SCSK CORPORATION

VDMTools

VDM++開発手法ガイドライン

How to contact SCSK:

http://www.vdmtools.jp/	VDM information web site(in Japanese)
http://www.vdmtools.jp/en/	VDM information web site(in English)
http://www.scsk.jp/	SCSK Corporation web site(in Japanese)
http://www.scsk.jp/index_en.html	SCSK Corporation web site(in English)
vdm.sp@scsk.jp	Mail

VDM++開発手法ガイドライン 2.0

— Revised for VDMTools v9.0.2

目 次

概 要

オブジェクト指向分析におけるグラフィカルモデルの限界については、広く知られていることである。 視覚化においては優れている一方で、正確かつ明白な記述を行うために十分なものではない。

本書では、オブジェクト指向形式仕様言語 VDM++において、UML におけるグラフィカルモデルの利点を形式モデル化や正当性確認への合理的かつ簡易なアプローチに結びつける手法とツールを、**VDMTools** の機能を借りて明らかにしていく。

VDMTools は VDM++ と UML(Rational Rose) の相互変換を提供している。これにより UML のユーザーもまた、不変条件や事前条件および事後条件といった自動的な注釈チェックを用いて、型チェックを行ったり実行可能モデルのテストを行うといった、強化された分析と正当性確認の機能を得ることとなる。

1 はじめに

オブジェクト指向モデルにおいては、グラフィカルな視覚化を用いることで、モデル化された “問題” の複雑さに対処して理解を十分深めるための助けが得られる。グラフィカルな図式を用いると、モデルを抽象化した高水準の概要をつかむことが可能である。これはたとえば、ソフトウェア開発プロセスの早期フェーズでの要求分析に対して、大きな価値となり得る。また続くフェーズにおいても、再度図式が概要理解を助けてくれるため、コード記述の改良が図れるのである。

分析のためのグラフィカルモデルは、ソフトウェア技術者にとってもプログラミング経験のないクライアントに対しても、両者に適していると言える。しかしながら、グラフィカルモデルが実際に意味するものに対して、それと異なる認識を人々が持ってしまうことがある。このような共通理解の欠如が起きる原因はモデルの複雑さにあるとすることはできるが、たとえ小さくてもあるモデルを評価したり検証することは難しいことであるし、さらにグラフィカルモデルに含まれる箱形や矢印がどこまで影響するものなのかを、正確に予見することもまた困難である。

対照的に、実行可能なモデルならばソフトウェア開発過程の早い段階でモデルの有効性が確認されるので、モデル評価の自動化に適している。グラフィカルなフロントエンドをデザインする高速なプロトタイピング能力と結びつけば、実行可能モデルは手戻り作業の頻度ときつさを大きく削減できるため、生成物を予定通りに予算内で納品することも可能となる。例えば、技術者はモデルを実行させることで仕様上に即時にフィードバックを得る。さらにモデルを拡張して、クライアントの問題領域および諸活動と一致させたグラフィカルなフロントエンドとすることで、フロントエンドを介してモデルと直接的な相互作用を行うことができるため、クライアントは諸要求に対する仕様の取り決めを明確に把握できることになる。

本書では、UML のグラフィカルモデル化の利点をオブジェクト指向形式仕様言語 VDM++[1] における実行可能モデルの利点と結びつけることで、オブジェクト指向分析および設計を強めていこう、という手法とツールを提供する。形式仕様においては、抽象概念を保ちつつも精度を高めることで、モデル化プロセスの改善を図ることができる。正当性確認プロセスのどの部分においても、精度なしでの自

[1] VDM++ は ISO 標準 VDM-SL のオブジェクト指向拡張である。標準 VDM-SL は構造化機構を提供しないが、VDM-SL ではモジュール群のサポートを行う。 VDM++ は UML に似たクラスや他の概念のサポートを行う.

動化は不可能であろう。またそれらプロセスの、詳細で複雑な実装にではなくむしろ概念上の性質に焦点をあてるためには、抽象化が不可欠なことである。明白な構造化された方法で要求を述べること、加えて、技術者に親しみやすいツールやテクニック、つまりテストやラピッド・プロトタイピングを用いることで、精度の高いモデルなら自動的に分析され正当性確認がされる。SCSK ではこのアプローチを “Validated Design through Modelling モデル化を通して正当性確認される設計” (VDM) と呼び、 **VDMTools** がこれを裏付けしている。この文書は、読者が既に一般的なオブジェクト指向ソフトウェア開発にいくらかでも慣れ親しんでいるもの、と仮定している [Mey88, SM88, RBP+91]。

1.1 VDM++ と UML

大雑把に述べて、同じ方法でソフトウェアの開発を行う会社は２つとない。たとえ１つの会社内であっても、異なるプロジェクトや人々の間でのソフトウェアプロセスは様々に変化する可能性がある。オブジェクト指向ならば VDM++、構造化ならば VDM-SL、その他特殊な場合は VDM++または VDM-SL のどちらかを使い、原則としてどのようなソフトウェアプロセスにもこれを改良するアドオンとして VDM を用いることができる。しかしながら本書では、UML と VDM++がどのように一緒に用いられるかに焦点をあてる。さらに、要求分析とかシステム設計といったソフトウェア開発の典型的なフェーズのいずれかを区別することなく、どのようにモデルを構築するかに焦点をあてていくこととする。これはフェーズ間の主な違いというものが、抽象化のレベルにあるからである。モデルを構築するために基本的なアプローチを行うことと、 **VDMTools** を用いてこれらのモデルを分析することが、同じになる。

ユーザーに UML と VDM++の相補的な利点を享受してもらうため、**VDMTools** のツールセットである VDM++ Toolbox は Rose-VDM++ Link [SCSa] を提供している。これにより、UML モデルと VDM++モデルの間に簡単な移行が用意された。UML クラス図は自動的に VDM++の仕様に変換され、逆もまた同様に行えるので、完全な双方向 (ラウンドトリップ) エンジニアリング能力を備えたものとなっている。ユースケースのような UML の他の局面も、独立に VDM++モデルのものとして用いることができる。典型的には、ユースケースは VDM 開発に先行することになるであろう。大雑把に述べて VDM++モデルが実質的な原文であり、そこでしばしば書き落とされたり自然言語であいまいに述べられていた詳細すべては、UML クラス図の精密版が取り入れる。それゆえに UML モデル と VDM++

モデルは相補的なものであって、実際は、同一モデルの２つの見方として捉えることができる。

UML の最近の版には OCL(Object Constraint Language) と呼ばれる表記法があるが、これは VDM++と多くの類似点、たとえば事前条件や事後条件、をもつ。しかしながら VDM++では、 **VDMTools** を用いてテストを行うという原則に立った正当性確認が可能なので、OCL より好ましいといえる。OCL では実行可能な部品を含まないので、これは不可能である。

本書は、UML、VDM++、あるいは VDM++ Toolbox を完全に紹介しているものではない。更なる情報を求める場合には、 [GBR99, SCSb, SCSc] を参考にすることができる。

1.2 モデルの構築

モデル構築の方法は、概念の選択のみによるものではない。モデルの目的は何かが同様に重要不可欠であり、これは分析の種類により決定される。例えば車を設計する場合、あるモデルの目的は空気力学をテストすることであり、もう１つのモデルの目的はブレーキシステムをテストすることであり、第３のモデルの目的は燃料噴射システムであり、といった具合である。これらのモデルにおいて、各々を他のモデルと別に抽象化することができる。たとえば、ブレーキと燃料噴射のシステムモデルでは各々別々の分析が可能だ。どの詳細を表現し、また分析に関係ないため無視するのかは、抽象化で決定する。モデル構築上でシステムの複雑さを処理するツールが、抽象化である。

モデルの目的を設定した後は、以下の手順リストがモデル構築を助けるガイドとなるであろう：

1. 要求項目を読み取る。
2. 要求項目より機能行動を分析する。
3. 可能性のあるクラスまたはデータ型 (通常は名詞) および操作 (通常は動詞) のリストを抽出する。リスト中の項目に説明を与えることで、用語集の作成を行う。

4. UML を用いてクラス表示を作図する。これは属性やクラス間の関連を含むものとなる。VDM++モデルとしての内部的一貫性をチェックする。
5. 操作に対するシグニチャを作図する。VDM++モデルとしての一貫性をチェックする。
6. 要求項目から潜在的な不変条件を決定しそれらを形式化することで、クラス (およびデータ型) の定義を完成する。
7. 事前条件や事後条件そして操作本体を決定することで、必要があれば型定義を修正しながら、操作定義を完成する。
8. テストとラピッド・プロトタイピングを用いて、仕様の正当性確認を行う。
9. 自動コード生成または手動でのコード作成により、モデルを実装する。

ステップ２は UML から伝統的ユースケースを用いて対処できるであろうし、あるいは構造化から外した少しの VDM 定義を行うことで対処できる。VDM++に初めて挑戦しようとする大部分の読者は、たぶんユースケースの方を用いることと思われるが、もう一方のアプローチについても [Gro00] に記してある。大まかに述べれば、上記のステップ１から５は、従来通りのオブジェクト指向 UML 設計にて実装されている部分である。唯一の違いはステップ４と５において、VDM++を結合することでさらに多くの矛盾を見つけることができるというところにある。そうとはいえ後で述べるが残りのステップ (６から８) では、システムの統合性を確実のものとするために対処されるべき、大変微妙かつ重要な設計考察の確認を行うことができるのである。

以下の章では、化学プラント向けの簡単なアラームおよび専門家スケジュールシステムに対する要求項目を提示する。これは本書の残り部分においてモデル化される。モデル化プロセスは、本書の中では１つのプロセスとして記述されているが、実際の開発においては通常繰り返されるものとなるであろう。さらに、本書で説明目的に用いる例は大変小さいものであるため、上記リストのステップ２は省いて進めるものとする。

2　化学プラントの例

この章では、化学プラントの簡単なアラームシステムに対する要求項目を提示する。この例は大規模なアラームシステムの部分構成を基に考えられたもので、

Fitzgerald と Larsen [FL98] により VDM-SL の本の中で取り上げられている。Rose-VDM++ Link を含めた VDM++ Toolbox の機能を用いて、このシステムモデルを発展させ正当性確認を行っていく。

化学プラントでは、プラント内の状況に反応してアラームを鳴らせるように、数多くのセンサーが装備されている。アラームが鳴れば、専門家がその場に呼ばれることが必要となる。専門家は様々なアラームに対処するための様々な資格をもつ。様々な各々の要求項目は、参照を行うために R1-R8 とラベル付けしてある。

R1 このプラントのアラームを管理するために、コンピューターベースのシステムが開発されるべきである。

R2 アラームに対処するために 4 種類の資格が必要とされている。それらは、電気系資格、機械系資格、生物系資格、化学系資格、である。

R3 システムが稼動している期間では常に、専門家が勤務していなければならない。

R4 各々の専門家が複数の資格を保有することができる。

R5 システムに報告を行う各々のアラームには、それに関連する資格が 1 つあり、そのアラームの説明はその専門家であれば理解できるものである。

R6 システムによってアラームが検知された場合は常に、正しい資格を持った専門家を見つけ呼ぶ必要がある。

R7 専門家は、自分の任務がいつであるかチェックできるシステムデータベースが利用できるべきである。

R8 任務に就く専門家数の査定が可能でなければならない。

次の章で、これらの要求項目と合致するアラームシステムモデルの開発を始めることとする。

3 UML におけるグラフィカルモデル

さていよいよ、上記アラームシステムの最初のモデルを開発していこう。モデルの目的は、アラームを取り扱うために、任務名簿を管理し専門家の呼出しを行

うルールを明確にすることである。UML で 1 つモデルを構築し、その一貫性のチェックに **VDMTools** を使用する。UML/VDM++組み合わせに特有の手法ガイドラインを強調したい箇所では、短文でのハイライトを行うこととする。

3.1 用語集の作成

モデル構築のガイドラインに従って、まず最初に可能性のあるクラスやデータ型のリストを抽出する：

潜在的なクラスとデータ型 (名詞)

- アラーム: 必要資格と警報文
- プラント
- 資格 (電気系、機械系、生物系、化学系)
- 専門家: 保持資格のリスト
- 時間帯
- システムおよびシステムデータベース？おそらくはスケジュールの類

用語集のこの部分は、Rational Rose といった UML ツールを用いてしばしばクラス図 (ダイアグラム) 中でモデル化される。このガイドでは次のステップでなされる。しかし設計されるべきシステムが大きい場合は、ここに示すようにディレクトリ中のすべての潜在的なクラスと型をシステム立ててリストしておく方が、より安全であるかもしれない。

潜在的な操作 (動詞)

- 専門家を呼出す: アラームが与えられている (必要: アラームオペレーターとシステム)
- 専門家が任務につく: 任務時間をチェックする (必要: 専門家とシステム)
- 任務にある専門家の数: 時間帯が与えられている (必要: オペレーター？とシステム)

UML ツールを用いることで、用語集のこの部分はしばしばユースケースと同様に記述される。

この簡単なシステムに対して、もっと適した説明もできるであろうが、上記のリストをシステムにおける主概念の用語集として活用していく。

3.2 クラス表現の作図

用語集を作成してしまうと、次はクラスと型の違いは何かという説明が必要となる。純粋な OO アプローチにおいての違いはないのだが、しばしばとても大きなクラス集合を扱うと、クラス数が多いことの複雑さに対処することが困難となる可能性がある、というのが結論である。VDM++ ではクラスと型の間には 1 つの違いがあるので、我々のモデルではクラス数削減のためにこれを用いていきたい。一般的に述べて型は、よくある OO アプローチにおいてコンテナクラスとして知られるものに相当する。これらは、このクラスを利用するユーザーが読み書きできる属性を、単にたくさんもっているクラスである。

Guideline 1: 用語集にある名詞は、そのモデルの目的に関して “実質的” 機能 (読み／書きに加えるもの) をまったく持たないのならば、型としてモデル化されるべきである。

上記の用語集にある資格は、型としての方がよりよくモデル化される名詞の 1 例である。資格は列挙型であるが、ちょうど 4 つの可能な値をもち明白な機能を持たないからである。もし我々のモデルの目的が、別々の専門家たちが彼らの資格に基づいて実行できる別々の種類の行為を含むとしたなら、資格は 1 つのクラスとする方がより自然であったろう。

ここで用語集からアラームと専門家について考察してみよう。両者とも、例題における物理的世界で現実的な対象物のグループに相当するもので、共存ししかも独立して行動する。したがってこれら両方を、クラスとしてモデル化していこう:

アラーム は属性として 資格 と 警報文 をもつクラスである。警報文は文字列であり、資格 (図 1 参照) と同様に 1 つの型と考えることができる。

専門家 は属性として 資格 をもつクラスである。要求項目では、資格リストがあることと指定している。単語リストは暗黙にある種の順番を示すものであるが、要求からはこの順番をどのようにすべきかは何もわからない。この

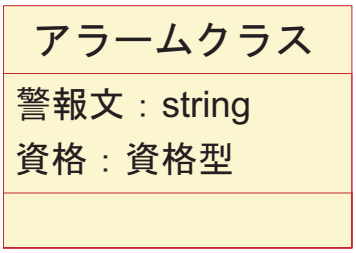

図 1: アラームクラス

ことは我々の “顧客” と共に明らかにする必要があるが、しばらくの間は求められる機能に対して重要性を持つような具体的な順番付けはないと仮定することとする。図 2 に示されている。

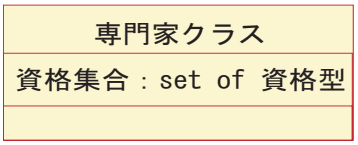

図 2: 専門家クラス

ともかくも、アラームと専門家のスケジュールを関連付ける何らかの結合を行う必要がある。このような結合に対して、 UML/VDM++ 手法において 厳密な限界を表すことを可能にするために、他のクラスへの関連を含めた main クラスの作成を典型として推奨している。

Guideline 2: 異なるクラスとそれらの関連間の厳密な限界がここに表せるような、全システムを象徴する “main” クラスを作成する。

我々の例題中では、 main クラスは **プラント** である。我々のモデルの目的は、任務スケジュールおよびアラームを処理する専門家の呼出しに関するルールを明確にすること、であることを思い起こそう。我々はこの目的に焦点をあて、抽象的な展望をモデル化したクラスとしてプラントを選んだ。それゆえに、“プラント” の 2 局面: 任務にある専門家の スケジュール および (登録されている) 可能な限りの アラーム集合 の取り集め、が重要である。まずはアラームについて考えてみよう。プラントクラスからアラームクラスへの関連が必要であるが、システム内のアラームは複数であり得るため、この関連には多重度を用いる必要がある。以後この関連を アラーム集合 と呼ぶ (下の図を参照)。

Guideline 3: １つの関連が導入された場合は常に、その多重度についての考察が必要で、その関連の利用を望む方向での役割名を与えるべきである。

次に、様々な時間帯に配置される多くの専門家を包括するスケジュールについて考えてみよう。各々の時間帯で 0 人あるいは何人かの専門家が任務につく可能性がある。このように、時間帯と共に制限が与えられゼロまたはそれ以上の多重度をもつ関連を、我々は用いる必要がある。この関連を スケジュール と呼ぶ。

Guideline 4: ある関連が何かの値にしたがうのならば、その関連に対し制限を与えるものが導入されるべきである。制限を与えるものの名称は VDM++ 型でなければならない。

要求項目における時間帯についてはあまり説明されていないため、モデルをこの焦点から多く知る必要はない。時間帯は“実質的”機能は含まないため、型としてモデル化することができる。

図 3 に示すように、現時点でシステムクラス図は３つのクラスをもつものとして描くことができる。

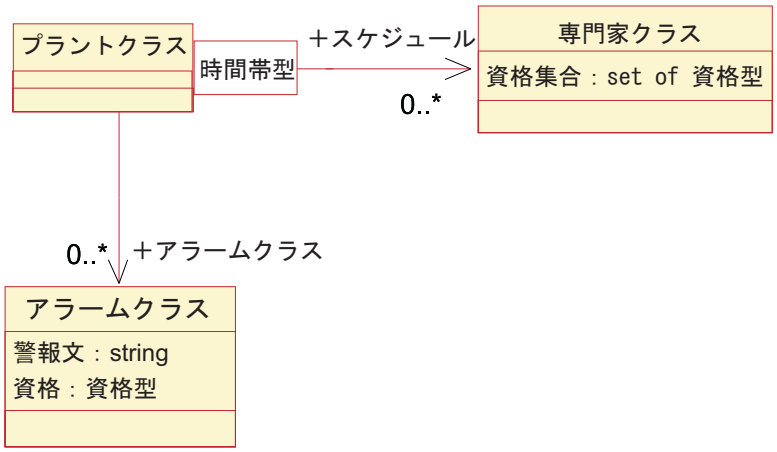

図 3: 初期クラス図

ここで我々は、属性を加えクラス間に関係をもたせるために、 Rose を用いた。Rose で提供されるテンプレートに従って、属性と型を入力した。型は VDM++型であり、たとえば、集合型構成子 “set of” やクラス名称や他の型識別子といったようなものである。このすべてを VDM++において直接に、同じように具合よく働かせることができたので、開発者たちは場合によっては、Rose-VDM++ Link を用いて最初に原文モデルを書きそれからグラフィカルな視覚化を図った方が速い

と気づく。

正しいモデルというものは存在しない。たとえば我々が、このアラームシステムの使用に関連するビジネスプロセスを明示したいと考えたならば、もっと大きなコンピューターシステムの一部としてであったろうし、そしてあそらくはモデルも異なったものとなっていたであろう。それよりむしろ、どのようにスケジュールが構築されそして保守されるかに焦点をあてたかったのである。

次には Rose-VDM++ Link を用いて、UML モデルから VDM++スケルトンへ自動的にマッピングする。これらは以下に含まれる。

プラントクラス

```
--
-- THIS FILE IS AUTOMATICALLY GENERATED!!
--
-- Generated at Mon 09-Aug-99 by the Rose VDM++ Link
--
class プラントクラス

instance variables
 アラーム集合 : set of アラームクラス;
 スケジュール : map 時間帯型 to set of 専門家クラス;

end プラントクラス
```

スケジュール については、これは UML では制限が与えられた関連であるが、VDM++においては写像に変換されていることに注目したい。１つの写像はテーブルと似ていて、(定義域において) キーを用いた検索を行い (値域において) それに関連付けられた値を得ることができる。加えて、ここでのキーは時間帯型であり、値は専門家クラスの集合である。さらに VDM++の集合型は、**アラーム集合** インスタンス変数におけるのと同様、順をつけない“０以上”の関連を表すのに用いられることに注目したい。

専門家クラス

```
--
-- THIS FILE IS AUTOMATICALLY GENERATED!!
--
-- Generated at Mon 09-Aug-99 by the Rose VDM++ Link
--
class 専門家クラス

instance variables
 資格集合 : set of 資格型;

end 専門家クラス
```

繰り返すが、 VDM++集合型は **資格集合** インスタンス変数に対する “0 以上” の多重度との UML 関連を表すために用いられることに注目したい。

アラームクラス

```
--
-- THIS FILE IS AUTOMATICALLY GENERATED!!
--
-- Generated at Mon 09-Aug-99 by the Rose VDM++ Link
--
class アラームクラス

instance variables
 警報文 : string;
 資格 : 資格型;

end アラームクラス
```

UML クラスの属性は、UML の関連と同様に、VDM++ クラスのインスタンス変数になることに注目したい。

クラスの型チェック

Guideline 5: スケルトンクラスが完成したらすぐ(何らかの機能が導入される前に)、内部的一貫性のチェックのために **VDMTools** を用いること。

VDMTools を用いることで、我々は内部的一貫性の確認のためにモデルの型チェック、つまりすべての識別子がちゃんと定義されているか、を試みることができる。上記の３つのクラスは、３つの定義されていない識別子: **時間帯型**、**資格型** そして *string* (VDM++のビルトインではない) があるために型が正しいとされない。したがって、以下の３つの型定義が３つの VDM++クラスのそれぞれに挿入される。

```
class プラントクラス

types
 public 時間帯型 = token;

instance variables
...

end プラントクラス
```

token 型 は、明確でない値の無限集合を含み、唯一の演算子として等号がある。時間帯型に対するどのような表現が最終的に実装されるかは現時点では問題にしないため、これを用いる。ここから抽象化が始まる。型 **時間帯型** は `public` に宣言されている必要があるが、 **プラントクラス** の外で利用できなくては困るからである。

```
class 専門家クラス

types
 public 資格型 = <機械> | <化学> | <生物> | <電気>;

instance variables
...
end 専門家クラス
```

資格型 は列挙型で、VDM++においては引用型の和集合 (垂直バーで表す) として定義されている。引用型は、それがもつ単一の値と同じ名称をもつが、これは “<” と “>” 記号の中に書かれていなければならない。

```
class アラームクラス

types
 public string = seq of char;

instance variables
 警報文 : string;
 資格 : 専門家クラス`資格型;

end アラームクラス
```

資格 はここで定義されるので、インスタンス変数 **資格型** の前にクラス名称 **専門家クラス** が置かれていることに注目しよう。Rose-VDM++ Link を用いることで、この変化は図 4 に示すように Rose UML level において自動的に更新されている。

UML モデルにおいては、型定義はカウンター部をもたないため、変換されない。しかしそれらは VDM++モデルファイル内に残され、UML モデル内の更新によって削除 (または変更) されることはない。

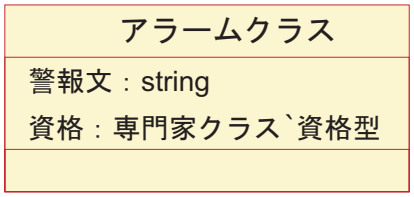

図 4: 更新されたアラームクラス

3.3 操作に対するシグネチャの作図

我々はここで UML の操作シグネチャを追加することにより、開発を続ける。上記の用語集にリストされている 3 つの操作は、それらすべて割り当てられたスケジュールに従うものであるため、そのままプラントクラスに属することとなる。更新されたプラントクラスのクラス図が、図 5 に示されている。

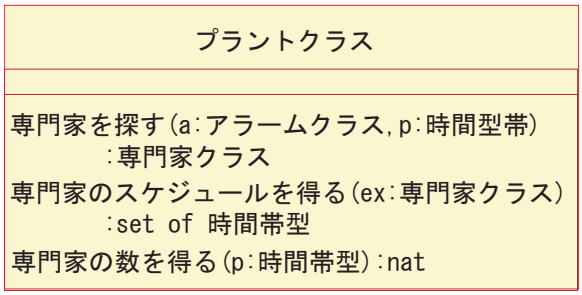

図 5: 更新されたプラントクラス

操作のシグネチャは前方直進型である。たとえば、 専門家を探す 操作は入力としてアラームクラスと時間帯型をとり、結果として専門家クラスを返す。

Guideline 6: パラメーター型と結果型については注意深く考えること。これにより、クラス図の中に欠けていた結合を見つけ出せることが多い。

更新されたスケルトンの VDM++ クラスは、自動的に生成される。

```
class プラントクラス

types
 public 時間帯型 = token;

instance variables
 アラーム集合 : set of アラームクラス;
 スケジュール : map 時間帯型 to set of 専門家クラス;

operations
 専門家を探す : アラームクラス * 時間帯型 ==> 専門家クラス
 専門家を探す(a, p) ==
   is not yet specified;

 専門家のスケジュールを得る : 専門家クラス ==> set of 時間帯型
 専門家のスケジュールを得る(ex) ==
   is not yet specified;

 専門家の数を得る : 時間帯型 ==> nat
 専門家の数を得る(p) ==
   is not yet specified;

end プラントクラス
```

上記のように開発したモデルを構文チェックや型チェックすることは可能だが、様々な正当性確認目的に用いるには精確さに欠けるものである。我々の次のステップとしては、VDM++を用いてこのモデルにもっと厳密さを与えることだ。従来よりほとんどのUML開発者は、この段階でストップしてしまっていた。以下に挙げていくが、最も重要な分析と設計における展開はまだここからなのである。

4　モデルをより厳密にする

この段階で、我々のモデルが個々の要求項目を十分に満たすものか、見直すというのもよい考えである。明らかに、まだ操作の詳細にわたる考察がなされていないので、**R6—R8** は完全には網羅できていない。それ以外は、要求項目 **R3** をいずれでも文書化しなかった場合を除き、要求項目は納得できる程度にはうまく網

羅されていたように思える。:

R3 システムに割り当てられたすべての時間帯で、任務についた専門家がいなければならない。

しかしながら、上記のグラフィック化され型チェックされたVDM++モデルには、以下に述べるが、いくつか更に隠れた仮定と明瞭とはいえない面がある。

モデルを実装する人は典型的に、始めの要求項目から直接行わず、また行うべきでもなく、いくつかコメントを伴った分析や設計のモデルを用いる。それがモデルの１つの目的ともなっている。したがってさらに厳密なる抽象化を行えば行うほど、実装を選択するしないにかかわらず、誤った実装を行うというリスクを低減できるのである。

4.1 不変条件プロパティの追加

要求項目 **R3** は、VDM++モデルにおいて厳密な方法で形式化することができる。第一に考えることは、システムにおいて時間帯が割り当てられるとは何を意味するのか？ということ。これは専門家にとってはスケジュールの中でいつ任務に就くかということであり、つまりインスタンス変数スケジュール定義域において、時間帯型から専門家クラス集合への１つの写像なのである。

```
forall p in set dom スケジュール &
  時間帯 p で任務についている専門家がいる
```

次に考えることは、１つの時間帯に対し専門家が任務についているとは何を意味するのか？ということ。時間帯型と関連する値域の値が、専門家クラスの集合と呼ばれていて、空でないことを意味する。

```
forall p in set dom スケジュール & スケジュール(p) <> {};
```

VDM++においては、この述語は１つの不変条件(inv と略される) として **プラントクラス**のインスタンス変数の部分に加えられている：

```
class プラントクラス
...

instance variables
 アラーム集合 : set of アラームクラス;
 スケジュール : map 時間帯型 to set of 専門家クラス;
 inv
   forall p in set dom スケジュール & スケジュール(p) <> {};

...
end プラントクラス
```

不変条件とは、常にオブジェクトの状態を保持しなければならないという条件である。

Guideline 7: 重要なプロパティあるいは不変条件としての制限は、文書化を試みるべきである。なぜならシステムのある部分で、他の部分では当然気づくであろう何らかの不変条件プロパティの保持を、暗黙に仮定してしまっているかもしれないからである。

4.2 操作定義の完成

次に我々は、操作と関連する条件である事前条件と事後条件のそれぞれについて考える。事前条件は操作が行う仮定、つまり 1 つの操作が実行される前にパラメーターやオブジェクト状態の何が保たれなくてはならないのか、の形式化を行う。事後条件では操作が何をもたらすか、つまり実行が行われた後に何が保持されるか、を述べる。このように事後条件とは、一方の初期設定のオブジェクト状態とパラメーター値、他方の最終状態と結果値、との間の関係である。たとえば、操作 **専門家を探す** はアラームクラスと時間帯型を入力として取り込み、アラームクラスを扱う専門家クラスを返す。では、(事前条件は) 任意のアラームクラスや時間帯型を受け入れるのだろうか？また、(事後条件は) 専門家クラスの何を保持しなければならないのだろうか？

専門家を探す は以下のシグネチャをもつ:

```
専門家を探す : アラームクラス * 時間帯型 ==> 専門家クラス
```

しかし上記で示されているように、これはどんなアラームクラスも時間帯型も扱えるわけではない。第一に、任務に就いている専門家がいることを保証するために、時間帯型はスケジュールの中で考えられる必要がある。アラームはシステム内で登録済みである、つまりインスタンス変数アラーム集合に含まれる限られたものの中にある、という必要性も明らかであろう:

```
専門家を探す : アラームクラス * 時間帯型 ==> 専門家クラス
専門家を探す(a, p) ==
  is not yet specified
pre a in set アラーム集合 and
    p in set dom スケジュール
```

さらに、この操作では任務についてしまっている専門家を返すべきではない。専門家はアラームを扱うための正しい資格を持つべきである。このことが事後条件において文書化されている。

```
post let ex = RESULT in
       ex in set スケジュール(p) and
       a.要求される資格を得る() in set ex.資格を得る();
```

let 式は単純に操作の結果を 1 つの識別子と結びつける。最初の連結がこの結果である専門家は任務にあることを述べ、2 番目が資格要求に行き着く。ここで我々は、アクセス操作 要求される資格を得る と 資格を得る が各々アラームクラスと専門家クラスに対して定義されていると仮定する。情報交換を行うために、この種のアクセス操作は OO 開発中しばしば定義される必要があるが、定義と言ってもごく簡単である。

この段階で操作の本体を指定する必要はないが、後に操作を実行可能とするために導入されるのである。

ここで与えられた定義では、もし正しい資格をもった複数の専門家が指定の時間帯で任務についている場合に、どの専門家が選ばれるべきかについては述べていない。要求項目からもっとも望まれる選択は何か知る方法は得られない、だからこそこの暗黙の定義は、後の開発プロセスにおいてこの問題を決定する自由を残してくれているのである。

Guideline 8: いくつかの選択が可能である場合には、求められている機能を記述する暗黙の方法の利用を試みるべきである。

ここで、このように正しい資格を持つ専門家を返すことは常に可能なのだろうかと、自問できる。登録されたアラームに関連し任務につく専門家のスケジュールに制限はない、とする現在のモデルにおいて、確実に返せるとは言えない。ただし不変条件プロパティを加えることで、この信頼性を導ける。:

```
instance variables
 アラーム集合 : set of アラームクラス;
 スケジュール : map 時間帯型 to set of 専門家クラス;
 inv forall p in set dom スケジュール & スケジュール(p) <> {};
 inv forall a in set アラーム集合 &
        forall p in set dom スケジュール &
           exists ex in set スケジュール(p) &
              a.要求される資格を得る() in set ex.資格を得る();
```

新しい不変条件では、システムに登録されたすべてのアラームと時間帯について、そのアラームを操作する正しい資格をもつ専門家がその時間帯内で任務に就いていなければならない、と述べる。これは、 専門家を探す の記述が意味を持ち実行可能であること、つまり、求められる専門家は常に見つかるということ、を保証する。VDM++モデルの厳密さのため、システムに求める機能に関連して、我々はこのような詳細にわたり自問自答してきたということに注目して欲しい。たとえばもしシステムを拡張し、専門家が自ら交替勤務を変更できるよう支援を行うとしたら、上記の不変条件がまた重要となる。不変条件は、システムに対して支配的な設計要因 (パラメーター) となるプロパティを識別するが、それは通常の OO アプローチでは識別されないのである。

Guideline 9: いくつかの操作が記述されている場合には、不変条件プロパティを追加してプロパティの識別を試みるべきである。

専門家の数を得る は以下のシグネチャをもつ:

```
専門家の数を得る : 時間帯型 ==> nat
```

これは、与えられた時間帯で任務にある専門家の数を返すものでなければならない。おそらく我々は時間帯として、スケジュール内の既知のものを求めている。これは事前条件である。結果としては、正確にその時間帯で任務に就いている専門家集合の序数となるべきで、これが操作の本体となる:

```
専門家の数を得る : 時間帯型 ==> nat
専門家の数を得る(p) ==
  return card スケジュール(p)
pre p in set dom スケジュール;
```

この場合は事後条件は必要でない。というのはそれは本体をただコピーしただけになってしまうから、公正で高水準な方法で既に記述されているからである。

専門家のスケジュールを得る は以下のシグネチャを持つ:

```
専門家のスケジュールを得る : 専門家 ==> set of 時間帯型
```

専門家には、いつ任務に就いているのか訊ねることができなくてはならないため、この操作に対して事前条件を指定する必要はない。さらに操作の本体が高水準な方法で自然に記述できるので、事後条件も必ずしも必要でない:

```
専門家のスケジュールを得る : 専門家 ==> set of 時間帯型
専門家のスケジュールを得る(ex) ==
  return {p | p in set dom スケジュール &
              ex in set スケジュール(p)};
```

この本体は、与えられた専門家が複数の時間帯で任務に就いている場合のように、スケジュール定義域内の時間帯型の集合を返すものである。厳密で、明瞭で、し

かも抽象化されている! このような文ならば、プログラム言語中のコード何行かで済むはずである。Java ではこれが次のようなものとなるであろう:

```
import java.util.*;

class AlarmClass {

  Map Schedule;

    Set getExpertSchedule(Integer ex) {
      TreeSet resset = new TreeSet();
      Set keys = Schedule.keySet();
      Iterator iterator = keys.iterator();

      while(iterator.hasNext()) {
        Object p = iterator.next();
        if ( ( (Set) Schedule.get(p)).contains(ex))
            resset.add(p);
      }

      return resset;
   }
}
```

Guideline 10: VDM++における明示的な操作定義は、プログラム言語で書かれたコードと比較して、厳密で明瞭でしかも抽象化したものとなるよう心がけること。

まとめとして、Rose-VDM++ Link を用いて、(型チェックされ一貫性のある)VDM++ モデルにおける変更を UML モデルへマッピングしよう。結果は次の図の通り:

これはモデルに良い概観を与えているが、要求項目についての詳細な情報はもたない。VDM++ モデルは、システム実行に対する本質的な情報をもつ。このように２つのモデルは、あるいは同一モデルの２つのビューは、相補的である。

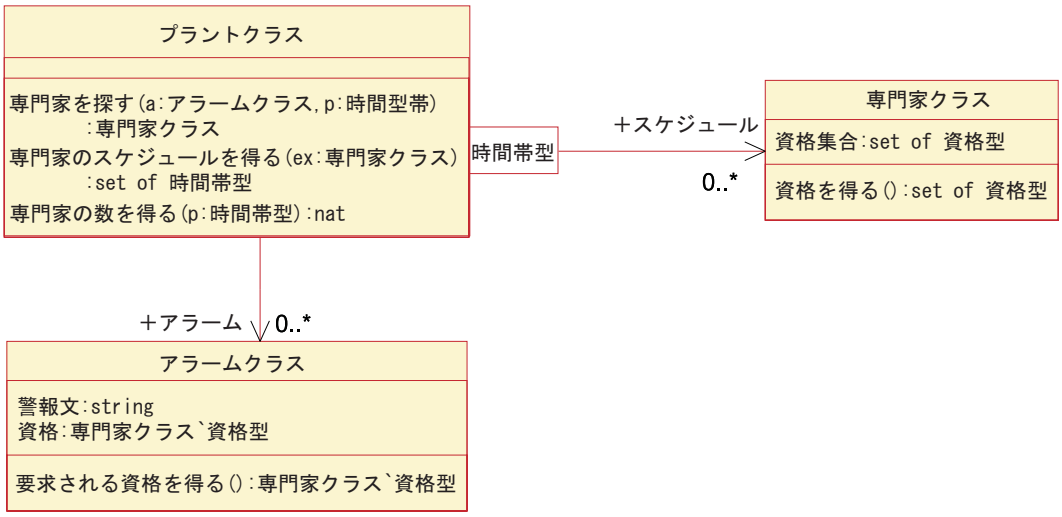

図 6: 更新された UML クラス図

5 VDM++の正当性確認

前の章で、１つのモデル構成の第一ステップを体験した。**VDMTools** の型チェック機能が、モデルの内部的一貫性をチェックするためにどのように用いられるのかを見てきた。このような機能は役に立つが、すべてのエラーを見つけることができるわけではない。加えて、顧客が求めたものをモデルが記述していること、また顧客が考えたことを明確にすることが求められたことへの信頼を得るために、正当性確認が用いられる。

本来モデルを検証するには、システムテストとラピッドプロトタイピングのそれぞれをベースとした 2 つのアプローチがある。したがって、**VDMTools** は他のコード (例えばグラフィカルなフロントエンド) と共に、従来のテスト技術を利用することで、モデルの検証を実現する支援をする。それ故、VDMTools のインタープリタでは、それらが実装される前に象徴的に仕様を実行することができる。実行中は自動的に注釈、つまり不変条件、事前条件、事後条件、のチェックを行う。何らかの条件が成立しない場合には、ユーザーには違反された条件とどこでその違反が起きたかという具体的な情報が通知される。条件が強ければ強いほど、またテスト範囲が広ければ広いほど、正当性確認の結果としてのモデルの信頼性は高まる。

5.1 システムテストを用いて自動化された正当性確認

モデルは伝統的方法でテストされる、したがってこの文書ではテスト理論の詳細に踏み込まない。しかし、インタープリタの動作に２つのモードが備わっていることには、触れておく価値がある。モデルの相互作用的なテストが可能で、その場合テスト式がインタープリタに手渡しされすぐに評価がなされる。他にバッチモードがあり、この場合は潜在的に大きなテストスイート上でインタープリタが自動的に実行される。このモードでは、実行において統計出力を行うのと同時にモデルの実行されない部分を色分けし、テストカバレッジの提示を行うことができる。

テストを遂行するためには、しばしば、上記に用いている 要求資格をセットするや 警報文をセットするといった操作のような、異なるインスタンス変数の値を設定する操作を導入する必要が生じる。さらに、上記で定義された 専門家を探すといった操作は、この現在の形式の中での実行は不可能である。これは我々が陰操作と呼ぶもので、操作本体を持たないからである。しかしながら、簡単に事後条件を本体に組み入れて操作を実行可能にできる。:

```
public
専門家を探す : アラームクラス * 時間帯型 ==> 専門家クラス
専門家を探す(a, p) ==
  let ex in set スケジュール(p) be st
      a.要求される資格を得る() in set ex.資格を得る()
  in return ex
pre a in set アラーム集合 and
    p in set dom スケジュール
post let ex = RESULT in
       ex in set スケジュール(p) and
       a.要求される資格を得る() in set ex.資格を得る();
```

これは let-be-such-that 式と呼ばれる高水準の VDM++ 式を用いて、“be st” に続く述語が保持している任意の専門家を選択する。

インタープリタは次のようなコマンドを実行することができる：

```
create a1:= new アラームクラス()
print a1.要求資格をセットする(<機械>)
```

```
print a1.警報文をセットする("機械系故障")
```

コマンド create はあるクラスの１つのインスタンスをつくるために用いられる。コマンド print は、たとえば上記のような手法の起動として、式は評価される。各々の行はインタープリタに手動で打ち込むことができるし、あるいはこれらのコマンドを含むスクリプトファイルにすべてのテストシナリオをセットアップすることもできる。スクリプトファイルはコマンドスクリプトを用いて実行される。スクリプトファイルは他のスクリプトファイルも同様に呼び込むことができる。以下のファイルは test1 と呼ばれ、インタープリタのウインドウの中で script test1 とタイプすることによって実行される。

```
init
script testing/alarm1
script testing/expert1
create plant:= new プラントクラス()
print plant.スケジュールをセットする({plant.p1 |-> {ex1}})
print plant.アラームをセットする({a1})
print plant.専門家のスケジュールを得る(ex1)
print plant.専門家の数を得る(plant.p1)
print plant.専門家を探す(a1,plant.p1)
```

これは２つの他のスクリプトを呼んでいて、ファイル alarm1 は新しいアラームオブジェクト a1 を定義する:

```
create a1:= new アラーム()
print a1.要求資格をセットする(<機械>)
print a1.警報文をセットする("機械系故障")
```

そしてファイル expert1 は新しい専門家クラスオブジェクト ex1 を定義する:

```
create ex1:= new 専門家クラス()
print ex1.資格をセットする({<機械>,<化学>})
```

図 7: インタープリタの利用

テストスクリプトはスケジュールとアラームの集合を更新する。同時に、不変条件と他のプロパティを自動的にチェックする。

識別子 `p1` は token 定数値で、プラントクラスで定義されている。テスト中に用いられたクラスのバージョンを、付録 A でみることができる。上記の図は、図 7 で表示されたものと同じで、上記 `test1` スクリプトを実行させた後の Toolbox インタープリタのスクリーンダンプを見せている。

以下のテストスクリプトでは上記 test1 を実行し、次に電気系資格を必要とする新しいアラーム a2 を追加しようとしている。

```
script test1
script alarm2
print plant.アラームをセットする({a1,a2})
```

ここに alarm2 スクリプトは以下のコマンドを含む:

```
create a2:= new アラームクラス()
print a2.要求資格をセットする(<電気>)
print a2.警報文をセットする("電気系故障")
```

この専門家は電気系資格を持たないことに注意しよう。そのため 図 8 に見るスクリーンダンプが示すように、これが不変条件違反で中断する。

インタープリタが実行時エラーの具体的な位置情報を提供すること、そしてエラーメッセージが何が誤っているかを示してくれることに注目しよう。表示画面は仕様ソースの中でどこに問題があったかを表示し、関数トレースが呼出しスタックを、このケースでは一階層のみであるが、与えてくれる。関数トレース画面で３点をクリックすることで、リストされた操作呼出しの引数が拡張され得る。

インタープリタは、仕様のデバッグにも用いることができる。関数上にブレークポイントを設定することができるし、その後に相互作用的に実行を１ステップずつ進め、スコープ内の変数を調べることが可能である。

清書機能の一部として、仕様に色を用いてテストカバレッジ情報を清書することが可能である。VDM++ Toolbox は DOS または UNIX シェルの中で vppde (VDM++ development environment) をタイプすることで実行される。このコマンドは様々なオプションを取り、たとえば、“-t” はタイプチェッカーを起動し、“-i” はインタープリタを起動する。したがって、上記の仕様は次のタイプによって型チェックされる

```
vppde -t plant.rtf alarm.rtf expert.rtf
```

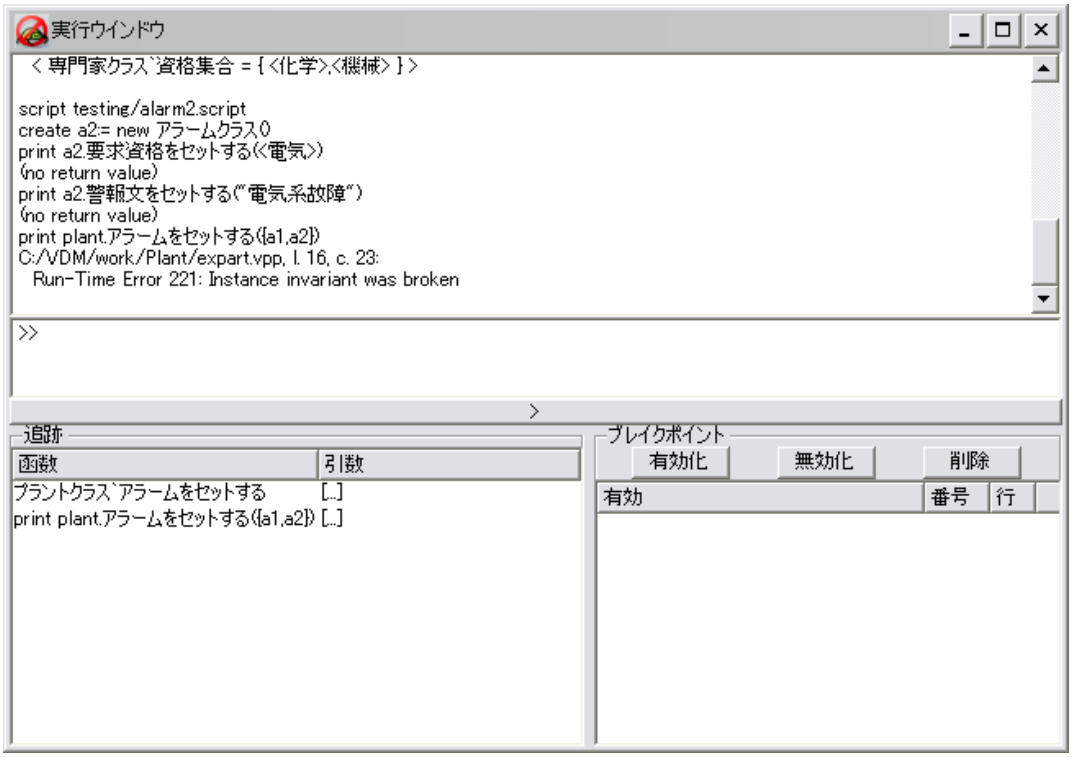

図 8: インタープリタにおける不変条件違反の中断

コマンドラインインタープリタでは、我々がいわゆるテスト引数ファイルを作成することを要求する。これには、インタープリタによって評価される式のリストを含めることができる。以下の `test1.arg` と呼ばれ上記のテストスクリプトのバージョンで動くファイルを、１つの例としてあげることができる:

```
new Test1().run()
```

これは新しいクラス Test1 を用いる:

```
class Test1

instance variables
 a1 : アラームクラス;
 ex1 : 専門家クラス;
 plant : プラントクラス;

operations
 public run: () ==> set of プラントクラス`時間帯型 * 専門家クラス
 run() == test1();

 test1: () ==> set of プラントクラス`時間帯型 * 専門家クラス
 test1() ==
   (alarm1();
    expert1();
    plant:= new プラントクラス();
    plant.スケジュールをセットする({plant.p1 |-> {ex1}});
    plant.アラームをセットする({a1});
    let periods = plant.専門家のスケジュールを得る(ex1),
        expert = plant.専門家を探す(a1,plant.p1)
    in
      return mk_(periods,expert));

 alarm1: () ==> ()
 alarm1() ==
   (a1:= new アラームクラス();
    a1.要求資格をセットする(<機械>);
    a1.警報文をセットする("機械系故障"));

 expert1: () ==> ()
 expert1() ==
   (ex1:= new 専門家クラス();
    ex1.資格をセットする({<機械>,<生物>}))

end Test1
```

その後以下のコマンドが実行される:

```
vppde -iDIPQ test1.arg plant.rtf alarm.rtf expert.rtf
```

コマンドラインからテストする間、インタープリタはテストカバレッジ情報を集めることができる。このためには、最初に以下のようなテストカバレッジファイルを生成する必要がある。

```
vppde -p -R vdm.tc plant.rtf alarm.rtf expert.rtf Test1.rtf
```

そしてその後上記の引数ファイルを基盤としたテストケースを実行する

```
vppde -iDIPQ -R vdm.tc test1.arg plant.rtf
                                  alarm.rtf expert.rtf Test1.rtf
```

生成された出力ファイルは期待された結果を含むものだ:

```
mk_( { mk_token(
 "月曜日" ) },
 objref7(専門家クラス):
 < 専門家クラス`資格集合 = { <機械>,<生物> } > )
```

結果であるテストカバレッジ情報は、付録 A における清書の仕様で表示されている。

もちろん上記の vppde コマンドはバッチまたは Windows や Unix のシェルスクリプトファイルから呼ぶことができる。システムテストではこのアプローチが推奨され、Toolbox ユーザーマニュアル [SCSc] においてソートの例題に描かれている。

5.2 ラピッド・プロトタイピングを用いた視覚による正当性確認

テストはモデルの正当性確認を行うためのすばらしいアプローチであり、技術者たちはテスト技術に精通している。しかしながら、上記で紹介した比較的低水準のテスト類は、主に技術者に適するもので、VDM (または UML) やモデルの詳細に親しんでるわけではない顧客や技術外のスタッフおよび管理者に対しては、あまりなじまないものである。VDM++ Toolbox は このような聴衆にグラフィカルなフロントエンドを通したモデル提示を行う必要に対して、機能的に働く API を

準備する。フロントエンドはこれによりモデルに対するグラフィカルなユーザーインターフェイスとして働くことが可能となり、顧客はGUIを通して、モデルを直接テストすることができる。アラーム例に対する簡単なフロントエンドの例が図9のようなものである。

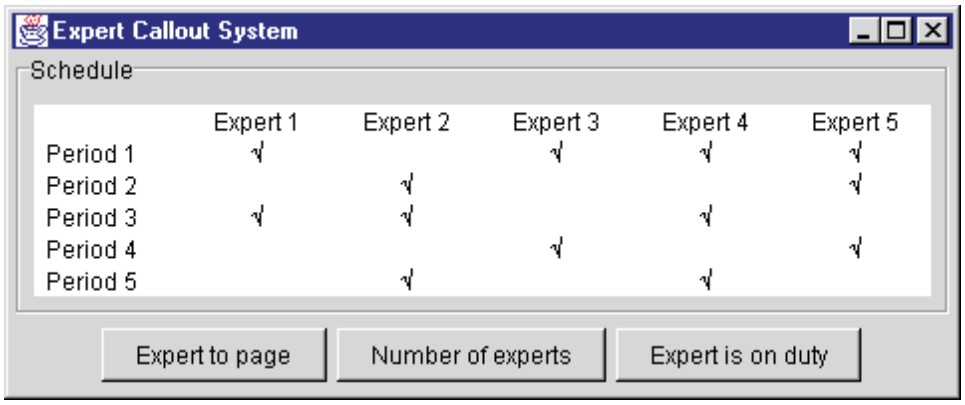

図9: アラームシステムに対するプロトタイプGUI

明らかにもっと優れたGUIが開発可能であろうが、それは本質ではなく、むしろユーザーはシステムの要求項目に対する開発者の理解をテストするため、簡単にこれを利用できるということの方が本質である。この方法を利用することで、VDM++モデルはシステムの早期のプロトタイプとして用いられる。VDM++から独立して、技術者の好みのツールで、フロントエンドをつくりあげることが可能である。上記のものはJava/Swingで開発されている。

このAPIはレガシーシステムと共にモデルテストに用いることも可能であり、実際にはプログラム言語から独立した他のコードであって、その一般的性質にしたがうものである。これはCORBA(the Common Object Request Broker Architecture)を用いているからで、オブジェクトを要求する仲介(ORB)が存在し、何らかの言語で書かれたコードとの通信を許している。現在すべての主要な言語に対するフリーあるいは商用のORBが複数存在する。CORBAがネットワーク基盤の構造であるということは、ネットワーク上に複数配置された外部の顧客が同時並行して1つのVDM++ Toolboxと通信し合えるということも意味する。そのためこの正当性確認アプローチは、分散モデルに対しても用いることができる。

6 VDM++モデルのコード生成

このVDM++モデルはアラームシステムの抽象化モデルであるが、JavaやC++のコード生成、コンパイル、実行、に対応できるほどに十分具体的である。VDM++ Toolboxでは、直接コンパイル可能で、製品レベルのコードを生成することができるような、自動コード生成を提供する。一般的に上記のように抽象化されたモデルの場合、コード生成されて顧客に納品されるまでにはもっと多くの手間が必要である。コード生成を行うためのモデルは一般的には設計や実装指向となるが、特に効率が強く要求される場合などはそうである。しかし、前の章で紹介したlet-be-such-that式のような高水準の式であっても、JavaやC++でのコード生成、コンパイル、実行は可能である。

実装を行うこの段階で明らかに強力な縮小が行えることに加え、自動コード生成はたくさんの利点をもつ。原則は、抽象モデルと生成コード間に強い対応性をもつことである。これがコードとその構成の理解を単純化してくれる。またこれが、従来モデルのテストに使用されていたテストデータの再利用に向けての可能性を高めている。もちろん生成されるコードに用いられるアルゴリズムが特定のインスタンスには適当でないこともあるかもしれないが、このような場合は後で手作業で修正できるようなスケルトンコードの生成が可能である。

7 まとめ

本書は、VDM++を用いたソフトウェアモデル構築にむけて、いくつかの手法ガイドラインを提示した。簡単な例である化学プラント向けのアラームシステムに、手法を適用している。我々は簡潔さを求めるため、提示したモデルでは詳細の多くを省略し抽象化に注意を向けたかった。しかしながらVDM++は、もっと具体的でかなり大きいモデルたとえば何百ページにおよぶUMLや何千行ものVDM++に相当するモデルにも、同様によい支援を行うことができる。今まで述べてきたアプローチでのキーは、UMLとVDM++の結合なので、UMLとVDM++の相補的な利点を最後にまとめることとする。

7.1 だれが Rose-VDM++ Link を使用すべきか

以下の２つユーザーグループは、**VDMTools** とその Rose-VDM++ Link の恩恵を得られるものと確信する:

1. UML のようなグラフィカルモデル記法をソフトウェア開発において現在適用しているあるいは適用したいと望む人々、またモデルのチェックや正当性確認を改善および自動化したいと望む人々。これに対する動機付けとしては、彼らのソフトウェアがある程度クリティカルなものかあるいはリスクを削減したいと考えるかであり、そのため VDM 技術を用いてできるだけ早くに要求項目を明確にしバグを見つけ、モデルへの早期の高い信頼を得たいと望むからである。
2. VDM++のようなオブジェクト指向形式仕様言語を開発に適用しているあるいは適用したいと望む人々、また文書化目的でグラフィカルな視覚化能力を処理したりあるいはモデルのチェックや正当性確認を改善および自動化したいと望む人々。

両グループともたぶん多かれ少なかれ同じやり方で、リンクを用いることになるであろう。第一のグループは UML の方でより多く、一方で第二のグループは VDM++の方でより多くモデル化を行おうとするかもしれないが、 リンクのラウンドトリップエンジニアリングが両グループにとっての中枢となるはずである。そのため、問題の全体像を把握するためにはグラフィカルな表記法がより優れていることから、ユーザーは UML レベルのモデル化からスタートすることで利益を得ることとなる。一旦モデルが比較的安定した状態に到達すれば、VDM++に変換しそこで分析を続けるとか、あるいは２つの表記法の間で必要に応じて行ったりきたりを切り替えることが、適切に行えるようになる。

7.2 UML におけるグラフィカルなモデル化

我々は UML と Rose は次のことにより適していると考える

- あるソフトウェアシステムのオブジェクト指向モデルの最初の作図を行う (ある程度は書面で補う) こと

- クラス名、属性や操作の名称、操作のシグネチャ、そしてクラス (継承、関連、役割、名称、他) 間の関係のようなモデルの視覚的な面を、グラフィカルに定義すること
- 例えば異なった図表示を通しての、モデルの視覚的で効率的なプレゼンテーション
- そして、抽象化された高水準のソフトウェアモデルの様相

このような UML と Rose の仕様例は本書で示されている。

7.3 VDMTools を使用したモデルの分析

VDM++ と **VDMTools** は、次の点においてより適したものであることを確信している

- 求められている機能の詳細な記述を行うこと、たとえばインスタンス変数における VDM++不変条件や、操作における事前条件や事後条件のように、要求項目のプロパティを形式化することによってである、
- 型チェックを用いてモデルの内部的一貫性をチェックすること、たとえば宣言の中で操作のシグネチャや型 (UML においては属性や関連) をチェックするというようなこと、
- どのような操作を行うべきかといったモデルの具体的様相を定義しチェックすること、通常ならばツールなどでチェックされ得ない自然言語で表現されるようなことをである、
- 文書化されたプロパティを一貫してチェックする一方、モデルの実行とテストに基づくシステマチックで反復可能なチェックプロセスを通し、モデルに対する信頼を得ること、そして、
- グラフィカルなフロントエンド、あるいは新しいコンポーネントが具体的に記されている既存のソフトウェアと共に、モデルが実行される場所でラピッド・プロトタイピングを行うこと。

SAFER モデルのあるバージョンを分析するための VDM 技術の効果的利用が、論文 [AL97, AS99] に記述されている。SAFER というのは、宇宙飛行士のための、修理を行うために宇宙船外に出るときに身に着けるバックパック部材である。安全面でかつセキュリティ面で重要な領域にある VDM 技術についての他の論文 [ALR98, LFB96]、もまた興味を引くものである。

参考文献

[AL97] Sten Agerholm and Peter Gorm Larsen. Modeling and Validating SAFER in VDM-SL. In Michael Holloway, editor, *Fourth NASA Langley Formal Methods Workshop*. NASA, September 1997. Available from http://atb-www.larc.nasa.gov/Lfm97/proceedings/.

[ALR98] Sten Agerholm, Pierre-Jean Lecoeur, and Etienne Reichert. Formal Specification and Validation at Work: A Case Study using VDM-SL. In *Proceedings of Second Workshop on Formal Methods in Software Practice*. ACM, Florida, March 1998.

[AS99] Sten Agerholm and Wendy Schafer. Analyzing SAFER using UML and VDM++. In John Fitzgerald and Peter Gorm Larsen, editors, *VDM in Practice*, pages 139–141, September 1999.

[FL98] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.

[GBR99] Ivar Jacobson Grady Booch and Jim Rumbaugh. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.

[Gro00] The VDM Tool Group. A “Cash-point” Service Example. Technical report, 2000.

[LFB96] Peter Gorm Larsen, John Fitzgerald, and Tom Brookes. Applying Formal Specification in Industry. *IEEE Software*, 13(3):48–56, May 1996.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice-Hall International, 1988.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International, 1991. ISBN 0-13-630054-5.

[SCSa] SCSK. *The Rose-VDM++ Link*. SCSK.

[SCSb] SCSK. *The VDM++ Language*. SCSK.

[SCSc] SCSK. *VDM++ Toolbox User Manual*. SCSK.

[SM88] S. Shlaer and S. Mellor. *Object-Oriented Systems Analysis*. Prentice-Hall International, 1988.

A　UMLモデルとVDM++モデル

この付録は、化学プラントの例におけるモデルを清書したVDM++クラスにテストクラスを加えて、リストする。各々のクラスで、**VDMTools**の清書機能によってテストカバレッジテーブルが挿入される。モデルの非実行部は赤(または灰色)に色づけされているので、プラントクラスで専門家の数を得る 操作を見てみよう。

図 10 における UML クラス図がこのモデルの概要である。

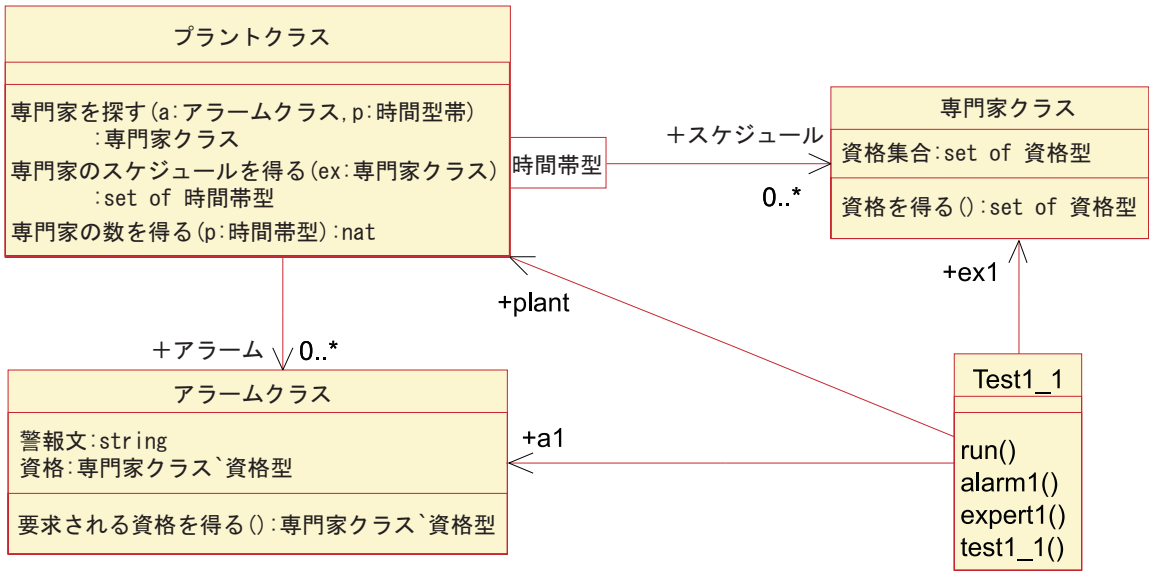

図 10: UML クラス図モデルの全体

A.1 プラントクラス

```
class プラントクラス

types
 public 時間帯型 = token;

instance variables
 アラーム集合 : set of アラームクラス := {};
 スケジュール : map 時間帯型 to set of 専門家クラス := {|->};
 inv
   forall p in set dom スケジュール & スケジュール(p) <> {};
 inv
   forall a in set アラーム集合 &
      forall p in set dom スケジュール e &
         exists ex in set スケジュール(p) &
            a.要求される資格を得る() in set ex.資格を得る();

operations
 public
 専門家を探す : アラームクラス * 時間帯型 ==> 専門家クラス
 専門家を探す(a, p) ==
   let ex in set スケジュール(p) be st
       a.要求される資格を得る() in set ex.資格を得る()
   in return ex
 pre a in set アラーム集合 and
     p in set dom スケジュール
 post let ex = RESULT in
       ex in set スケジュール(p) and
       a.要求される資格を得る() in set ex.資格を得る();

 public
 専門家のスケジュールを得る : 専門家クラス ==> set of 時間帯型
 専門家のスケジュールを得る(ex) ==
   return {p | p in set dom スケジュール
             & ex in set スケジュール(p)};

 public
 専門家の数を得る : 時間帯型 ==> nat
 専門家の数を得る(p) ==
   return card スケジュール ( p )
  pre  p in set dom スケジュール;
```

```
 public
 スケジュールをセットする: map 時間帯型 to set of 専門家クラス ==> ()
 スケジュールをセットする(sch) ==
   スケジュール:= sch;

 public
 アラームをセットする: set of アラームクラス ==> ()
 アラームをセットする(als) ==
   アラーム集合:= als;

values -- テスト用
  public p1:時間帯型 = mk_token("月曜日");
  public p2:時間帯型 = mk_token("火曜日");
  public p3:時間帯型 = mk_token("水曜日");
  public p4:時間帯型 = mk_token("木曜日");
  public p5:時間帯型 = mk_token("金曜日");

end プラントクラス
```

名称	*call* 回数	カバレッジ
プラントクラス`専門家のスケジュールを得る	1	100%
プラントクラス`専門家を探す	1	100%
プラントクラス`専門家の数を得る	0	0%
プラントクラス`アラームをセットする	1	100%
プラントクラス`スケジュールをセットする	1	100%
合計		***84%***

A.2　専門家クラス

```
class 専門家クラス

types
 public 資格型 = <機械> | <化学> | <生物> | <電気>;

instance variables
 資格集合 : set of 資格型;

operations
 public
 資格を得る: () ==> set of 資格型
 資格を得る() == return 資格集合;

 public
 資格をセットする: set of 資格型 ==> ()
 資格をセットする(qs) ==
   資格集合:= qs;

end 専門家クラス
```

名称	*call* 回数	カバレッジ
専門家クラス`資格を得る	3	100%
専門家クラス`資格をセットする	1	100%
合計		***100%***

A.3 Test1 クラス

```
class Test1

instance variables
 a1 : アラームクラス;
 ex1 : 専門家クラス;
 plant : プラントクラス;

operations
 public
 run: () ==> set of プラントクラス`時間帯型 * 専門家クラス
 run() == test1();

 public
 test1: () ==> set of プラントクラス`時間帯型 * 専門家クラス
 test1() ==
   (alarm1();
    expert1();
    plant:= new プラントクラス();
    plant.スケジュールをセットする({plant.p1 |-> {ex1}});
    plant.アラームをセットする({a1});
    let periods = plant.専門家のスケジュールを得る(ex1),
        expert = plant.専門家を探す(a1,plant.p1)
    in
      return mk_(periods,expert)
  );

 public
 alarm1: () ==> ()
 alarm1() ==
   (a1:= new アラームクラス();
    a1.要求資格をセットする(<機械>);
    a1.警報文をセットする("機械系故障"));

 public
 expert1: () ==> ()
 expert1() ==
   (ex1:= new 専門家クラス();
    ex1.資格をセットする({<機械>,<生物>}))

end Test1
```

名称	***call*** **回数**	カバレッジ
Test1`alarm1	1	100%
Test1`expert1	1	100%
Test1‘run	1	100%
Test1`test1	1	100%
合計		***100%***