフトウェア工学の研究には，数学やシステム理論など，関連する多くの研究分野の成果が取り込まれている．認知科学もソフトウェア工学に関連する研究分野の 1 つである．

本研究は，認知科学での問題解決に関する研究で知られる知見を形式手法に取り込んだ試みである．本研究がもたらす貢献を，以下に形式手法の領域ごとに示す．

6.1.1　開発手法

従来の形式仕様による開発手法では，仕様記述を段階的に行う場合には記述対象の種別（型や定数，関数，状態，操作など）や仕様記述の抽象度によっていて，仕様記述工程での良定義問題と悪定義問題の分離がされていなかった．本研究では，認知科学での問題解決の研究の知見である良定義問題と悪定義問題を仕様記述工程に適用し，工程を探索的仕様記述と精緻化に分離することで，悪定義問題である探索的仕様記述に特化した困難を解決するアプローチを示した．

探索的仕様記述では，形式手法が持つ数学的な道具立てだけでなく，記述対象への理解やコミュニケーションといった人間の認知活動に関わる道具立てが必要であり，その道具立てとしての支援ツールを設計するため

の指針を明らかにした．学際的なコミュニケーションとして UI プロトタイピングを取り上げ，仕様アニメーションにおいて形式手法の知識を前提としないインタラクションを実現することで，これまで形式手法による支援が不足していたユーザインターフェイス設計において，形式手法の適用可能性を広げた．また，形式仕様を Web API としてアニメーション実行させるツールを実現することによって，形式仕様と下流工程の間の連携を強化した．

6.1.2 ツール

形式手法はソフトウェア開発を数学の対象として扱うための道具立てであり，主として数学的な裏付けによって開発者を支援する支援ツールが開発され利用されてきた．それらの支援ツールは形式手法に習熟した技術者をユーザとして想定して設計された．本研究では探索的仕様記述において形式仕様記述に関するコミュニケーションや理解の問題に取り組み，そのための設計指針を示した．

探索的仕様記述およびその支援ツールの設計指針の概念実証として ViennaTalk を開発した．ViennaTalk 上において，探索的仕様記述における試行錯誤を支援する Web IDE として VDMPad を開発し評価することによって，設計指針の妥当性を評価し，既存ツールと比較して探索的仕様記述においてより妥当性が高いことが示された．トランスパイラによって，外部の開発環境に依存しないプログラム自動生成を実現し，高速な仕様アニメーションを手軽に利用できるようにした．トランスパイラが生成した Smalltalk プログラムについて，性能および可読性を評価し，既存ツールに対する優位性を示した．また，Smalltalk 環境が得意とする動的なユーザインターフェイスを利用して，UI プロトタイピング環境を実現した．Smalltalk 環境の特徴であるライブプログラミング技術とイメージベース環境が形式手法のツール構築においても有効であることを，VDM に熟練した技術者による ViennaTalk の評価により示した．これらのツール機能の実現とその評価は，社会的および認知的な対象としての形式仕様記述に関する理解を深めることに貢献した．

6.1.3　仕様の編集

ライブプログラミング技術を形式仕様に応用した．既存のツールでは仕様の編集とアニメーション実行は別の作業として実装されていた．アニメーション実行中に仕様の編集が可能なライブなアニメーションにより，問題を発見したその場で修正し，その効果を確認できることで，探索的仕様記述を支援することを示した．VDMPad で，ツールとユーザの間での対話の頻度を高めるユーザインターフェイスを実現し，ユーザ評価によってその有効性を示した．

6.1.4　コミュニケーション

従来の形式仕様記述による開発事例では，形式手法の技術的背景を持たない関係者とのコミュニケーションでは自然言語による非形式的記述や UML や表形式による準形式的記述によって形式仕様記述の内容を説明していた．本研究では，形式仕様記述の役割としてコミュニケーションを明示し，解決策として UI プロトタイピング環境 Lively Walk-Through や VDMPad でのデータの可視化を実現し，形式手法の知識を求めることなく仕様に関する議論をすることを可能にした．また，Web API サーバやトランスパイラによって，外部システムとの結合を容易にした．実行可能仕様によるプロトタイプは技術要素としては従来から取り組まれてきた課題であり，実装そのものの新規性は高くない．それらの技術が有効な場面として探索的仕様記述を示した点が本研究の貢献である．形式仕様記述の初期段階における，妥当性の獲得や機能項目の網羅性の獲得および実現可能性の獲得を行う工程として探索的仕様記述を定義し，その工程の中で様々な対象ドメインおよび技術ドメインの専門家とのコミュニケーションを通した知識獲得のプロセスを示し，それら専門家とのコミュニケーションツールとして，ViennaTalk およびそのコンポーネントである VDMPad の妥当性を評価した．妥当性の評価の結果，探索的仕様記述におけるコミュニケーションに，既存ツールと比較してより適していることが示された．

6.1.5　導入教育

VDMPad は形式仕様入門セミナーや大学での講義で利用され，講師および学習者の双方から肯定的な評価や感想が得られた．また，VDMPad

サーバを公開し，2014 年からの 3 年 6ヶ月で 56,400 件のアクセスがあり，VDMPad 上で 82,248 回仕様のアニメーションが実行された．日本や欧州を中心に，VDM をソフトウェア工学の教育に取り入れている大学からも多くのアクセスがあり，肯定的な評価を受けた．VDM を学ぶことは単に記法を記憶するだけでなく，演習を通してモデリングを実際に行うことが必要である．演習でのモデリングは精緻化以前にまず探索的な記述が求められることから，探索的仕様記述のための設計指針が導入教育においても有用であると考えられる．

6.2　形式手法 7 つの神話

形式手法 7 つの神話 [Hal90] は Hall によって示された形式手法に関する典型的な誤解とそれに対する反論を示したものである．Hall は形式手法一般についての見解を示したが，本節では形式仕様記述の中でも探索的モデリングの観点に立った議論を示す．7 つの神話は主に，数学は難解であり，一部の専門家にしか理解できない，手間のかかる，証明のことであるという前提に基づいている．特に，難解であり一部の専門家にしか理解できないという前提から，ステークホルダー，特に顧客と仕様記述者の共通理解の妨げになるとして，それが形式仕様の問題点として指摘されている [Fis94]．成功事例の調査により形式仕様は共通理解の妨げではなく，むしろ共通理解の中心に位置付けられていることが報告されている． 2.4 節で示した通り，本研究で示した探索的仕様記述でのコミュニケーションを介した試行錯誤においても，仕様記述は多様な領域の専門家との共通理解を構築する上で中心的な役割を担っている．以下に，7 つの神話それぞれについて探索的モデリングの観点から論じる．Hall の主張を補足することになっていると見なすことができる．

Formal methods can guarantee that software is perfect.

Hall が指摘した通り，形式手法はソフトウェアが完璧であることを保証するものではない．形式仕様は開発するソフトウェアがどのような性質を持つものであるかを明確に示し，その性質の根拠を検証可能にし，また，後工程での自動化を推進するための技術的なアプローチである．探索的仕様記述においては，どのようなソフトウェアを開発するかが多くのステークホルダーによって妥当性と機能項目の網羅性および実現可能

性が合意されることが，目標の 1 つとなる．また，探索的仕様記述は妥当性，機能項目の網羅性および実現可能性を高めるための工程であり，それらの品質において完璧であることを保証するものではない．

Formal methods are all about program proving.

探索的仕様記述では仕様記述の遵法性といった数学的な正しさだけでなく，共通理解と合意事項といったステークホルダーのコミュニケーションと認知に関する問題が重要である．ドメイン専門家や実装技術者など多くのステークホルダーと共通理解を形成して妥当性や実現可能性についての合意事項を積み上げる．その過程において，実行可能な形式仕様であることを利用して仕様アニメーションを実行したり，また，特定の性質について証明を含む分析や検討を行うことがある．しかしそれらは妥当性や実現可能性を確認するための手段であり，全てではない．

Formal methods are only useful for safety-critical systems.

探索的仕様記述は，高い信頼性が求められるシステム開発だけでなく，妥当性や実現可能性の確認が重要なソフトウェア開発に有用である．開発を遂行する組織がまだ経験を積んでいない新しい対象ドメインのためのソフトウェアを開発する場合においては，生産性の向上など経済性を重視したシステムの開発にも適用することができる．

Formal methods require highly trained mathematicians.

探索的仕様記述の段階で仕様記述者に求められる数学的な知識は，プログラミングで求められる数学的知識と大きな違いはない．適切なツールを利用することで，技術的な知識がないステークホルダーにもソフトウェアの仕様を理解しどのようなシステムが出来上がるかを知ることができる．したがって，仕様記述者も仕様の妥当性や実現可能性を検討する担当者も，高度な訓練を受けた数学者である必要はない．

Formal methods increase the cost of development.

探索的仕様記述の目的の 1 つは，コストに見合ったものができるかどうか判断する材料としての仕様記述を提供することである．

Formal methods are unacceptable to users.

探索的仕様記述では顧客やエンドユーザが形式仕様記述の妥当性の確認に参加することを想定している．探索的仕様記述のためのツールは，システム開発に関与したい顧客やエンドユーザにとって有力な道具になる．

Formal methods are not used on real, large-scale software.

形式手法は実際の大規模なソフトウェア開発にも適用され成功した事例が多くある．それらの事例では形式仕様記述工程において妥当性や実現可能性の確認はされていることから，探索的仕様記述も実施可能であると考えられる．ただし，探索的仕様記述と精緻化を分離して形式仕様記述工程を遂行した事例はまだ存在しない．従って，実際の大規模ソフトウェア開発に探索的仕様記述を精緻化から分離して適用可能であるかどうかはまだ実証されていない．

6.3　まとめと今後の展望

本研究は，形式仕様記述工程を探索的仕様記述と精緻化の 2 つの工程に分離した．精緻化は従来からの形式手法の研究が進められてきた型理論や証明に基づいた系統的な開発工程である一方，探索的仕様記述で求められる学際性やコミュニケーションの問題については形式手法での研究は少ない．本研究では，それらの問題を探索的仕様記述での困難として特定し，その解決を支援するための環境が満たすべき性質を明らかにした．そして，概念実証として探索的仕様記述支援環境 ViennaTalk を開発し，探索的仕様記述における ViennaTalk の妥当性を確認した．

本研究の成果は，形式手法を適用したソフトウェア開発手法での形式仕様記述工程だけでなく，支援ツールの開発，仕様に関するコミュニケーション，および，導入教育に新たな視点を与えた．本研究は形式仕様記述言語として VDM-SL を例として採用したが，研究成果は特定の形式仕

様記述言語に限定されるものではない．今後は，形式手法の中では新たな領域である探索的仕様記述について，探索的仕様記述に向いた新しい形式仕様記述言語の設計や，個々の対象ドメインや技術ドメインに着目した研究をはじめとして，探索的仕様記述をより深く掘り下げた研究に発展することを期待する．

謝辞

私を形式手法の世界に導き，以後長きに渡って熱心にご指導いただき，本論文の執筆を勧めて下さった，九州大学大学院システム情報科学研究院情報知能工学部門 荒木啓二郎教授に深く感謝致します．

本論文の調査委員として貴重なご助言とご指導を賜りました，九州大学情報基盤研究開発センター学術情報研究部門の廣川佐千男教授，九州大学大学院システム情報科学研究院情報知能工学部門の鵜林尚靖教授，九州大学マス・フォア・インダストリ研究所数学理論先進ソフトウェア開発室溝口佳寛教授に，心から御礼申し上げます．

長崎県立大学情報システム学科の日下部茂教授，名古屋大学情報科学研究科の山本修一郎教授，デンマーク オーフス大学工学科の Peter Gorm Larsen 教授，英国ニューカッスル大学ソフトウェア信頼性センターの John Fitzgerald 教授，南山大学数理情報学部情報通信学科の張漢明准教授，宮崎大学工学部情報システム工学科の片山徹郎准教授，筑紫女学園大学人間科学部人間科学科の持尾弘司准教授，九州大学大学院システム情報科学研究院情報知能工学部門の大森洋一助教から仕様記述についてソフトウェア工学の様々な研究領域の視点から多くのご討論とご助言を頂きました．深謝いたします．

形式手法の実践について多くの有益な助言をいただいた法政大学大学院情報科学研究科の佐原伸兼任講師，有限会社デザイナーズ・デンの酒匂寛氏，ソニー株式会社の栗田太郎博士，Fujitsu UK の Nick Battle 氏，株式会社メタテクノの宮本陽子氏，新谷 IT コンサルティングの新谷勝利氏に深謝します．

株式会社 SRA 先端技術研究所所長・京都大学学際融合教育研究推進センターデザイン学ユニットの中小路久美代特定教授からは，人間を中心にした開発環境のデザインについてご指導いただくとともに，本研究の遂行にあたって強くご支援をいただきました．深謝いたします．

米国コロラド大学ボウルダー校計算機科学科の Gerhard Fischer 名誉教授，京都大学学際融合教育研究推進センター デザイン学ユニットの山本

恭裕特定教授，米国アビリーンクリスチャン大学情報技術計算機学部の Brent Reeves 准教授，葉雲文博士から，学際的な協調を支援するためのデザインやドメイン指向のデザインについてご指導いただきました．深く感謝します．

株式会社 SRA の石曽根信社長，岸田孝一最高顧問は研究活動を深く理解くださり，本研究の遂行を支援してくださいました．深く感謝します．

同僚として本研究に協力してくださいました，株式会社 SRA の土屋正人氏，松原伸人氏，方学芬氏，鶴田範子氏，川辺義勝氏に感謝します．

最後に，常に私を励まし，本研究の遂行に様々な面で協力してくれた妻，晶に感謝します．

参考文献

[Abr07] Jean-Raymond Abrial. Formal methods: Theory becoming practice. *Journal of Universal Computer Science*, Vol. 13, No. 5, pp. 619–628, May 2007.

[ADL+91] J.R. Abrial, S.T. Davies, M.K.O. Lee, D.S. Neilson, P.N. Scharbach, and I.H. Sørensen. The B Method. Technical report, Information Science and Engineering Branch BP Research, Sunbury Research Centre, October 1991. Also published at the VDM '91 symposium.

[AEF97] Ernesto Arias, Hal Eden, and Gerhard Fisher. Enhancing communication, facilitating shared understanding, and creating better artifacts by integrating physical and computational media for design. In *Proceedings of the 2nd conference on Designing interactive systems: processes, practices, methods, and techniques*, pp. 1–12. ACM, 1997.

[AL98] Sten Agerholm and Peter Gorm Larsen. A Lightweight Approach to Formal Methods. In *Proceedings of the International Workshop on Current Trends in Applied Formal Methods*, Boppard, Germany, October 1998. Springer-Verlag.

[Bat09] Nick Battle. VDMJ User Guide. Technical report, Fujitsu Services Ltd., UK, 2009.

[BBB+09] Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal versus agile: Survival of the fittest? *IEEE Computer*, Vol. 42, No. 9, pp. 37–45, September 2009.

[BDN+09] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[BH95] Jonathan P. Bowen and Michael G. Hinchey. Ten Commandments of Formal Methods. *IEEE Computer*, Vol. 28, No. 4, pp. 56–62, April 1995.

[DNBM12] Torgeir Dingsøyr, Sridhar Nerur, VenuGopal Balijepally, and Nils Brede Moe. A decade of agile methodologies: Towards explaining agile software development, 2012.

[Fis94] Gerhard Fischer. Domain-oriented design environments. *Automated Software Engineering*, Vol. 1, No. 2, pp. 177–203, 1994.

[Fis05] Gerhard Fischer. Distances and diversity: sources for social creativity. In *Proceedings of the 5th conference on Creativity & cognition*, pp. 128–136. ACM, 2005.

[FJ98] J.S. Fitzgerald and C.B. Jones. Proof in the Validation of a Formal Model of a Tracking System for a Nuclear Plant. In J.C. Bicarregui, editor, *Proof in VDM: Case Studies*, FACIT Series. Springer-Verlag, 1998.

[FLM+05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object–oriented Systems*. Springer, New York, 2005.

[FLS08] John Fitzgerald, Peter Gorm Larsen, and Shin Sahara. VDMTools: Advances in Support for Formal Modeling in VDM. *ACM Sigplan Notices*, Vol. 43, No. 2, pp. 3–11, February 2008.

[FR92] Gerhard Fischer and Brent Reeves. Beyond intelligent interfaces: exploring, analyzing, and creating success models of cooperative problem solving. *Applied Intelligence*, Vol. 1, No. 4, pp. 311–332, 1992.

[GT79] J.A. Goguen and J.J. Tardo. An Introduction to OBJ, a Language for Writing and Testing Software Specifications. *Specifications of Reliable Software*, 1979.

[Gut91] John V. Guttag. The Larch Approach to Specification. In *VDM '91: Formal Software Development Methods*, p. 10. Springer-Verlag, October 1991.

[Hal90] Anthony Hall. Using Z as a Specification Calculus for Object-Oriented Systems. In C.A.R.H̃oare Dines Bjørner and Hans Langmaack, editors, *VDM '90 VDM and Z– Formal Methods in Software Development*, pp. 290–318. VDM Europe, Springer-Verlag, April 1990.

[IKM+97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future - the story of squeak, a practical smalltalk written in itself. *ACM SIGPLAN Notices*, Vol. 32, No. 10, pp. 318–326, 1997.

[IPA13] IPA/SEC. 厳密な仕様記述における形式手法成功事例調査報告書. Technical report, 独立行政法人情報処理推進機構 技術本部ソフトウェア・エンジニアリング・センター, 2013.

[Kay72] Alan C Kay. A personal computer for children of all ages. In *Proceedings of the ACM annual conference-Volume 1*, p. 1. ACM, 1972.

[KG77] Alan Kay and Adele Goldberg. ”personal dynamic media”. *Computer*, Vol. 10, No. 3, pp. 31–41, March 1977.

[KN09] T. Kurita and Y. Nakatsugawa. The Application of VDM to the Development of Firmware for a Smart Card IC Chip. *Intl. Journal of Software and Informatics*, Vol. 3, No. 2-3, pp. 343–355, October 2009.

[LLB10] Peter Gorm Larsen, Kenneth Lausdahl, and Nick Battle. Combinatorial Testing for VDM. In *Proceedings of the 2010 8th IEEE International Conference on Software Engineering and Formal Methods*, SEFM '10, pp. 278–285, Washington, DC, USA, September 2010. IEEE Computer Society. ISBN 978-0-7695-4153-2.

[LLJ+13] Peter Gorm Larsen, Kenneth Lausdahl, Peter Jørgensen, Joey Coleman, Sune Wolff, and Nick Battle. Overture VDM-10 Tool Support: User Guide. Technical Report TR-2010-02, The Overture Initiative, www.overturetool.org, April 2013.

[MS95] John H Maloney and Randall B Smith. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pp. 21–28. ACM, 1995.

[NY13] Kumiyo Nakakoji and Yasuhiro Yamamoto. *Conjectures on how designers interact with representations in the early stages of software design*, pp. 379–398. Chapman and Hall/CRC, 2013.

[OA93] Tomohiro Oda and Keijiro Araki. Specification slicing in formal methods of software development. In *Computer Software and Applications Conference, 1993. COMPSAC 93. Proceedings., Seventeenth Annual International*, pp. 313–319. IEEE, 1993.

[OA13] T. Oda and K. Araki. Overview of VDMPad: An Interactive Tool for Formal Specification with VDM. In *International Conference on Advanced Software Engineering and Information Systems (ICASEIS) 2013*, Nov 2013.

[OAL15] Tomohiro Oda, Keijiro Araki, and Peter Gorm Larsen. VDMPad: a Lightweight IDE for Exploratory VDM-SL Specification. In Nico Plat and Stefania Gnesi, editors, *FormaliSE 2015*, pp. 33–39, Florence, May 2015. In connection with ICSE 2015.

[Ola03] Michael Olan. Unit testing: test early, test often. *Journal of Computing Sciences in Colleges*, Vol. 19, No. 2, pp. 319–328, 2003.

[Rob95] Dave Robertson. Lightweight formal methods. In *Proceedings of the Monterey Workshop on Formal Methods: Software Architectures*, 1995.

[RW73] Horst WJ Rittel and Melvin M Webber. Dilemmas in a general theory of planning. *Policy sciences*, Vol. 4, No. 2, pp. 155–169, 1973.

[Sch84] Donald A Schon. *The reflective practitioner: How professionals think in action*, Vol. 5126. Basic books, 1984.

[Sim96] Herbert A Simon. The science of artificial 3rd ed, 1996.

[SMSB05] Syed M. Suhaib, Deepak A. Mathaikutty, Sandeep K. Shukla, and David Berner. XFM: An Incremental Methodology for Developing Formal Models. *ACM Transactions on Design Automation of Electronic Systems*, Vol. 10, No. 4, pp. 589–609, October 2005.

[Spi92] Mike Spivey. *The Z Notation – A Reference Manual (Second Edition)*. Prentice-Hall International, 1992.

[Tas02] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, Vol. 7007, No. 011, 2002.

[VBLN11] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible object layouts: Enabling lightweight language extensions by intercepting slot access. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pp. 959–972, New York, NY, USA, 2011. ACM.

[Ver09] Marcel Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. PhD thesis, Radboud University Nijmegen, 2009.

[VTSHO12] Nguyen Van Tang, Daisuke Souma, Goro Hatayama, and Hitoshi Ohsaki. Modeling and validating the train fare calculation and adjustment system using vdm++. In *International Conference on Verified Software: Tools, Theories, Experiments*, pp. 163–178. Springer, 2012.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, Vol. 41, No. 4, pp. 1–36, October 2009.

[荒木 08]　荒木啓二郎. フォーマルメソッドの新潮流: Part i: 歴史と概要: 1. フォーマルメソッドの過去・現在・未来-適用の実践に向けて. 情報処理, Vol. 49, No. 5, pp. 493–498, 2008.

発表論文一覧

発表論文一覧

1. Tomohiro Oda and Keijiro Araki, Development of Functional Programming Language Systems Based on Formal Method, Proc. 1992 Int'l Computer Symposium, pp.85–91, 1992.

2. Tomohiro Oda and Keijiro Araki, Specification Slicing in Formal Methods of Software Development, Proceedings of Seventeenth Computer Software and Applications Conference, pp.313–319, 1993.

3. Tomohiro Oda and Keijiro Araki, Application of Slicing Technique to Formal Specifications, Proceedings of Joint Conference on Software Engineering '93, pp.203–210, 1993.

4. Tomohiro Oda, Kumiyo Nakakoji and Yasuhiro Yamamoto, Use-Centric Information Re-presentation for Creative Knowledge Work, Proceedings of 2006 Symposium on Interactive Visual Information Collections and Activity, 2006.

5. Tomohiro Oda, Kenro Aihara and Hitoshi Koshiba, Persuasive Navigation Mechanisms for Consumer Generated Media, Proceedings of 2009 Symposium on Interactive Visual Information Collections and Activity, 2009.

6. 小柴等，相原健郎，小田朋宏, 森純一郎, 星孝哲，松原伸人，武田英明, 記憶の想起と記録のためのライフログ・ブログ連携型支援手法の提案, 情報処理学会論文誌, vol. 51(1), pp. 63–81, 2010

7. 小柴等, 相原健郎, 小田朋宏, 星孝哲, 松原伸人, 森純一郎, 武田英明, 説得性に基づく情報推薦手法の提案:送り手の属性に着目したモデルと検証, 情報処理学会論文誌, vol.51(8), pp. 1452–1468, 2010.

8. Tomohiro Oda, Kumiyo Nakakoji and Yasuhiro Yamamoto, SOMETHINGit: A Prototyping Library for Live and Sound Improvisation, Proceedings of Workshop on Live Programming, 2013.

9. 小田朋宏, 中小路久美代, 山本恭裕, ライブ UI プロトタイピングに向けたマルチ言語環境 SOMETHINGit, ソフトウェアシンポジウム 2013 論文集, 2013.

10. Tomohiro Oda and Keijiro Araki, Overview of VDMPad: An Interactive Tool for Formal Specification with VDM, International Conference on Advanced Software Engineering and Information Systems, 2013.

11. 小田朋宏, 荒木啓二郎, 形式的仕様スケッチのためのウェブベース開発環境 VDMPad, ソフトウェアシンポジウム 2014 論文集, pp.139-146, 2014.

12. Hiroko Satoh, Tomohiro Oda, Kumiyo Nakakoji, Takeaki Uno, Satoru Iwata and Koichi Ohno, ”Maizo”-chemistry Project: toward Molecular- and Reaction Discovery from Quantum Mechanical Global Reaction Route Mappings, Journal of Computer Chemistry, Japan, Vol. 14 (2015) No. 3 Special Issue: Selected Papers from the Annual Spring Meeting 2015 p. 77–79, 2015.

13. Tomohiro Oda, Keijiro Araki and Peter Gorm Larsen, VDMPad: a lightweight IDE for exploratory VDM-SL specification, Proceedings of the Third FME Workshop on Formal Methods in Software Engineering (FormaliSE’15), pp. 33–39, 2015.

14. 小田朋宏, 荒木啓二郎, VDM-SL 実行可能仕様による Web API プロトタイピング環境, ソフトウェアシンポジウム 2015 論文集, pp. 20–25, 2015.

15. Tomohiro Oda, Keijiro Araki and Peter Gorm Larsen, VDM Animation for a Wider Range of Stakeholders, Proceedings of the Thirteenth Overture Workshop, pp. 18–32, 2015.

16. 小田朋宏, 荒木啓二郎, VDM-SL 仕様からの Smalltalk プログラムの自動生成, ソフトウェアシンポジウム 2016 論文集, pp..1–10, 2016.

17. Tomohiro Oda, Keijiro Araki and Peter Gorm Larsen, ViennaTalk and Assertch: Building Lightweight Formal Methods Environments on Pharo 4, In Proceedings of the International Workshop on Smalltalk Technologies, pp.. 4:1–4:7, 2016.

18. Hiroko Satoh, Tomohiro Oda, Kumiyo Nakakoji, Takeaki Uno, Hiroaki Tanaka, Satoru Iwata and Koichi Ohno, Potential Energy Surface-Based Automatic Deduction of Conformational Transition Networks and Its Application on Quantum Mechanical Landscapes of d-Glucose Conformers, Journal of Chemical Theory and Computation, Vol. 12(11), pp. 5293–5308, 2016.

19. Tomohiro Oda, Keijiro Araki and Peter Gorm Larsen, Automated VDM-SL to Smalltalk Code Generators for Exploratory Modeling, In Proceedings of the 14th Overture Workshop: Towards Analytical Tool Chains, pp. 48–62, 2016.

20. Tomohiro Oda, Keijiro Araki and Peter Gorm Larsen, A Formal Modeling Tool for Exploratory Modeling in Software Development, IEICE TRANSACTIONS on Information and Systems, Vol. E100.D(6), pp. 1210–1217, 2017.

21. 小田朋宏, 荒木啓二郎, 形式仕様工程の初期段階に着目した統合仕様記述環境 ViennaTalk, コンピュータソフトウェア（平成２９年１１月 出版予定）.

索 引

functions, → 関数

JSON, 97

Lively Walk-Through, 94

operations, → 操作
Overture tool, 78

Pharo Smalltalk, 53

Smalltalk, 53
state, → 状態

types, → 型

UI プロトタイプ, 94

values, → 定数
VDM-SL, 6
VDMBrowser, 65
VDMJ, 53
VDMPad, 86
VDMTools, 78
ViennaTalk, 4, 53

Web API, 97
Web IDE, 86
Webly Walk-Through, 97

悪定義問題, 3, 9
アニメーションオブジェクト, 55, 65

イメージベース環境, 58

インタプリタ, 7
インタラクション, 54
インタラクションモデル, 88

学際性, 11
型, 24
型検査, 49
関数, 24

機能項目の網羅性, 16
機能定義の網羅性, 16

形式手法, 2
形式手法 7 つの神話, 117
形式仕様, 2, 11
形式仕様記述言語, 6, 15
検証容易性, 15, 16

構文解析, 48
構文検査, 62
顧客が抱えている問題, 10
個人作業での試行錯誤, 19
コミュニケーションを介した試行錯誤, 20

事後条件, 24
事前条件, 24
実現可能性, 11, 16
失敗要因, 41
自動単体テスト, 52
仕様, 1

仕様アニメーション, 7, 49, 55, 88
仕様記述工程, 1, 10
詳細化, 4
状態, 24
状態変数, 24
証明, 5

ステークホルダー, 2, 11

精緻化, 3, 17
静的型検査, 49, 62
正当性, 5
設計指針, 3, 45

操作, 24

ダイナブック, 53
妥当性, 16
段階的詳細化, 4
探索的仕様記述, 3, 17

定数, 24
適法性, 15
テストケース, 52
テストフレームワーク, 52

動的型検査, 49, 62
ドメイン, 10
トランスパイラ, 51, 61, 77

表明検査, 62
品質特性, 13

フィードバック, 12
不変条件, 24
プログラムの自動生成, 51
フロントローディング効果, 2, 11

変更可能性, 15, 16

無矛盾性, 15, 16

問題解決, 1, 9
問題定義, 9

ライブ性, 54, 90
ライブな仕様アニメーション, 56

良定義問題, 3, 9

ワークスペース, 68