ソフトウェア工学 100−13
（1994．9．8）

ＣＯＢＯＬプログラムから非手続き仕様を逆生成するリバースエンジニアＣＯＲＥ／Ｍ
A COBOL reverse enginer CORE/M - generation of a nonprocedural specification through COBOL program understanding -

原田　実*,吉川彰一**,永井英一郎***
*Minoru Harada, **Shouichi Yoshikawa, ***Eiichirou Nagai

梗概

プログラムの保守を効率化するためには、プログラムから仕様への逆生成が重要である．本研究では、ファイル処理問題に対するCOBOLプログラムから形式的な要求仕様を逆生成する手法を提案するとともに、それを実現するシステムCOBOL Reverse Engineer for Modules: CORE/Mを開発した．プログラム理解の結果を表す要求仕様としては、等関係仕様を採用した．この仕様は、ファイル処理問題における実体や関連の属性項目間の関係と出力・更新要求を等関係式という数式の集合で表したもので、要求を非手続的に表現できる．ここでは、各計算式の項目は、"."を介してそれが所属する実体の識別子によって、また"_"を介してそれが格納されているファイル名によって修飾されている．従って、等関係式はその左辺項目の充分な意味定義になっている．

CORE/Mはまず COBOLプログラムを構文解析し、ブロック構造に展開する．次にブロックが処理対象とする実体や関連を表す照合基準を抽出する．この照合基準を元に、実体識別子とファイル修飾子を導出し、ブロックに含まれる処理文内の項目を適切に修飾して等関係式に変換し出力する．事例として、176行のCOBOLプログラムを変換した結果、元のプログラムと同値の等関係式仕様を得た．今後の課題として、等価変換を用いて行儀の悪いプログラムも理解できるようにすることなどがある．

Abstruct

I propose a technique of reverse engineering of COBOL programs, and develop a "COBOL Reverse Engineer - CORE/M -" based on this technique.

CORE/M generates the non-procedural EOS specifi-cation from COBOL programs, especially performing the file processing. EOS specification consists of a set of equations, called equality relation. Each equality relation shows the relationship among the attribute items of entities. In this, each item is modified with "." by the entity having it as an attribute, and suffixed with "_" by the file storing it.

CORE/M, first, parses the COBOL program, and develops it into a block structure. Next, CORE/M decides which entities are processed under what conditions in each block. It can be thought that all the statements in a block are processing the same entity fulfilling the same condition. Thus, CORE/M basically modifies every item in such a statement with the same entity identifier and the same file modifier.

CORE/M actually converted the sample Cobol pro-gram consisting of 176 lines, and has generated its EOS specification bearing the same meanig.

*青山学院大学理工学部経営工学科

,*青山学院大学理工学部経営工学科　(**は現在日本電気株式会社)

1. はじめに

近年では、企業の情報処理部門において、保守業務が新規開発業務より大幅に増加し、その比率は保守:新規が7:3とも言われている[14]. これは、外部環境の変化や企業活動の見直し(ﾘｽﾄﾗｸﾁｬﾘﾝｸﾞ)やダウンサイジングによる稼働環境の変化による、ｼｽﾃﾑの再構築の必要性からきていると思われる.

企業では人事移動などにより、初期の開発者とは別の人が保守業務にあたることも珍しくない. また、対象システムに対して、開発中の仕様変更などにより最終的なシステムの仕様書が存在しないことも多い. 従って、システムを正確に保守するために、他人の書いたプログラムを直接読んで理解しなければならなくなっている. もし、完全な仕様書があればプログラム全体を読むよりは、より早く修正箇所を見つけられる. さらに仕様書がCASEツール化されていれば、仕様書の修正だけで新しいプログラムを自動生成することもできる. 従って、プログラムから仕様書、できることなら何等かの形式的意味を持ちCASEツール化されている仕様書、に逆変換するツールがあれば、膨大な保守業務を効率化することができる.

この様なプログラムから仕様書への逆変換の技術は、ソフトウェア工学分野ではリバースエンジニアリングと呼ばれ、また人工知能分野ではプログラム理解と呼ばれ、これまで多くの研究が為されている.

リバース・エンジニアリングには、大きく分けて2つのテーマが存在する. 1つはデータのリバース・エンジニアリングであり、もう1つはプログラムのリバース・エンジニアリングである[8].

前者は、プログラムのデータ定義部の記述から、プロジェクト内での同意のデータ名を識別してデータ名の統一化を行ったり、正規化された論理データモデルを生成するものである. 商用化されているものとしては、Chen & Associates社のE-R DesignerやBachman Information Systems社のBachman Re-engineering Product Setなどがあり[8]、既に実用の域に達している.

後者は、プログラムの手続き部の記述からモジュールや関数や式などのプログラムの個々の機能単位の意味を抽出するものである. 現状のツールでは、抽出した機能の表現方法は、フローチャートや構造化チャートやせいぜいデータフロー図などである. 従って、これらのツールでは入力と出力の抽象度のレベルが同程度であり、プログラム理解においては視覚性の向上程度の貢献しかない. このようなリドキュメンテーションツールには、C言語からSC(Structured Chart)を出力するVEST SOFTWARE社のVEST-SAVERやInterFace Computer社のPLASMAなどがすでに商用化されている[8].

より進んで、もっと抽象度の高い仕様への逆変換を行うものは、人工知能分野のプログラム理解の研究と差がなくなっている. これらには、例えば、WillsのRecognizer[17]、JohnsonのProust[12]、AdamのLAURA[1]、上野のIntelliTutor[15]、富士通研究所の開発保守支援システム[18]、BiggerstaffのDesire[4]などがある.

この内最初の4つでは、プログラム中によく現れる典型的なアルゴリズム要素やデータ構造要素をクリシェあるいはプランと呼ぶ知識として持っている. システムは対象プログラムを専用のグラフ表現に展開し、その中からクリシェを自動的に同定し、同定したクリシェの意味とクリシェ間の関係に基づいて設計書やバグレポートを生成する. しかし、この方法では、知らないクリシェを持つプログラムを理解できないし、また実規模程度のプログラムグラフから個々のクリシェを探索するのに膨大な時間を要するなどの問題がある. 富士通研究所で研究されているシステムはCOBOLプロセスから日本語文章によるプログラム解説書を逆生成する. この際クリシェを理解して、事務処理に多いデータの合成分解自の作業変数を消去したり、スイッチとして用いられる作業変数を消去したり、条件分岐式を表形式に展開するなど抽象度を上げる工夫をしている. しかし、プログラム全体が表す大きな制御構造を同定する段階には至っていない. Desireでは、モデルで用意された典型的な概念構造とプログラム上の手続き名や変数名やコメント中の言葉などとの間の経験的に得られた相関関係を利用して、モデルとプログラムの関連付けを行う方法をとっている. しかし、この方法ではプログラムの深い理解はできないし、言葉の一致が偶然によるものかもしれないという信頼性の問題もある.

このように、プログラムのリバース・エンジニアリングはデータのリバース・エンジニアリングに比べると、機能の点においても実用性の点においてもまだまだ遅れている. この大きな理由の1つは、逆生成する点でもまたその仕様からフォワードにプログラムを生成する点でも、理解結果を表す形式的仕様としてどのようなものが有利かが分っていないことが挙げられる. これが定まらないとリバースの過程をモデル化できないのである.

そこで、本研究では、事務処理分野ではいまだに数多く使われているCOBOLプログラムを入力とし、これを理解し結果を等関係式仕様として逆生成するCOBOL Reverse Engineer for Modules:CORE/Mを作成した. 等関係式仕様が得られれば、これをEOSに入力し設計を行わせてモジュール仕様を生成し[10]、さらにこれをSPACEに入力しCOBOLプログラムを生成させることができる[7].

以下では、2節においてCORE/Mへの入力と出力が何であるかを議論し、3節においてプログラムの解析と等関係式仕様の生成方式を説明し、むすびにおいては、CORE/Mを事例を用いて評価しその有効性を議論する.

2. CORE/Mの入力と出力

2. 1 プログラム構造とブロックの型

一般にどんな計算もそうであるが、ブロック内で実行される計算は同一対象、ファイル処理では特に同一実体あるいは同一関連、に対するものである. 一般のプログラムでは実行中のある瞬間においてプログラムが処理対象としている実体や関連はその属性項目の値によって識別される. この項目はプログラム中では作業領域の項目として定義される. ワーニエ[16]はこれを照合基準と呼んでいる. 一方、プログラムはデータを処理する分けで、データの方にもどの実体の属性を表しているかを識別する項目がある. この項目は、対象データがファイル中のレコードである場合はそのソートキー項目であり、対象データが作業領域中の配列である場合はその添え字項目である. ワーニエはこれらを識別基準と呼んでいる. 一般に計算では識別基準を照合基準や特定の値と比較することによって、現在入力中のレコードが処理対象であるのかどうかを決定する. なお、識別基準と照合基準をまとめてキー項目と呼ぶ.

一方、BohmとJacopiniの構造化定理によれば、適正なプログラムであればその論理構造は、順次、選択、繰返しの3つの構造単位（ブロックと呼ぶ）の組合わせによって記述できるとされている[3]. 本研究では、ファイル処

理プログラム中のブロックの実行条件を、照合基準や識別基準を用いて表すことにより、図1に示すように以下の4つに分ける．
①TYPE1:識別基準を照合基準と比較する条件による選択構造．すなわち、ファイルから読み込んだレコードの識別基準を照合基準と較べその値が一致するかしないか、すなわち照合するかしないかによって、そのファイルに処理対象のレコードがあるかないかを認識し、処理を分ける選択構造．
②TYPE2:識別基準がある値の時、処理を行う選択構造．
③TYPE3:識別基準を照合基準と比較する条件による繰返し構造．すなわち、ファイルからのレコード入力毎に識別基準を照合基準と較べ、その値が一定の間処理を繰返し行う構造．
④TYPE4:識別基準の値がある範囲の間処理を繰返す構造．

ただし、繰返し条件も選択条件も持たないブロックはTYPE0とする．

処理＼条件	キー項目比較	数値比較
順次処理	TYPE0	
選択処理	TYPE1	TYPE2
繰返し処理	TYPE3	TYPE4

図1.ブロックの分類

2．2　入力となるCOBOLプログラムに対する制約

CORE/Mが入力とするのは、バッチ型のファイル処理を行うCOBOLプログラムである．ここで、ファイルとは実体や関連の属性をレコード単位で記述したものである．現在のCORE/Mはまだプロトタイプである．従って、理解できるのは、一言で言えば、ワーニエ法やジャクソン法などに従って構造化された綺麗な制御構造をしているプログラムのみである．具体的に、これらの制約をまとめると以下の様になる．
1) 処理対象ファイルは実体や関連を表す．
2) レコードの左側に配置されているキー項目の順に(次節で定義する識別子項目の≦順に)ファイルは昇順にソートされている．
3) 複数項目比較による照合処理を含まない．必要なら複合キーを設定し照合する．
4) キー項目は、各ファイル内で同一名を使用する．
5) 全てのファイルは、出力ファイルも含めて、識別基準を持つ．
6) オンライン処理命令を含まない．
7) 変数は、すべてグローバル変数とする．
8) CALL文のような引数つき呼び出しは行わない．
9) 選択はIF文で行う．
10) 繰り返しはPERFORM UNTILで行う．
11) 作業領域の各項目は必ず初期化されている．
12) 変数を別変数に代入し直して制御するなど暗号的プログラミングを含まない．
13) 数値条件による繰り返し処理(TYPE4)を含まない．

制約1～6はバッチ型のファイル処理プログラムを処理対象にすることからきている．制約7～12は分析を効率的に行うために設けた．制約9は解析結果を表す等関係式仕様の拡張が必要とされるため現在のCORE/Mでは見送ることにした．なお、我々は処理構造からプログラムを理解する課程においてデータモデルを推定しているので、以下のような制約は設けない．
14) 各項目は唯1つの実体の属性である．すなわち、同一項目名が異なる実体の属性として現れることはない．ただし、制約14があると理解における推定の検証が行えて有効である．

2．3　出力としての等関係式仕様

CORE/Mはプログラム理解の結果を等関係式仕様(別名EOS仕様)として生成する[9,10]．等関係式仕様は、ファイル処理問題における実体や関連の属性項目間の関係と出力・更新要求を等関係式という数式の集合で表したもので、要求を非手続的に表現することができる．等関係式とは、左辺は単一の項目からなり、右辺はそれを定義する項目からなる算術式や条件式である．ここでは、項目は"."を用いてそれが属する実体や関連の識別子で修飾され、"_"を用いてその項目が存在するファイルの識別子で修飾されている．従って、項目の実体や関連の修飾が明示され、等関係式が項目の意味を明確に定義するのに役立っている．

制約1により、ファイルには、実体関連モデルで言うところのいわゆる実体や関連の属性情報が格納されている．ファイル処理とは、これらのファイルから特定の実体の属性値を求めて、出力したり更新することである．一般に、属性項目の値は特定のドメインDの要素dであり、ファイルFは同一の型のレコードfのリストであり、項目Xはファイルからドメインへの写像X:F→Dであり、レコードはいくつかの項目Xiの値xi(=Xi(f))の組<x1,x2,…,xn>であると考える．ファイルFの各レコードfがどの実体を表しているかを識別できる項目Iを実体識別子と呼ぶ．またどの実体間の関連を表しているかを識別できる項目の組<I1,...,In>を関連識別子と呼ぶ．この時、実体Iあるいは関連<I1,...,In>と書く．また、レコードを物理的に識別する仮想の識別子を考え@で表わす．ファイルFが識別子I2で識別子I1より先にソートされているなら、I1≦I2とする．また一般に、任意の識別子I,I'に対して@≦I,<I,I'>≦I,<I,I'>≦I'とする．

ファイルFにおいて項目Xの値が等しいという同値関係によって分割された個々のレコードリストをグループと呼び、特にX(f)=xのものをgroup_F(X,x)と書く．
group_F(X,x)={f|∃f ∈F,X(f)=x,}…(1)

項目XがファイルFに存在することを示すためにX_F(CORE/Mの出力においては添え字が使えないのでX_Fと表す)と書く．この時、添え字Fを項目Xのファイル修飾子と呼ぶ．ファイルFが実体を表し、その識別子がI_Fで項目X_Fが実体I_Fの属性である時、実体I_Fの属性X_Fの値リストを$X_{F.}I_F$と書く．この時、項目X_Fは識別子I_Fで修飾されているという．しかし、1つの実体の属性が複数のファイルに別れて存在する時や、新しいファイルを作成する時には、属性のファイル修飾子とそれを修飾する識別子のファイル修飾子とが異なることもある．この"."修飾は関連ファイルを介在することによって多段にすることができる．すなわち、計算対象となる実体I1を表すファイルGと属性Xを持つ実体I0を表すファイルF間をいくつかの関連ファイルRによって結合することによって、属性Xが実体I1とどの様な従属関係にあるかを"."修飾を複数回用いて指定できる．例えば、実体I1とRで表される関連を持つ実体I0の属性は$X_{F.}I0_{G.}<I1,I0>_{R.}I1_S$と表される．形式的には、属性Xの値を求める元になるレコードリストは、実体I1の各値i1に対して識別子<I1,I0>を持つ関連ファイルR内のI1(r)=i1なるレコードrと実体ファイルG内のI0(g)=I0(r)なるレコードgを経由して関係して

るグループgroup_F(I0,I0(g))で与えられる．すなわち、$X_F.I0_G.<I1,I0>_R.I1_S=\{X(f)|\exists s\in S,\exists r\in R,\exists g\in G,\exists f\in F,I1(r)=I1(s),I0(g)=I0(r),I0(f)=I0(g)\}$…(2)

ファイル処理に対する本研究の根底となる考え方は、『更新／出力ファイルにおける実体の属性項目の値はこの実体やこれと関連する他の実体の属性から求められる』ということである．具体的には、更新／出力ファイルLあるいは作業領域Lにおいて、『識別子I0で表わされる実体$I0_F$の属性$X_L.I0_F$の値は、他のファイルG0におけるこの実体$I0_{F0}$の別の属性$X0_{G0}.I0_{F0}$や、この実体$I0_{F1}$と関連Rを持つ他のファイルG1における実体$I1_{F1}$の属性$X1_{G1}.I1_{F1}.<I1',I0'>_R.I0_{F1}$や、ファイルF2が実体I2を表すが同時に実体I0との関連も表すファイルである時には、実体I2の属性$X2_{F2}.<I0,I2>_{F2}$などに、関数gを適用させて求まる』ということである．

$X_L.I0_F=g(X0_{G0}.I0_{F0},X1_{G1}.I1_{F1}.<I1',I0'>_R.I0_{F1},X2_{F2}.<I0,I2>_{F2},...)$…(3)

なお、ここで同じ実体を表す識別子Iが、ファイルによって異なるかもしれないので$I1_{F1}$と$I1'_R$の様に表したが、以後は簡単化のため同一記号で表す．この式(3)を等関係式と呼び、左辺の項目の意味を右辺の条件式を含む算術式で定義している[9]．なお、ファイル修飾子として2つのファイルF1とF2に供にレコードが存在する実体Iからなる以下のような照合ファイルF1&F2[I]をもちいることがある．

$F1\&F2[I]=\{f1\hat{}f2|\exists f2\in F2,\exists f1\in F1,I(f1)=I(f2)\}$…(4)

等関係式には繰り返し式は含まれていない．これは等関係式の各項目が既に(1)に示すように集合(ﾘｽﾄ)に対する式であり、すでに具体的に計算する際には繰り返し処理を誘発するからである．

3　CORE/Mにおける解析と生成の手順

我々は、COBOLプログラム中の計算式は項目の定義そのものであると考え、後はそこに含まれる各項目がどの実体のもので他の実体とどのような関連で結合されているかを明確にすればよいと考えた．ただし、この際ProustやLAURAのように、特定のクリシェの展開であると考えられる集計処理などは、その意味を表すSUMなどの関数に置き換えることにした．

一般に、プログラマは同一ブロック内には特定の条件を満たす実体に対する計算式のみを配置する．バッチ型のファイル処理プログラマでは、この条件はキー項目の比較条件のことがほとんどである．等関係式では、この条件は項目に対する"."を使った実体識別子修飾と"_"を使ったファイル修飾で表す．従って、等関係式を生成するためには、プログラムをブロックに分けそのブロックの持つこれらの条件を見出すことが必要である．

ただし、CORE/Mが生成する等関係式は全て式(3)において、項目の"."修飾が１回のものである．これは１段修飾と言われEOS/M[9]がモジュール仕様を自動設計するために入力とするプログラム仕様を記述するものである．式(3)のような多段修飾を含むものはEOS/P[10]によってプロセス設計を行う必要があるジョブ仕様を記述する際に用いる．CORE/Mでは１つのプログラムを入力するもので、プログラム間のファイルを経由したプロセスフローの解析は行わない．言い換えれば、CORE/MはEOS/MとSPACEが行う自動設計と自動プログラミングの逆を行うものである．この意味で、本研究のCOREには/Mの修飾語を付けている．

このようなことから、CORE/Mの処理ステップを図２に示すように6つの解析ステップと5つの生成ステップに分けた．以下では、これらの各過程を具体的に説明する．なお、解説にあっては事例として図３に示すような在庫管理問題を用いた．この問題は、在庫量の集計という繰り返し構造を持ち、照合処理と集計処理の2つ の処理を含むものである．この問題に対するCOBOLプログラムを学生に作らせたところ図４に示すような約230行からなるものが得られた．これをCORE/Mへの事例としての入力とする．

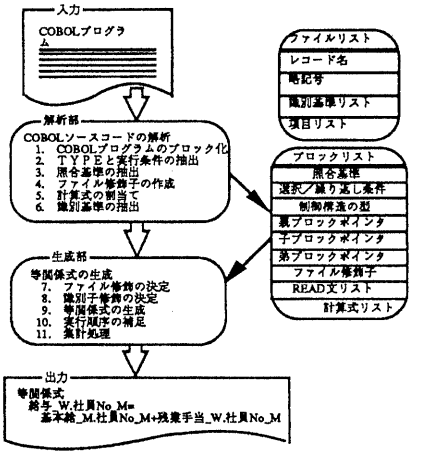

図2.COREおける処理の流れ

[処理概要]

入出庫ファイルと削除ファイルにより、在庫マスタを更新る。同時に発注ファイル及びエラーリストを出力する。

[プロセスフロー]

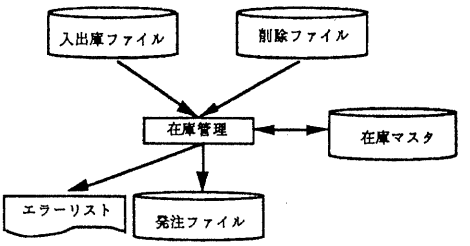

図3.　例題としての在庫管理問題

[処理条件]

1．各製品ごとに入出庫合計（符号付き）を求める。
　ただし入出庫ファイルの区分は、"I"（入庫）及び"O"（出庫）。
2．入出庫がある場合、新在庫量を次の方法で求め、正の時在庫マスタを更新する。
　新在庫量＝旧在庫量＋入出庫量合計
3．０≦新在庫量＜最低在庫量の場合　　発注量＝適正在庫量－新在庫量
　０＞新在庫量の場合　　発注量＝適正在庫量
　として、発注レコードを出力する。
4．在庫あり、入出庫なしで削除がある時は、在庫レコードを削除する。
5．エラーの種類は次の通り。
　●在庫がない時－－－＞"ZAIKO NASI"
　●在庫があり新在庫が負の時－－－＞"MINUS"
　●在庫、入出庫があり、削除がある時－－－＞"SAKUJO"

```
PROGRAM-ID.    ZAIKO-PROBLEM.

ENVIRONMENT   DIVISION.
INPUT-OUTPUT  SECTION.
FILE-CONTROL.
  SELECT 在庫マスタ       ASSIGN TO "zaiko.mst".
  SELECT 入出庫ファイル   ASSIGN TO "zaiko.trn".
  SELECT 削除ファイル     ASSIGN TO "zaiko.skj".
  SELECT 新在庫マスタ     ASSIGN TO "zaiko.new".
  SELECT 発注ファイル     ASSIGN TO "zaiko.htu".
  SELECT エラーリスト     ASSIGN TO "zaiko.err".

DATA DIVISION.
FILE SECTION.
FD 在庫マスタ.
01 MST-RECORD.
  02 製品NO      PIC 9(3).
  02 在庫量      PIC 9(7).
  02 適正在庫    PIC 9(7).
  02 最少在庫    PIC 9(7).

FD 入出庫ファイル.
01 TRN-RECORD.
  02 製品NO      PIC 9(3).
  02 区分        PIC X(1).
  02 数量        PIC 9(7).

FD 削除ファイル.
01 SAK-RECORD.
  02 製品NO      PIC 9(3).

FD 新在庫マスタ.
01 NEW-RECORD.
  02 製品NO      PIC 9(3).
  02 在庫量      PIC 9(7).
  02 適正在庫    PIC 9(7).
  02 最少在庫    PIC 9(7).

FD 発注ファイル.
01 HTU-RECORD.
  02 製品NO      PIC 9(3).
  02 発注量      PIC 9(7).

FD エラーリスト.
01 ERR-RECORD.
  02 製品NO      PIC 9(3).
  02 エラーメッセージ  PIC X(50).

WORKING-STORAGE SECTION.
01 作業領域.
  02 製品NO      PIC 9(3).
  02 在庫量      PIC S9(7).

PROCEDURE DIVISION.
PROLOGUE.
  OPEN INPUT     在庫マスタ.
  OPEN INPUT     入出庫ファイル.
  OPEN INPUT     削除ファイル.
  OPEN OUTPUT    新在庫マスタ.
  OPEN OUTPUT    エラーリスト.
  OPEN OUTPUT    発注ファイル.

  READ 在庫マスタ
    AT END MOVE HIGH-VALUE TO 製品NO OF 在庫
マスタ.
  READ 入出庫ファイル
    AT END MOVE HIGH-VALUE TO 製品NO OF 入出
庫ファイル.
  READ 削除ファイル
    AT END MOVE HIGH-VALUE TO 製品NO OF 削除
ファイル.

  PERFORM 製品処理
    UNTIL ((製品NO OF 在庫マスタ = HIGH-VALUE)
AND
          (製品NO OF 入出庫ファイル = HIGH-VALUE)
AND
          (製品NO OF 削除ファイル = HIGH-VALUE)).

EPILOGUE.
  CLOSE 在庫マスタ.
  CLOSE 新在庫マスタ.
  CLOSE 入出庫ファイル.
  CLOSE 削除ファイル.
  CLOSE エラーリスト.
  CLOSE 発注ファイル.

  STOP RUN.

製品処理.
  IF (製品NO OF 在庫マスタ < 製品NO OF 入出庫ファ
イル)
    THEN MOVE 製品NO OF 在庫マスタ TO 製品NO
OF 作業領域
    ELSE MOVE 製品NO OF 入出庫ファイル TO 製品
NO OF 作業領域.
  IF (製品NO OF 削除ファイル < 製品NO OF 作業領
域)
    THEN MOVE 製品NO OF 削除ファイル TO 製品NO
OF 作業領域.
  IF (製品NO OF 在庫マスタ = 製品NO OF 作業領域)
    THEN PERFORM マスタ製品有処理
    ELSE PERFORM マスタ製品無処理.

マスタ製品有処理.
  MOVE 製品NO OF 在庫マスタ TO 製品NO OF 新在
庫マスタ.
  MOVE 適正在庫 OF 在庫マスタ TO 適正在庫 OF 新
在庫マスタ.
  MOVE 最少在庫 OF 在庫マスタ TO 最少在庫 OF 新
在庫マスタ.
  IF (製品NO OF 入出庫ファイル = 製品NO OF 作業領
域)
    THEN PERFORM トランス製品有処理
    ELSE PERFORM トランス製品無処理.
  READ 在庫マスタ AT END MOVE HIGH-VALUE TO
製品NO OF 在庫マスタ.

マスタ製品無処理.
  IF (製品NO OF 作業領域 = 製品NO OF 入出庫ファイ
ル)
    PERFORM トランス在庫無
    UNTIL ((製品NO OF 入出庫ファイル = HIGH-
VALUE) OR
          (製品NO OF 入出庫ファイル NOT = 製品NO OF
作業領域)).
  IF (製品NO OF 削除ファイル = 製品NO OF 作業領
域)
    PERFORM 削除在庫無.

トランス製品有処理.
  MOVE 在庫量 OF 在庫マスタ TO 在庫量 OF 作業領
域.
  PERFORM 在庫量計算処理
    UNTIL ((製品NO OF 入出庫ファイル = HIGH-
VALUE) OR
          (製品NO OF 入出庫ファイル NOT = 製品NO OF
作業領域)).
  PERFORM 在庫量検査.
  IF (製品NO OF 削除ファイル = 製品NO OF 作業領
域)
    PERFORM 削除エラー処理.
  WRITE NEW-RECORD OF 新在庫マスタ.

トランス製品無処理.
  IF (製品NO OF 削除ファイル = 製品NO OF 作業領
域)
    THEN READ 削除ファイル AT END MOVE HIGH-
VALUE TO 製品NO OF 削除ファイル
    ELSE PERFORM 削除製品無.

トランス在庫無.
  MOVE 製品NO OF 入出庫ファイル TO 製品NO OF
エラーリスト.
  MOVE 'ZAIKO NASI' TO エラーメッセージ OF エラー
リスト.
  WRITE ERR-RECORD OF エラーリスト.
  READ 入出庫ファイル AT END MOVE HIGH-
VALUE TO 製品NO OF 入出庫ファイル.

削除在庫無.
  MOVE 製品NO OF 作業領域 TO 製品NO OF エラー
リスト.
  MOVE 'ZAIKO NASI' TO エラーメッセージ OF エラー
リスト.
  WRITE ERR-RECORD OF エラーリスト.
  READ 削除ファイル AT END MOVE HIGH-VALUE
TO 製品NO OF 削除ファイル.

在庫量計算処理.
  IF (区分 OF 入出庫ファイル = 'I')
    THEN COMPUTE 在庫量 OF 作業領域 = 在庫量 OF
作業領域 + 数量 OF 入出庫ファイル
    ELSE COMPUTE 在庫量 OF 作業領域 = 在庫量 OF
作業領域 - 数量 OF 入出庫ファイル.
  READ 入出庫ファイル AT END MOVE HIGH-
VALUE TO 製品NO OF 入出庫ファイル.

削除エラー処理.
  MOVE 製品NO OF 削除ファイル TO 製品NO OF エ
ラーリスト.
  MOVE 'SAKUJO' TO エラーメッセージ OF エラーリ
スト.
  WRITE ERR-RECORD OF エラーリスト.
  READ 削除ファイル AT END MOVE HIGH-VALUE
TO 製品NO OF 削除ファイル.

削除製品無.
  MOVE 在庫量 OF 在庫マスタ TO 在庫量 OF 新在庫
マスタ.
  WRITE NEW-RECORD OF 新在庫マスタ.

在庫量検査.
  IF (在庫量 OF 作業領域 < 0)
    THEN PERFORM 在庫量負処理
    ELSE MOVE 在庫量 OF 作業領域 TO 在庫量 OF 新
在庫マスタ.
  IF ((在庫量 OF 作業領域 > 0) AND (在庫量 OF 作業領
域 < 最少在庫 OF 在庫マスタ))
    THEN PERFORM 発注処理.

在庫量負処理.
  MOVE 製品NO OF 在庫マスタ TO 製品NO OF エラー
リスト.
  MOVE 'MINUS' TO エラーメッセージ OF エラーリ
スト.
  WRITE ERR-RECORD OF エラーリスト.
  MOVE 製品NO OF 在庫マスタ TO 製品NO OF 発注
ファイル.
  MOVE 適正在庫 OF 在庫マスタ TO 発注量 OF 発注
ファイル.
  WRITE HTU-RECORD OF 発注ファイル.
  MOVE 在庫量 OF 在庫マスタ TO 在庫量 OF 新在庫
マスタ.

発注処理.
  MOVE 製品NO OF 在庫マスタ TO 製品NO OF 発注
ファイル.
  COMPUTE 発注量 OF 発注ファイル = 適正在庫 OF
在庫マスタ - 在庫量 OF 新在庫マスタ.
  WRITE HTU-RECORD OF 発注ファイル.
```

図4. 在庫問題のCOBOLプログラム

3．1　COBOLプログラムのブロック化

特定の条件下の元で実行される一連の文の集合をブロックと呼ぶ．COBOLプログラムでは、節名から次の節名までの間や個々のプロシジャなどがブロックである．

ブロックはPERFORM文で他のブロックを呼出すことができる．この場合、図5に示すように、呼出すブロックを親ブロック、呼出されるブロックを子ブロックという．さらに呼出す順で、先のものを兄ブロック、後のものを弟ブロックという．

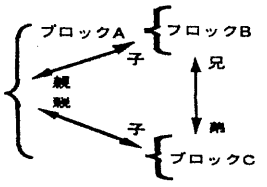

図5. ブロック間の関係

CORE/Mはまず、COBOLプログラムを構文解析し、ブロックに分割し、その間の親子関係、兄弟関係を明らかにする．さらに、ブロック毎にブロックに関する各種の情報を図２に示したようなブロックリストと呼ばれるリストに登録する．このリストには、図１に示したように、ブロックが処理する対象を表す照合基準、ブロックの選択／繰り返し条件、制御構造の型(選択／繰り返し／順序)、親ブロックへのポインタ、子ブロックの先頭へのポインタ、弟ブロックへのポインタ、ファイル修飾子、READ文リスト、ブロック内の計算式リストなどを登録していく．事例に対しては、図６に示すようなブロック構造が生成される．

一方、データ定義部からは、ファイル毎にレコード名や構成項目や識別基準をファイルリストとして抽出する．

TYPE0:①,⑪
TYPE1:③,④,⑤,⑥,⑦,⑨,⑫,⑬,⑭
TYPE2:⑮,⑯,⑰,⑱,⑲
TYPE3:②,⑧,⑩
TYPE4:なし

３．３　ブロックの照合基準の抽出

CORE/Mは、ブロックが処理する対象を表す照合基準を、以下のような3つの条件を満足する項目としてプログラムから抽出する．

①作業領域中(WORKING-STRAGE SECTION)に定義されている．

②ファイル定義中の同名の項目(識別基準)によって初期化されている．(照合基準の設定)

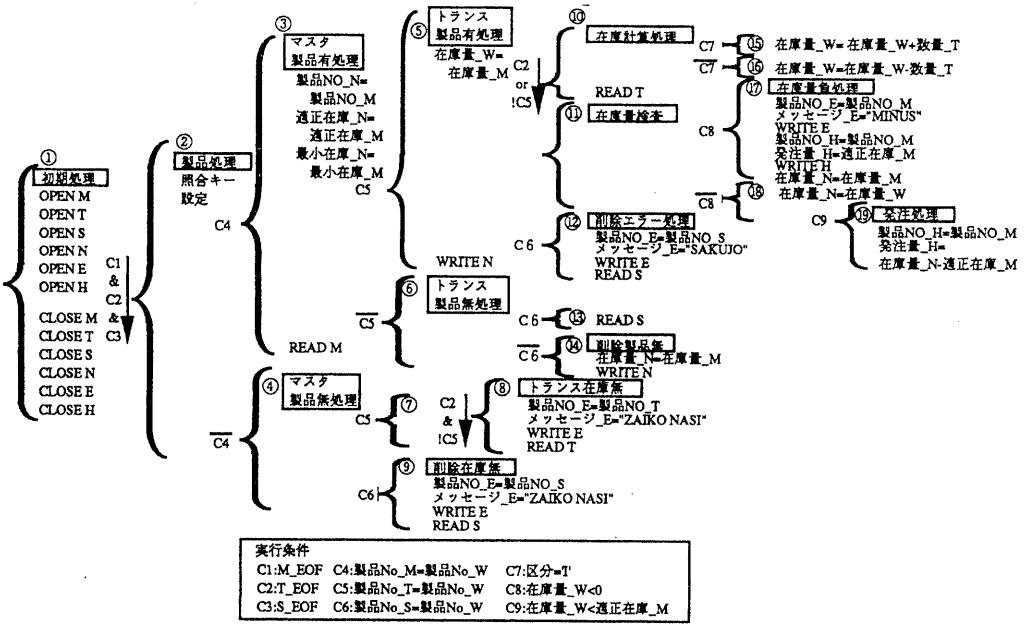

図6. 例題のCOBOLプログラムのブロック構造への展開と実行条件

３．２　ブロックのTYPEと実行条件の抽出

分割されたブロックに対して図１に示した５つのタイプのどれかを決定する．ただし、今回はTYPE4は取り扱わないため、これが検出されたらエラーを表示する．次に、各ブロックの実行条件を抽出し、ブロックリストに登録する．事例に対しては、図６の下段に示したようになる．ここで、C2～C6はキー項目比較であり、C1とC7～C9は数値比較であり、その他はこれらの合成である．なお、C{BはブロックBを条件Cの時実行する意味である.またC↓{BはブロックBが条件Cが満足されるまで繰り返される意味である.この図に示したように、ブロックの実行条件は、ブロックのTYPE毎に、照合基準と識別基準の比較条件であったり項目の数値比較条件であったりする．なお、図中の丸番号はブロック番号で以下の議論で事例として引用する時に用いる．この番号を使って、各ブロックのTYPEを抽出した結果は以下のようになる．

③選択処理や繰返し処理の条件に使用されている．

次に、各ファイルに対して、レコード定義から以下のような識別基準を抽出する．

①ファイルのレコード記述(FILE SECTION)中に定義されている．

②同名の照合基準に代入されている．

③選択処理や繰返し処理の条件に使用されている．

照合基準は、対象条件を構成するものの中でも、現在処理対象の実体や関連を識別する項目という意味で特に重要である．照合基準は、プログラム中のある時点で設定され、次に設定されるまでその照合基準が有効である．言い換えれば、プログラムを図６のようなブロックに展開すると、あるブロックで照合基準が設定されると、次にその子孫ブロックで別の照合基準が設定されるまでの途中のブロックでは、先に設定されたものが有効である．この様にして各ブロック毎に有効な照合基準が一意に決

まる．

さて、照合基準を先の条件で見つけることができれば、この照合基準への識別基準の代入文が設定文となる．しかも、制約10から繰り返しはPERFORM UNTIL型なので、この設定文はその照合基準によって特定されるTYPE3ブロック内の始めの方に記述されている．このように、通常TYPE3ブロックが処理する対象の照合基準を設定する．

しかし、時には図６のブロック⑩のように照合基準を設定しないTYPE3ブロックもある．同一の実体に対する計算であれば繰り返し構造を入れ子にする必要は無いため、この場合にもなにか新しい照合基準が設定されているはずである．この場合考えられる見えない照合基準は唯一個々のレコードを識別する照合基準@である．また、図６のブロック①のようにプログラムの最外側のブロックは全ての実体を対象として含むので、そのことを表すために照合基準として$を用いる．従って、事例に対する各ブロックの照合基準は以下のようになる．

照合基準＝$のブロック:①

照合基準＝製品Ｎｏのブロック:②～⑦,⑨,⑪～⑭,⑰～⑲

照合基準＝@のブロック:⑧,⑩,⑮,⑯

３．４　ブロックのファイル修飾子の作成

このステップでは、いわゆる複数ファイルに対する照合処理を認識する．先のステップでブロックの照合基準が抽出され、処理対象が、製品ＮＯなど、どんな識別子で修飾される実体なのかが明らかになった．次に明らかにすべきは、この実体が削除された製品なのかされなかった製品なのかなど、どんな関連を持つ実体なのかを明らかにすることである．ファイル処理問題では、同一実体の属性が異なるファイルに別れてレコード内に記録されている．従って、個々の実体インスタンスがどんな関連を持つかは、これを表すレコードを各ファイルに持つか持たないかによって表現される．従って、各ブロックが表す照合基準で指定される実体がどんなファイル群に情報を持つものなのかを明らかにする必要がある．このファイル群をファイル修飾子(以下FLで表す)としてブロックリスト内に登録する．

ファイル修飾子の作成を図６の事例を用いて説明する．最外側のブロック(事例ではブロック①)のファイル修飾子FLを、このブロックが複数のファイルに対する入力を伴うかファイル入力を全く伴わないなら空を表すφとし、１つのファイルFからの入力のみならFとする．以下、このブロックの子孫ブロックのファイル修飾子をその実行条件を元に決定していく．より精密には、ブロックがTYPE1の時ファイル修飾子が変更される．そうでない時は、親ブロックのファイル修飾子を継承する(例:ブロック②ではFL=φ)．この変更は、TYPE1ブロックの条件において照合基準(例:製品No_W)と比較されている識別基準(例:製品No_M)が所属するファイルをF(例:M)とすれば、これら基準間の比較条件が＝か≠かに応じて"&F"あるいは"&!F"を親ブロックのファイル修飾子に追加することによって求める(例:ブロック③ではFL=&M)．なお、この議論において照合基準が変化していくのに(事例では$→製品No→@)、ファイル修飾子が子ブロックに継承されることが不思議に思えるかもしれないが、これらの識別子が≦順序で線形ソートされていることを考慮すれば、理解できる．

ただしここで問題が１つある．プログラムを分析した結果、ファイルに対して初期読み込み(例えばブロック①のREAD M、READ T、READ S)があれば、基本的にこれら３つのファイルに対して照合処理が行われていると考えられる．ファイルの照合処理において、ワーニエ法やジャクソン法に従って各ファイルからの入力レコードの識別基準の最小値を照合基準に設定する方式を採っていれば、照合基準がどのファイルの識別基準とも一致しないという状況は発生しない．言い換えれば、例えば２つのファイルX,Yにおいて、一方のファイルXに当該レコードがないということは他方に当該レコードがあるということと同値である．従って、ファイル修飾子!XはY&!Xと考えてよい．この推定は、このブロック内にファイルYに対する読み込みがあれば、より確実なものとなる．これは、プログラマがこの同値関係を配慮して、照合基準とファイルYの識別基準のTYPE1ブロックを省略したためと考えられる．EOSでは肯定的に条件を明示化することを好むので、このようなファイル修飾子!XをY&!Xとする．これらを考慮すると事例に対する各ブロックのファイル修飾子は以下のようになる．なお、φになんらかのファイルが&で追加されればφは取り除く．また、φはプログラムでは処理できないので-とする．

$に対するファイル修飾子＝-のブロック:①

製品Ｎｏに対するファイル修飾子＝-のブロック:②

製品Ｎｏに対するファイル修飾子＝Mのブロック:③

製品Ｎｏに対するファイル修飾子＝!Mのブロック:④

製品Ｎｏに対するファイル修飾子＝M&T[製品No]のブロック:⑤⑪⑫⑱⑲

製品Ｎｏに対するファイル修飾子＝M&!T[製品No]のブロック:⑥

製品Ｎｏに対するファイル修飾子＝!M&T[製品No]のブロック:⑦

製品Ｎｏに対するファイル修飾子＝!M&S[製品No]のブロック:⑨

製品Ｎｏに対するファイル修飾子＝M&T&S[製品No]のブロック:⑫

製品Ｎｏに対するファイル修飾子＝M&!T&Sのブロック:⑬

製品Ｎｏに対するファイル修飾子＝M&!T&S[製品No]のブロック:⑭

@に対するファイル修飾子＝!M&T[製品No]のブロック:⑧

@に対するファイル修飾子＝M&T[製品No]のブロック:⑩⑮⑯

３．５　ブロックへの計算式の割当て

各ブロックに含まれる処理文から計算式を抽出する．ここで抽出すべき計算式とはMOVE文、COMPUTE文、WRITE文の3つの文である．ブロック内にあるREAD文は等関係式には変換しないので、計算式としては抽出しない．READ文はどのファイルに対するREADなのかを含めて、ブロックリストのREAD欄に登録する．

なお、MOVE文のTO句の後の項目、COMPUTE文の左辺項目、WRITE文の出力項目、TYPE2ブロックの条件を構成する項目、をまとめて左辺項目と言う．またMOVE文のTO句の前の項目やCOMPUTE文の右辺項目をまとめて右辺項目と言う．

３．６　レコードからの識別基準の抽出

各ファイルのレコードは実体か関連を表す．実体を表す時はその実体の識別子項目(識別基準)をレコード内に含む．関連を表す時はそれが結合する２つの実体の識別子項目を含む．従って、識別基準が求まれば、これを含むレコードを持つか持たないかによって、各ファイルがどの実体あるいはどの実体とどの実体間の関連かを明らかにできる．この結果をファイル毎にファイルリストに記録しておく．

この識別基準の抽出が終わった段階で、ファイルリストからは、等関係式仕様のデータ記述部として、ファイル毎のレコードの構成項目を図７の[3]の様に生成する．また、制約2から識別子は≦の降順にレコードに定義さ

れていることから、その順序関係を図7の[2]の様に生成する．

3．7　項目のファイル修飾の決定

COBOLプログラム中の全ての項目Dは、どのファイルF(作業領域も含めて)に所属する項目かを一意的に表すために、必要ならOF句を用いてファイルやレコードによって修飾されている．OF修飾を持たない場合は、FILE SECTION、WORKING-STRAGE SECTIONを検索することによって、この項目を持つファイルFを一意的に定めることができる．なお、WORKING-STRAGE SECTION内で見つかった場合、作業領域を表すWを所属ファイルとする．この結果、項目Dをそれが所属するファイルFをファイル修飾子として修飾し、D_Fに変換する．なお、定数に対してはファイル修飾子は付けない．

ただし、ファイルFの01レベルのレコード定義に2種類以上(例:R1とR2)あり、プログラム中でD OF R1 (OF F)やD OF R2 (OF F)があるときは、D OF R1やD OF R2が一意的な項目名であると考えD～OF～R1_FやD～OF～R2_Fに置き換える．

3．8　項目の識別子修飾の決定

一般に、ブロックの照合基準がIでファイル修飾子がFLなら、このブロックが処理対象とする実体は、識別子I_FLで指定される．すなわち、この実体は、識別子Iで識別され、そのレコードがファイル修飾子FLに&付きで現れるファイルに含まれかつ&!付きで現れるファイルには含まれない実体である．このブロックに現れる式の全ての項目D_Fがこのような実体の属性であれば、単にD_FをI_FLで修飾しD_F.I_FLと変換すればよい．しかし、実際はそう簡単ではなく集計計算などに見られるように、加算項目は当該ブロックの照合基準で識別される実体の属性であるけれども、合計項目は当該ブロックの照合基準より大きな照合基準で識別される実体の属性であることが普通である．従って我々は項目の識別子による修飾をプログラムの制御構造から推定することにし、具体的には以下のように考えた．

CORE/Mでは、項目D_Fを修飾する識別子の決定は、項目D_Fが入力ファイルの項目、出力ファイルの項目、作業領域の項目によって異なる．なお、ファイルFが入力ファイルか出力ファイルかは、read Fかwrite Fのどちらがプログラムに現れているかによる．またrewrite Fの場合は入力ファイルとして扱う．

ファイルFが入力ファイルの時は、項目D_Fが現れているブロックあるいはその先行ブロック(この項目を含む式よりも前にその全体あるいは一部が実行されるブロック)の内で、そのファイル修飾子FL中にFが!なしで現れる直近のブロックを見付け、その照合基準をIまたファイル修飾子をFLとして、項目D_FをD_F.I_FLとする．なお、通常は項目D_Fが現れているブロックであることが多い．例えば、ブロック③の適正在庫_Mに対しては、ブロック③そのもののファイル修飾子にMがあるので、適正在庫_M.製品NO_Mとなる．

ファイルFが出力ファイルの時は、項目D_Fが現れるブロックあるいはその後続ブロック(この項目を含む式よりも後にその全体あるいは一部が実行されるブロック)の内で、その項目を含むレコードを出力している直近のブロックを見付け、その照合基準をIまたファイル修飾子をFLとして、項目D_FをD_F.I_FLとする．例えば、ブロック⑬の在庫量_Nに対しては、ブロック⑬そのものがNへのWRITEを持つので、在庫量_N.製品NO_M&!T&!Sとなる．

最後に、作業領域内の項目D_WにおけるWは、どの照合基準のファイル修飾子FLにも含まれないし、出力もされない．しかし制約11より作業領域の項目は正しく初期化されているはずである．D_Wが現れるブロックをBまたそのファイル修飾子をFLとする．ブロックBあるいはその先行ブロックの内で、項目D_Wに定数を代入している式を含むブロックがあればそのようなもので直近のブロックB'の照合基準I'を用いて、なければ項目D_Wを左辺に持つ式を含むブロックがあればそのようなもので直近のブロックB'の照合基準I'を用いて、それもなければD_Wが現れるブロックBをブロックB'としてその照合基準I'を用いて、D_WをD_W.I'_FLに変換する．なお、ここでFLはブロックBのファイル修飾子である．例えば、ブロック⑮の在庫量_Wに対しては、先行するブロック⑤にそれを左辺として持つ式があるので、在庫量_W.製品NO_M&Tとなる．

なお、後出のFOLLOW文内の項目の修飾は出力ファイル内の項目と同様にする．また、定数Cは、それを含むブロックBの識別子Iとファイル修飾子FLを用いて、C.I_FLに変換する．

また、いずれの場合もファイル修飾子が照合ファイルになる時は、その照合キーは通常ブロックの照合基準であるが、これが@の時は、当該ブロックを含む直前のTYPE3ブロックの照合基準とする（例:ブロック⑮).ただし、ファイル修飾子中の照合キーがそれが修飾する識別子に等しい場合は繁雑を減らすために省略する（例:在庫量_M.製品No_M&T[製品No]→在庫量_M.製品No_M&T）．

3．9　計算式から等関係式の生成

以上の変換の結果、ブロックに割り当てられた計算式の項目は全て識別子とファイル修飾子で修飾された．最後に、これらの修飾子を伴って、計算式を等関係式に変換する．ただし、この際TYPE2ブロック内の計算式に対しては、それら全体をIF(C){と}で取り囲む．ここで、CはTYPE2ブロックの条件式を識別子やファイル修飾子で修飾した結果である．なお、この条件項目の修飾に際してはこれらが特定する(特定されるではない)ブロックの照合基準やファイル修飾子を使う．またもし、TYPE2ブロックが連続して入れ子になっているなら、これらの条件を&で結合して1つの条件にする．言い換えれば、IF形式の等関係式は入れ子にしない．なおこの際、ELSEで結合される条件はその前のIF条件を否定したものを&で結合していく必要がある．

3．10　実行順序の補足

EOSでは、右辺項目はそれが左辺項目として登場した後で計算するという規則に従って、非手続き仕様から計算式の実行順序を決定している．従って、ヘッダレコード間の物理的な出力順序や、レコード出力はレコード構成項目の計算が終了した後で行うなどという計算順序もこの方式に従って指定しなければならない．しかしこれらに関与する式では右辺や左辺の区別がないので、恣意的にこの順序を明示化する計算式が必要となる．EOSではこれをFOLLOW文を用いて行う．

具体的には、各WRITE(R)を生成する前に、レコードRを構成する下位レベルの項目D1,D2,...毎に、R=FOLLOW(D1^D2^...)なる等関係式を生成する．また連続するWRITE(R1)とWRITE(R2)に対しては、R2=FOLLWO(R1)を生成する．

例えば、ブロック⑤のWRITE(N)に対して、以下の式が生成される.

```
新マスタレコード_N.製品No_M&T= FOLLOW(製品No_N.製品No_M&T^在庫量_N.製品No_M&T^適正在庫_N.製品No_M&T^最少在庫_N.製品No_M&T);
```

3．11　クリシェの意味変換

TYPE3ブロックにCOMPUTE　A=A+Bがあり、その先祖ブロックでこのブロックに先行するところにMOVE 0 TO Aあるいは COMPUTE　A=0があれば、これらは1つの集計処理を表すクリシェを構成するとみなすことができる．この場合は、これら2式を1つのA=SUM(B)に変換する．なお、この様な関数形への意味的変換は、SUM以外にも、グループの最大値MAX、最小値MIN、先頭値1STなどのクリシェに対して行う.

以上3ステップの結果、図7の[1]の機能記述が得られる.

```
[1]機能記述
③
製品NO_N.製品NO_M = 製品NO_M.製品NO_M;
適正在庫_N.製品NO_M = 適正在庫_M.製品NO_M;
最少在庫_N.製品NO_M = 最少在庫_M.製品NO_M;
⑤
在庫量_W.製品NO_M&T = 在庫量_M.製品NO_M&T;
WRITE (新マスタレコード_N.製品NO_M&T);
新マスタレコード_N.製品NO_M&T = FOLLOW(製品NO_N.製品NO_M&T^在庫量_N.製品NO_MT^適正在庫_N.製品NO_M&T^最少在庫_N.製品NO_M&T);
⑧
製品NO_E.@_!M&T[製品NO] = 製品NO_T.@_!M&T[製品NO];
メッセージ_E.@_!M&T[製品NO] = "ZAIKO NASI".@_!M&T[製品NO];
WRITE (エラーレコード_E.@_!M&T[製品NO]);
エラーレコード_E.@_!M&T[製品NO] = FOLLOW(製品NO_E.@_!M&T[製品NO]^メッセージ_E.@_!M&T[製品NO]);
⑨
製品NO_E.製品NO_!M&S = 製品NO_W.製品NO_!M&S;
メッセージ_E.製品NO_!M&S = "ZAIKO NASI".製品NO_!M&S;
WRITE (エラーレコード_E.製品NO_!M&S);
エラーレコード_E.製品NO_!M&S = FOLLOW(製品NO_E.製品NO_!M&S^メッセージ_E.製品NO_!M&S);
⑫
製品NO_E.製品NO_M&T&S = 製品NO_S.製品NO_M&T&S;
メッセージ_E.製品NO_M&T&S = "SAKUJO".製品NO_M&T&S;
WRITE (エラーレコード_E.製品NO_M&T&S);
エラーレコード_E.製品NO_M&T&S = FOLLOW(製品NO_E.製品NO_M&T&S^メッセージ_E.製品NO_M&T&S);
⑭
在庫量_N.製品NO_M&!T&!S = 在庫量_M.製品NO_M&!T&!S;
WRITE (新マスタレコード_N.製品NO_M&!T&!S);
新マスタレコード_N.製品NO_M&!T&!S = FOLLOW(製品NO_N.製品NO_M&!T&!S^在庫量_N.製品NO_M&!T&!S^適正在庫_N.製品NO_M&!T&!S^最少在庫_N.製品NO_M&!T&!S);
⑮
IF (区分_T.@_M&T[製品NO] = T'.@_M&T[製品NO]) 在庫量_W.製品NO_M&T = 在庫量_W.製品NO_M&T + 数量_T.@_M&T[製品NO]
⑯
IF NOT (区分_T.@_M&T[製品NO] = T'.@_M&T[製品NO]) 在庫量_W.製品NO_M&T = 在庫量_W.製品NO_M&T - 数量_T.@_M&T[製品NO];
⑰
IF (在庫量_W.製品NO_M&T < 0.製品NO_M&T) {
 製品NO_E.製品NO_M&T = 製品NO_M.製品NO_M&T;
 メッセージ_E.製品NO_M&T = "MINUS".製品NO_M&T;
 WRITE (エラーレコード_E.製品NO_M&T);
 エラーレコード_E.製品NO_M&T = FOLLOW(製品NO_E.製品NO_M&T^メッセージ_E.製品NO_M&T)
 製品NO_H.製品NO_M&T = 製品NO_M.製品NO_M&T;
 発注量_H.製品NO_M&T = 適正在庫_M.製品NO_M&T;
 WRITE (発注レコード_H.製品NO_M&T);
 発注レコード_H.製品NO_M&T = FOLLOW(製品NO_H.製品NO_M&T^発注量_H.製品NO_M&T);
 在庫量_N.製品NO_M&T = 在庫量_M.製品NO_M&T;
}
⑱
在庫量_N.製品NO_M&T = 在庫量_W.製品NO_M&T;
⑲
IF ((在庫量_W.製品NO_M&T > 0.製品NO_M&T)
 AND (在庫量_W.製品NO_M&T < 最少在庫_M.製品NO_M&T))
{
 製品NO_H.製品NO_M&T = 製品NO_M.製品NO_M&T;
 発注量_H.製品NO_M&T = 適正在庫_M.製品NO_M&T - 在庫量_N.製品NO_M&T;
 WRITE (発注レコード_H.製品NO_M&T);
 発注レコード_H.製品NO_M&T = FOLLOW(製品NO_H.製品NO_M&T^発注量_H.製品NO_M&T);
}

[2]識別子の順序
@<製品NO

[3]レコード記述
在庫マスタ;M;*製品NO;在庫量;適性在庫量;最少在庫量;;;
新在庫マスタ;N;*製品NO;在庫量;適性在庫量;最少在庫量;;;
入出庫ファイル;T;*製品NO;区分;数量;;;
削除ファイル;S;*製品NO;;;
発注ファイル;H;*製品NO;発注量;;;
エラーリスト;E;*製品NO;エラーメッセージ;;;
作業領域;W;製品NO;在庫量;;;
```

図7. 逆生成されたEO仕様

4．　むすび

4．1　評価

事例とした図4のCOBOLプログラムから図7に示したような、[1]機能記述:約50行、[2]識別子の順序記述:1行、[3]レコード記述:7行からなる、等関係式仕様を生成することができた．これをEOSに送りプロセス設計とモジュール設計を行わせると、モジュール仕様が生成された．これらをSPACEに送り、COBOLプログラムを生成させそれを実行させると、図4のCOBOLプログラムと同じ結果を得た．従って、当初の目的であるCOBOLプログラムからの非手続き的仕様の逆生成は正しく行えたと考えられる.

このように本研究では、COBOLプログラムをブロック構造に展開し、『①TYPE1ブロックの選択条件はファイル修飾子の決定に、②TYPE2ブロックの選択条件はIF条件への変換に、③TYPE3ブロックの繰り返し条件は照合基準の決定に、それぞれ対応する』という基本的考えに従って、COBOLプログラムの意図を理解し、非手続き的なEOS仕様に変換する方式を確立した．さらに、そのプロトタイプシステムCORE/Mを開発し、事例に適用することで方式の有効性を実証することができた.

なお、実際に保守用のプログラム修正を行うという観点からは、CORE/Mはこれ単独で利用するのではない．常にCORE/Mの出力はEOSを経由してSPACEに送られ、

SPACEの設計図エディタを用いて、視覚的にも抽象的にも高度なレベルでプログラム仕様を編集する．これによって、プログラムの保守を効果的に行うことができる．

4．2　今後の課題

基本的には2章で述べた制約を取り除いていくことが今後の課題である．そのためには、

[1] EOSがTYPE4(数値条件による繰り返し)を表現できるように拡張する．これによって、制約13が取り除ける．

[2] 等価変換を用いて、お行儀の悪いあるいは暗号的なプログラミングを理解できるようにする．これによって、制約7～11などを取り除ける．[3] プログラムの作者が暗に設けた微妙な処理順序の指定を理解できない．例えば、連続するTYPE2ブロックの実行順序は、TYPE2の条件が独立ならば独立であるが、ブロック内におけるスイッチ的な項目の設定と参照によって微妙に制御を指定している時がある．

さらに、プログラム理解やリエンジニアリングのより高度な目標としては、プログラムから等関係式仕様よりさらに高度な仕様、例えばGRACE[11]における業務仕様やデータモデルなど、への逆変換を行うことである．

謝辞

本システムの一部であるCOBOLプログラムの構文解析部の開発において協力してくれた㈱富士通研究所の上原三八氏と金谷延幸と川辺敬子さんに感謝します．

参考文献

1) Adam, A. and Laurent, J. P.: LAURA - A System to Debug Student Programs, Artif. Intell., Vol.15, No.1-2, pp.75-122(1980).

2) Arnold, R. S.: Software Reengineering, IEEE Computer Society, pp.275-283,520-541(1993).

3) Bohm, C. and Jacopini, G.: Flow Diagrams, Turning Machines And Languages With Only Two Formation Rules, Comm. ACM, Vol.9, No.5, pp.366-371(1966).

4) Biggerstaff, T. J.: Design Recovery for Maintenance and Reuse, IEEE Software, July, Vol.22, No.7, pp.36-49(1989).

5) Chen, P. P.: The Entity-Relationship Model-Toward a Unified View of Data, ACM Trans. Database Syst, Vol.1, No.1, pp.9-36(1976).

6) Chikofsky, E. J. and Cross II, J. H.: Reverse Engineering and Design Recovery: A Taxonomy, IEEE Software, January, Vol.7, No.1, pp.13-17 (1990).

7) 原田実: COBOLﾌﾟﾛｸﾞﾗﾑ自動生成ｼｽﾃﾑSPACEにおける仕様の視覚化と抽象化, 電子情報通信学会論文誌, Vol.J71-D, No.12, pp.2555-2562 (1988.12).

8) 原田実監修: CASEのすべて, オーム社, pp.315-329,451-460 (1991).

9) 原田実, 中村義幸: プログラムの構造と論理の自動設計システムEOS/M, 情報処理学会論文誌, Vol.34, No.9, pp.2013-2024(1993).

10) 原田実, 西村淳一, 中村義幸: 非手続き仕様からのﾌﾟﾛｾｽ設計の自動化, 電子情報通信学会論文誌D-I, Vol.J77-D-I, No.2, pp.196-206(1994).

11) 原田実, 大坪稔房: 出力様式から形式的要求仕様を生成する要求分析システムGRACE, 電子情報通信学会論文誌D-I, (1994採録決定).

12) Johnson, W. E. and Soloway, E.: PROUST: Knowledge Based Program Understanding, IEEE Trans. on Softw. Eng., SE-11, No.3, pp.11-19(1985).

13) 大野豊監修, 原田実編著: 自動プログラミングハンドブック, オーム社, pp.23-40(1989).

14) Sommerville, I.: Software Engineering, Addison-Wesley (1992).

15) 上野春樹: 知的プログラミング環境とプログラム理解, 自動プログラミングハンドブック(原田編), オーム社, pp.23-40(1989).

16) Warnier, J. D. and Flanagan, B. M. : ENTRAINEMENT A LA PROGRAMMATION, Les Edition d'Organisation, PARIS, (1971)(鈴木訳: ﾜｰﾆｴ･ﾌﾟﾛｸﾞﾗﾐﾝｸﾞ法則集, 日本能率協会, 1975).

17) Wills, L. M.: Automated Program Recognition: A Feasibility Demonstration, Artif Intell, Vol.45, No. 1-2, pp. 113-171(1990).

18) 吉野利明, 上原三八, 直田繁樹, 野呂正明, 大久保隆夫: 事務処理プログラムからの仕様抽出, 人工知能学会全国大会予稿集, Vol.6th, No.Pt2, pp.503-506(1992).