ソフトウェア工学 80－8
（1991. 7. 18）

VDMによる記述実験に基づく設計プロセスの一考察

松浦 佐江子
（株）管理工学研究所

概要

多様なシステム化の対象に対して、様々な言語や設計の支援のためのツールが提案されている。しかし、設計の支援を考えた場合にも、言語やモデルの形成等の断片的な支援ではなく、設計のプロセス全体を支援する統合的なツールが必要である。そのためには、設計のプロセスを明確にし、それに沿った支援を考察する必要がある。設計のプロセスを考える場合、設計を静的に見た組織としての役割と、その組織をいかに運営するかの計画が重要であると考える。本論文では、形式的な設計技法であるVDM（Vienna Development Method）を1つの具体的な設計プロセスの例として、設計のプロセスをその組織と計画の観点から考察する。

An Experimental Consideration of Design Process on VDM

Saeko Matsuura
Kanri Kogaku Kenkyusyo, Ltd.
Ebisu CS Bldg. 1-9-6 Ebisuminami, Sybuya-ku, Tokyo 150 Japan

As a variety of systematization objects has become increased, various tools which support programming languages and software design methods are proposed. At design phase, these supports are fragmentary. And we need to construct an integrated tool which supports through the whole design process. So we must make clear design process and consider how to support it. On a design process, there are a static part and a dynamic part. The former is an organization of design and the latter is a plan to manage it. In this paper we discuss about design process in such two viewpoints by analyzing a concrete design method VDM(Vienna Development Method) for an example.

1 はじめに

多様なシステム化の対象に対して、様々な言語や設計の支援のためのツールが提案されている。しかし、ソフトウェア開発の支援を考えた場合、言語やモデルの形成等の断片的な支援ではなく、設計のプロセス全体を支援する統合的ツールが必要である。そのためには、設計のプロセスを明確にし、それに沿った支援を考察する必要がある。いわゆる設計技法は設計の断片（さまざまな中間生成物）とそれらを組み立てていく指針を規定した1つのプロセス・モデルであると考えられる。そこで1つの例として具体的な設計技法のプロセスを分析することには意義があると思われる。本論文では、形式的な設計技法である VDM(Vienna Development Method)[3][4] を例として設計のプロセスを考察する。

2 VDM における設計プロセス

設計プロセスを仕様の静的な構造を表す組織（枠組）とその組織をいかに運営するかといった設計の計画（方針）という2つの側面で考察する。組織ということを考える場合、人間の社会においても、人間の集合だけでは、意味のある活動を行なう社会とは言えない。社会の活動のためには、個々の人間の関係が必要である。このような集合の要素とそれらの関係によって、ある種の組織が形成され、それらが社会を形成する。また、数学の概念で、カテゴリ理論がある。これは数学的な対象をオブジェクトとその関係（モルフィズム）によって抽象的に表現する体系である。この体系によってこれまでいろいろな分野で考えられてきた概念を統一的に表現することができる。最近では計算機科学、特にプログラミング言語の分野においてもこの体系で研究することが試みられている。

一方、組織だけ与えられてもその組織をどのように運営するかといった方針や戦略がなければ、組織は行動できない。どのような計画を与えているかが技法のポイントである。そこで、本論文では、VDM の設計プロセスをこのような視点から考察し、他の設計技法の設計プロセスを考察する指標としたい。

- 組織
  組織を仕様を構成する要素とその関係によって定義する。要素とそれらの関係を1つの組織とすると、組織はまた要素となりうる再帰的な構造である。
- 計画
  設計の手順、および着目点を上記の組織に対してどのように適用するかといった設計方針および戦略である。

2.1 VDM の特徴

VDM は1970年代に研究が始った、数学的な言語体系を与えてその枠組みによって検証を行なう形式的な設計技法である。VDM における仕様とは要求を満たすシステム全体を1つの状態として捉え、その状態の変化としてシステムのモデルを構築したものである。はじめは要求を how としてではなく、what として捉える。そしてその仕様を段階的に洗練して実現に近づけていく方法である。特徴は段階的に詳細化する場合における対応の厳密な記述と証明による検証である。要求に対するシステムの正当性は最も抽象的に要求を捉えた what の段階に行ない、次段階の仕様との間は回復関数の記述により保証する。VDM 自身では、大規模な問題に対する分割や並列性の方法は規定されていない。厳密な記述による仕様レベルにおける正当性や一貫性の保証を重視した技法である。一貫性の保証とは、設計の意図をプログラムに反映させることを意味する。

2.2 VDM における組織

本項では、組織を構成する要素や関係を、VDM の定義に従って説明する。いわゆる VDM におけるキーワードである。ここで、洗練化の各段階をレベルと呼ぶ。図1参照。

1. ステート
   有効な状態をもったクラス又は集合を意味する。
   ステートはデータの型とそのデータの制約条件（データ型不変式）および初期条件を含む複合データ型である。ここで、データ型不変式は、ステートで定義されたデータの制約を記述した述語である。
   この条件はオペレーションによるステートの変化においても満たされなければならない。
   ステートは集合としてデータ型、その関係として2つの述語による規定をもつ組織である。
2. オペレーション, ファンクション
   ステートを変化させる操作である。ステートの初期状態に関する述語とステートの初期状態および、終了状態に関する述語によって記述される。組織であるステートの関係を規定する。ステートとこ

れらのオペレーションで1つの組織を構成する。これを定義と呼ぶ。定義は各レベル毎に作成される。

3. 回復関数
具体化したレベルと前段階の抽象的なステートのレベルとの関係を厳密に規定する関数である。回復関数によって、段階的洗練における対応を保証する。VDM の仕様はレベルの列と考えられ、そのレベル間の関係を規定するのが、回復関数である。

4. 直接定義
ファンクションやオペレーションの記述に対して直接定義を行ない、証明によりファンクションやオペレーションの記述の正当性を確認する。前記の第2項目と同様、ステートの関係を規定する。同レベルにおける第2項目の定義との関係を規定する証明である。

5. 証明義務
洗練化段階ごとにさまざまな証明の義務が生じる。例えばつぎのようなオペレーション等の実現可能性や回復関数による各レベル間の洗練の妥当性の証明等の証明義務がある。すなわち、オペレーションの有効性を証明する。
$\forall \sigma' \in \Sigma \cdot pre - OP(\sigma) \Rightarrow$
$\exists \sigma \in \Sigma \cdot pos - OP(\sigma, \sigma')$
$\forall a \in A \cdot \exists r \in R \cdot ret(r) = a$
前者は第2項目の証明であり、後者は各レベルにおける定義間の関係の証明である。

モジュールは上記の定義と他のモジュールとの関係（インタフェース）を規定した組織である。他のモジュールのインポート、あるいは他へのイクスポートができる。

2.3 VDM における計画

VDM における計画を組織形成の手順（方針）および戦略の観点から考察する。2.2項で述べた組織の規定の手順はつぎのとおりである。

1. ステート定義
ステートを記法（META-IV、BSI(British Standard Institute) の記法等）に従って記述する。データは集合・列・写像・レコード型の形式で記述される。データ型不変式はデータに関する述語である。

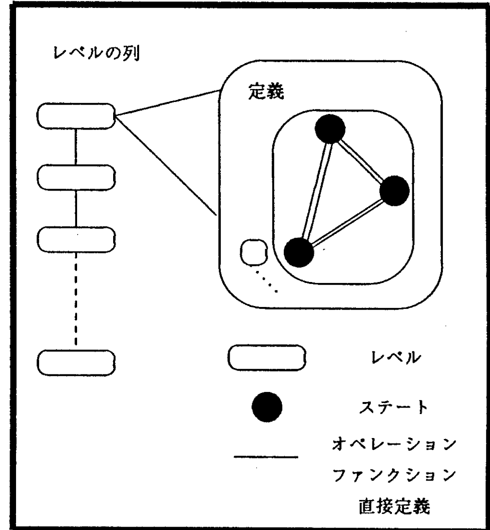

図 1: VDM における組織

2. オペレーション定義
ステートに対する要求されたオペレーションを記法に従って前条件、後条件として記述する。参照するステートの宣言と、ステートのオペレーションによる操作の前状態と後状態に関する述語として定義される。

3. 仕様の証明
2.2項の第5項目で述べたようなことを各オペレーションやファンクションに対して証明する。同レベル内での証明である。

4. 次段階のステート等の定義
上記の 1～ 3を次段階の仕様として記述する。

5. 回復関数の定義による検証
洗練化したレベルと前段階のレベル間の回復関数を記述し、それによって2.2項の第5項目で述べた証明を行なう。

すなわち、2.2項で述べた1レベルの組織を構成し、それを繰り返すことによって仕様を構成する。VDM では、初期の仕様は実現を考慮せず、抽象的に表現される。設計、アルゴリズム、実現の詳細を含むように開発を進める2つの戦略が提案されている。それがデータの洗練とオペレーションの分解である。

- データの洗練
  データをより実現に近づけた新しいステートを定義し、新しいステート上でオペレーションを再定義することである。

- オペレーションの分解
  ステートはそのままで列、選択、繰返しといった制御構造を使って、より単純なオペレーションの結合にオペレーションを再定義することである。

3 記述実験

VDM における設計プロセスを前記の 2 つの観点から考察するために、ソフトウェアの仕様と設計に関する国際ワークショップにおいて提示された問題 [7] の中から図書館システム、フォーマッタ、リフトコントロールシステムの 3 つの記述実験を行なった。問題や結果の詳細は、[9] を参照のこと。なお、付録に記述の一部を示す。以下において各問題を [LIB]・[FOM]・[LIFT] と略記する。本節では、以下の点に着目して、記述実験結果を問題毎に整理する。はじめの 2 項目は組織の観点で、他の項目は計画の観点で考察する。

- システムの捉え方
- 制約の与え方
- データの洗練
- オペレーションの分解
- バックトラック
- 検証

3.1 システムの捉え方

与えられた問題に対してある状態の変化として対象をモデル化しようとするために、トップの状態＝システムの状態をまずはじめに定義し、その状態の変化としてシステムをモデル化する。

- [LIB] では 図書館 という 本の集合 をステートとし 貸し出し や 返却、検索 によってその集合がどのように変化するかをオペレーションとして記述する。

- [FOM] では入力された文字の集合を出力形式の文字の集合とすることから、文字の集合 が最上位のステートとなる。オペレーションは要求にある条件を満たすように フォーマットする ことである。

- [LIFT] では、全状態（＝システム状態）すなわち コントローラ をステートとし、リフト や フロアー の情報を管理する。コントローラの状態遷移がこのリフト群を最適に運行することである。

全体を 1 つの状態として意識して設計した。最終的なモデルのイメージは以下の通りである。

- [LIB] は、本の集合 と ユーザ によって構成された図書館に対して 1 人のユーザが行なう操作による状態遷移としてモデル化した。これは繰り返し行なわれるが、ここでは複数のユーザに対する仕様は要求されていないので、この繰り返しのメカニズムは考慮しなくてもよい。遷移毎に 本の集合 の状態は変化する。図書館はこのシステムが順次動作することによって運営される。

- [FOM] は、文字列 というステートの 1 回の状態遷移が要求された仕様である。1 回毎の遷移は独立して考えられる。

- [LIFT] は コントローラ のステートの遷移としてモデル化したが、は リフト やは ボタン の独立な動作との関連が表現できなかった。

VDM のようなモデル化をした場合、問題の特徴と関連して、つぎのことが言える。[LIFT] では、コントローラのステートの遷移のサイクルが重要な情報である。外からコントローラに作用する動作に対してどう対処するかが、問題である。これは、システムの状態＝全状態としてモデル化したために、前記 2 つのモデルはその状態の内部情報に外からアクセスすることがないのに対して、[LIFT] は内部情報としてのリフトとボタンの 2 つに対する外界とのやりとりを行なうシステムであるからである。すなわち、リフトやボタンをコントローラと同列に見たモデル化が必要である。

3.2 制約の与え方

データに関する条件は一般に機能の表現の中で実現される。VDM では、データ型不変式として、データに対する条件として分析する手段を与えている。ただし、早い段階から制約を記述すると、洗練化するごとに、制約が増えて、各段階ごとに証明しなくてはならない。

- [LIB] では、要求に現れた条件に従って記述した。

  inv 貸し出し状態 (*library*) $\triangleq$

∀ *copy_book*∈ *lib_book*(*library*)
· *book_state*(*copy_book*) = '貸し出し中'
∧ *book_state*(*copy_book*) = '貸し出し可';

- [FOM] では、条件付き文字列である 単語 の概念を記述した（付録参照）
- [LIFT] では、直接要求の中には現れないが、リフトやボタンに関するデータの考察からいくつかの不変式を記述した。
  inv 向き ： リフト → B
  inv 向き（*lift*）
  (*lift_state*(*lift*) = '運行中'
  ∧ direction(*lift*)∈ {'上','下'})
  ∧(*lift_state*(lift)∈{'停止','非常停止'}
  ∧ direction(*lift*) = *nil*);

3.3 データの洗練

本項以降では、VDM における計画として洗練化の２つの戦略（データ洗練・オペレーション分解）について考察する。

- [LIB] では、始め 図書館 を 本の集合 として捉えた。まず、見えている状態から表現してしまったが、本を抽象化した本の識別子を引き出して、それをキーとして、次の情報を得る写像として表現した方がより抽象的な仕様と考えられる。この理由は第4.3項で述べる。以下のような２つのステート定義が考えられる。

  (I)図書館 ::lib_book : 本のコピー -set
  本のコピー ::book_id : 本識別子
  book : 本
  book_state : 貸し出し状態
  book_borrower : 借り手氏名
  borrow_date : 貸し出し日;
  ユーザ = { '職員','一般利用者'}
  (II)図書館 :: lib_book : 本識別子 $\xrightarrow{m}$ 本のコピー
  user : ユーザ識別子 $\xrightarrow{m}$ ユーザ;
  本のコピー ::book : 本
  book_state : 貸し出し状態
  book_borrower : 借り手氏名
  borrow_date : 貸し出し日;
  ユーザ :: user_class : ユーザクラス
  user_name : ユーザ氏名;
  ユーザクラス= { '職員','一般利用者'}

- [FOM] では登場するデータは文字の列である 文字列 のみである。しかし、要求の中には、単語 とか 行 といった条件の付いた文字列が現れる。データ型不変式でこれらを表現するのは容易である（付録参照）。データ構造が単純であり、文字 列 という表現から、文字 -set ではなく 文字* として表現した。
- [LIFT] では、コントローラ を複数台の リフト と複数の フロアー から構成され、それらを抽象化した ID から必要情報への写像として、表現した。さらにリフトやフロアーを独立した状態としてデータ構成を行なった。

まず、要求から大まかなステートを構成した。この時、曖昧なものは写像として構成を先送りする。[FOM] はデータ構造が単純であったので構造を変更することはなかったが、[LIB] や [LIFT] では、オペレーションの分解にともない変更した。しかし、構造上の変更は比較的容易である。

3.4 オペレーションの分解

第一段階の仕様としては、抽出したステートを変化させる操作はオペレーションとして、そうでないものはファンクションとして記述する。抽象レベルでは、要求されている操作の満たす条件を記述し、順次アルゴリズムを考える。[FOM] では、要求の表現がステートの満たす条件として記述されているため、そのまま表現できた。[LIB] では、図書館のステートに作用するオペレーションやファンクションの名前が明記されているために、各操作名の機能に対して制御構造を考えれば良い。他の２つの場合は、操作名となるものの抽出から行なわなければならなかった。

3.5 バックトラック

ここで考えられるバックトラックには、各レベルの構成における思考の戻りと前のレベルへの戻りとが考えられる。ステートの変更は構造の追加、オペレーションの追加は述語の列挙であるから、容易である。しかし、情報を管理するツールの支援が必要である。[LIB] の場合、データ型不変式によってデータの情報の欠落を見つけることができた。

- [LIB] では、問題文の表層に現れる項目からデータを構成したので機能の考察によって、データの変更が起きた。図書館におけるユーザの役割が見えず、最上位のステートを変更した。制約に記述において、データ型を変更するか、不変式を定義するかの2通りが考えられた。

3.6 検証

[LIB] の場合に、ファンクションの直接定義による実現可能性の証明と、ステートの変更にともなう、回復関数を定義し証明を行なった。証明は任意の値に対する有効性を明らかにする意味で重要なことではあるが、記述は面倒であるし、証明結果の記述量はかなり多くなる。適切な支援を考慮する必要がある。

4 分析および考察

第2節で述べた組織と計画の観点から上記の記述実験の結果を以下のような項目について分析し、考察する。比較例として、JSD(Jackson System Development) 法 [2] と実行可能仕様記述言語 PAISLey[10] を用いる。

4.1 組織

VDM における設計の組織について考察する。

- システムの捉え方（全体像）
- 関係の規定（制約の与え方等）

4.1.1 システムの捉え方（全体像）

思考過程の履歴やモジュールの構成について述べる。

●組織の特徴から、まずステートの分析を行なう。データと機能は相補しながら構成するものであるから、VDM での1つのレベルにおいてステートをデータとその不変式の関係から分析したり、定義をステートとオペレーションの関係から分析したりすることによって、あるレベルにおける仕様を確定できる。仕様を構成していく時、試行錯誤によってどこまで入り込むかが不明確になりがちである。JSD の場合には、設計は大きくわけて実世界をモデリングしたモデル・プロセスとシステムの機能を付加したシステムモデル・プロセスの2段階で構成される。VDM の場合にも、システムと実世界という境界は設けないが、各レベルに分けることで、思考の範囲を限定して段階的に整理できる。また、ステップごとの履歴は重要であると思われる。

● PAISLey におけるシステムの捉え方は、システムを状態の変化として捉える点は VDM と同じである。並列に動作するプロセスの状態の変化の様子を関数で記述する。この時各プロセスの情報のインタラクションを交換関数によって規定することができる。しかし、実現への変換は考慮されていず、履歴は関心事ではない。

● VDM は意味定義言語の枠組みであるから、モジュール構成がうまくできない。そのために条件の羅列になって全体構造がよくわからない。このような記述だけで自動的に実現プログラムが生成できるならばよいであろうが、人間が洗練化していくためには、実プログラムのターゲット言語を定めた指針が必要であろう。

4.1.2 関係の規定（制約の与え方等）

組織のインターラクション、モジュールの分割について述べる。

● VDM では1つの状態の遷移として考えてしまうために、その状態の要素が独立に動作するような状況を記述できない。（[LIFT] の場合） VDM ではモジュールとその間の関係を規定する方法がない。すなわち、JSD のようなプロセス間の結合状態を定義する仕組（CSP）や PAISLey におけるプロセス間の交換関数をもたない。VDM を基礎にした RAISE(Rigorous Approach to Industrial Software Engineering)[1] では、構造化の枠組や仕様の並列性等を考慮している。

●組織としては、組織間のインターラクションの規定が必要である。

●ステートの定義から行なうのでデータを分析することが先になる。ステートを段階的に詳細化していけばよいが、言語に不慣れであったので、抽象レベルにおける表現を確定するのが難しかった。ただし、集合による記述は曖昧なものを表現するにはやりやすい。そこから出発して徐々に構造を作っていけばよいからである。しかし、設計の構造としてはモジュール分割の規定が曖昧である。

4.2 計画

前記の組織をいかに運営するかという計画についてつぎの2つの観点から考察する。

- 洗練化の方法
- 検証

4.3 洗練化の方法

全体的な洗練化の方針矢データの洗練化の方針について述べる。

● VDM では段階的洗練化を特徴としているが、これは豊かな表現力をもつ言語の特性を利用して、抽象レベルから具体的なアルゴリズムレベルまで記述可能なことから考えられることである。すなわち、洗練化を行なうためには、このような記述言語が必要である。しかし、豊かな表現力は、裏を返せば、自由度が大きく難しい。そこで、データ洗練やオペレーション分解の指針として実現レベルの目標言語の知識（言語の意味）を利用して洗練化の方針を与えることも考えられるだろう。

●まずトップ・ダウンに入るので、考察がしやすい。しかし、つぎのステートを記述する際に何をステートとするかが難しい。[FOM] のようなデータが満たすべき条件として仕様が記述されている場合には要求文をそのまま素直に表現できる。逆に [LIB] のように要求されている機能が明確でそのアルゴリズムがわかる場合には、機能の満たすべき性質として抽象レベルを記述するのはわかりにくい。[LIFT] の場合も要求されているアルゴリズムよりも要求されているものの状況の説明になっている点で [FOM] と同様である。このようなアルゴリズムを分析するために段階的にオペレーションの分解を行なうことになる。

● 3.3節で見たように、集合のような全体が見えているプレーンな構造に対し、データのキーワードを抽出し、そこからの写像として次の情報を得るといった階層的な構造を作ることがデータ洗練の1つの方針である。そしてある程度洗練化された下位においてプレーンな構造に列等の操作しうる構造を入れる。この間はオペレーションは抽象的なデータに対する機能として分割できる状態であると考えられる。すなわちオペレーションがデータの構造を操作する段階になった時に操作しうる構造を入れる。

4.4 検証

VDM における検証について述べる。

● VDM では、要求を表現した各レベルに対する Verification を厳密に規定しているが、いわゆる要求に対する Validation に関しては、ユーザが表現した最初の抽象レベルの検証に留まってしまう。すなわち、分析的ではないので、問題の表現に左右されやすい。JSD 法では、実世界に基づく分析から、問題文の表現に依らない分析ができるという一面がある。しかし、VDM の組織的な洗練によって必要な情報は抽出できると思われる。

●要求はあくまで分析するための指針であるから、技法は要求の本質を捉えた組織化が行なえるものでなくてはならない。

●前記の 4.3項で述べたように、抽象的な表現から具体的な表現への洗練はどのような技法でも行なっていることである。しかし、その間の一貫性の保証や、思考の履歴の保存を行なっているものはあまりない。（フォーマルな技法と呼ばれるものはあるレベルにおける一貫性の保証はしている。）しかし、VDM の場合は回復関数によってこの 2 つの表現間の一貫性を保証する。他の方法頭の中で行なっている抽象的なものから具体的なものへの変換の保証を形式的には行なわずに、最終的な結果に対して検討することになる。ただし、2つのステートのギャップが大きければ、この証明も難しいであろう。

●思考過程をほとんど記述しようとするので、記述量が膨大になる。VDM では、各段階や証明を記述物として残すが、他の技法では、この思考は頭の中では行なってても、文書として残すことはない。頭の中のいい加減な検証より役立つが"明らか"とは言えないところが少々面倒である。

5 おわりに

本論文では、VDM の設計プロセスをその組織と計画の観点から考察した。VDM では組織としては組織間のインターラクションの規定が必要である。設計技法における重要なポイントは、組織をいかに運営するかの計画である。VDM では段階的洗練という大きな戦略はあるが、詳細部分における戦略が不足していると思われる。洗練するためには目標を定めてその方向に変換していかなくてはならない。そこで、例えば目標言語を定めてそのセマンティクスを知識とした戦略を考えることは有効であると思われる。

また、今後の課題として、組織と計画の観点から他の設計技法を考察し、設計プロセスにおける有効な指針や戦略について考察したいと考える。

謝辞

本研究の一部は、（社）情報サービス産業協会・（株）協同システム開発の委託による「やわらかなソフトウェアに関する調査研究」によるものである。

A フォーマッタの記述例

```
module 文字列
definitions
types
文字列=文字*;
文字 ::char: 名前
      class: 種類;
種類 = [分離文字];
分離文字= {'空白','改行'};
単語=文字*;
state of sequence: 文字列;

inv 単語: 単語 → B
inv 単語 (word ) ≜ ∀c ∈ word
·c ∉ {'空白','改行'};
init(sequence0 ) ≜ [ ]
end;
functions
単語列 : 文字列 →m 単語*
単語列 ( sequence : 文字列) b : 単語*
pre sequence ≠ [ ]
post ∀w ∈ b · ∀c ∈ elmsw · c ∈ sequence;
last(s:X*)rs : X
pres = []
posts = subseq(s, 1, lens − 1)⌒ rs;


operations
空白列を出力する ( )
ext wr sequnece: 文字列
    rd breakchar: 分離文字
pre ∃s ∈ { sequence (i,...,j ) |
∀i, j ∈ indssequence ∧i ≤ j ∧ j − i ≥ MAXPOS }
        ·∀c ∈ s·c ∉breakchar
post sequence = [ ];


フォーマットする ( )
ext wr sequnece: 文字列
    rd breakchar: 分離文字
    rd maxpos: Maxpos
pre ∀s ∈ { sequence‾(i,...,j)|∀i, j ∈inds sequence‾
∧i ≤ j∧j − i ≥ maxpos + 1 }
·∃c ∈ s · c ∈ breakchar
post ∀c ∈ sequence‾ · c ∈ sequence∧
単語列 (sequence‾)⊂ 単語列 (sequence ) ∧
hd(sequence) ∉breakchar∧ last(sequence) ∉breakchar∧
(∀s ∈ { sequence(i,...i+1) | ∀i∈ inds sequence }
·∀c ∈ s · class(c) ∈ breakchar)∧
∀i ∈inds sequence·∃c ∈ sequence(i,...,i+maxpos)
· class(c)='改行'∧
∀s ∈ { sequence(i,...,j) |
∀i, j ∈inds sequence ∧ i ≤ j ∧ j − i ≤ maxpos }
⊂ { sequence( i,...,j) |
(i = 1∧class(sequence(i))='改行')∧
(class(sequence(j))='空白'
∧ j=len sequence)·∀c ∈ sclass(c) ≠ '改行';
end 文字列
```

参考文献

[1] D.Bjørner,A.E.Haxthausen,K.Havelund: Formal,Model-Oriented Software Development Method, Proc. of InfoJapan'90,IPSJ,1990

[2] J.R.Cameron:An Overview of JSD,IEEE Tr.SE 12-2,222-240,1986

[3] C.B.Jones:Systematic Software Development using VDM, Prentice-Hall Intl.,1986

[4] C.B.Jones and R.Shaw:Case Studies in Systematic Software Development,Prentice-Hall Intl.,1990

[5] 加藤：情報システム開発法の比較，ソフトウェア工学研究会, 75-1, 1990

[6] 古宮 他，仕様記述過程モデル化のための実験と分析，ソフトウェア工学研究会,69,1990

[7] Problrem Set for the 4th International Workshop on Software Specification and Design, Proc. of 4th International Workshop on Software Specification and Design,pp.ix-x,1987

[8] 佐伯：ソフトウェア仕様化・設計の方法論の形式化について，ソフトウェア工学研究会, 72-4, 1990

[9] やわらかなソフトウェアに関する調査研究成果報告書，ソフトウェア開発に関するプロセス・モデルについての調査研究，情報サービス産業協会,1991

[10] P.Zave : An Overview of the PAISLey Project-1984 , SIGSOFT SEN ,Vol.9 , No.4 , pp.12-19 , 1984