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

原田 実[†] 吉川彰一[††,☆] 永井英一郎[††]

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，ファイル処理例題に対する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，等関係式仕様を採用した．この仕様は，ファイル処理例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十分な意味定義になっている．CORE/Mはまず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，234行のCOBOLプログラムを変換した結果，元のプログラムと同じ実行結果を与える等関係式仕様を得た．今後の課題としては，等価変換などを用いて，入力プログラムに課した制約の多くを除くことである．

A COBOL Reverse Engineer CORE/M

—Generation of a Nonprocedural Specification through COBOL Program Understanding—

MINORU HARADA [†], SHOUICHI YOSHIKAWA [††,☆] and EIICHIROU NAGAI [††]

We propose a method of reverse engineering of COBOL programs, and develop a “COBOL Reverse Engineer—CORE/M—” based on the proposed method. CORE/M generates the non-procedural EOS specification 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 program consisting of 234 lines, and has generated its EOS specification bearing the same meaning.

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グ）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

† 青山学院大学理工学部経営工学科
Department of Industrial Engineering, Faculty of Science and Engineering, Aoyama Gakuin University
†† 青山学院大学理工学研究科経営工学専攻
Department of Industrial Engineering, Graduate School of Science and Engineering, Aoyama Gakuin University
☆ 現在，日本電気株式会社
Presently with NEC Corporation

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し，さらに仕様書が計算機処理できる形で記録されていれば，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，できることなら何らかの形式的意味を持ち計算機で理解できる仕様書，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と思われる．

このよう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[2),6)]，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なされている．

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．現状の商用ツールでは，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，これらのツールは入力と出力の抽象度のレベルが同程度であり，視覚性の向上程度の貢献しかなく，単なるリドキュメンテーションツールとしか言いようがない．具体的には，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．

このうち最初の4つは，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[13)]．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，また大規模な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[18)]はCOBOLプログラム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時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適切であるかが明らかにな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精密に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2章で定義する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．ただし，現状では理解できるのはファイル処理を行うプログラムのみで，2章で述べるようないくつかの制約を満たすもの，いわゆるワーニエ法に従ってプログラミングされたもの，のみである．等関係式仕様が得られれば，これを自動設計システムEOS[9),10)]に入力しモジュール仕様を生成し，さらにこれを自動プログラミングシステムSPACE[7)]に入力しCOBOLプログラムを生成させることができる．

以下では，2章においてCORE/Mへの入力と出力が何であるかを議論し，3章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，事例を用いてCORE/Mを評価しその有効性を議論する．

2. CORE/Mの入力と出力

2.1 ファイル処理プログラム

現状のCORE/Mが理解できるのは，ファイル処理を行うCOBOLで記述された事務処理プログラムである．一般に企業内の情報は実体関連モデル[5]で言うところのいわゆる実体や関連の属性情報として，それぞれ対応するファイルに格納されている．ファイル処理プログラムは，これらのファイルからレコードを順次読み込み照合処理や集計処理を行いながら求める属性情報を出力したり更新したりする．この際行われる数値計算は加減乗除で表される簡単なものがほとんどである．

次にCORE/Mの解析方法を説明するのに必要なファイル処理に関する諸概念を定義する．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．また，レコードを物理的に識別する仮想の識別子を考え@で表す．さらに，識別子間の≦順序を，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として導入する．なお一般に，任意の識別子I, I′に対して@≦I, 〈I, I′〉≦I, 〈I, I′〉≦I′である．

2.2 照合基準と識別基準

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識別子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，データの方にもどの実体の属性であるかを表す識別子項目が記録されている．ワーニエはこれらを識別基準と呼んでいる．具体的にはこの項目は，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添え字項目である．一般にファイル処理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．

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
Fig. 1 Block types.

2.3 プログラムの制御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（COBOLでは節と呼ぶ）内で実行される計算は同一対象，特にファイル処理では同一実体あるいは同一関連，に対するものである．

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）の組合せによって記述できるとされている[3]．ファイル処理プログラム中のブロックの実行条件は，照合基準や識別基準を用いて表すと，図1に示すように以下の4つに分けられる．

① TYPE 1：識別基準を照合基準と比較することによる選択構造．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（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．

② TYPE 2：識別基準あるいは属性がある値あるいはある範囲内の時，処理を行う選択構造．

③ TYPE 3：識別基準を照合基準と比較することによる繰返し構造．すなわち，ファイルからのレコード入力ごと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り返し行う構造．

④ TYPE 4：識別基準あるいは属性の値がある範囲の間処理を繰り返す構造．

ただし，繰り返し条件も選択条件も持たないブロックはTYPE 0とする．

2.4 入力となるCOBOLプログラムに対する制約

現在のCORE/Mはまだプロトタイプである．理解できるのは，一言で言えば，ワーニエ法に従ってプログラミングされた綺麗な制御構造を持つファイル処理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ようになる．

1) ファイルはすべて順アクセスファイルである．
2) ファイルは実体や関連を表す．
3) オンライン処理命令を含まない．
4) すべてのファイルは，出力ファイルも含めて，識

別基準を持つ.

5) すべてのファイルはプログラムの最初で初期読み込みされ，レコードを一意に識別できる照合基準と合致する識別基準を持つブロックの終わりで先読み込みされている.

6) レコードの出力は，その構成項目の値が計算されるすべてのブロックを含む最小のブロックで行われる.

7) 作業領域の各項目は必ず，それが所属する実体が関連まで含めて計算対象として特定された初めてのブロックで初期化されている.

8) 照合基準の作成はすべてのファイルの識別基準の最小値とする.

9) 選択はIF文で行う.

10) 繰り返しはPERFORM UNTIL文で行う.

11) 変数を別変数に代入し直して制御するなどの暗号的プログラミングを含まない.

12) ファイルは識別子項目によって，それらがレコード定義中に左側にあるものから順に，それをキーとして昇順にソートされている．すなわち，識別子項目が左に位置しているものの方が≦順で大きい.

13) 複数項目比較による照合処理を含まない．言い換えれば照合基準は単一項目である.

14) キー項目は，各ファイル内で同一名を使用する.

15) 変数は，すべてグローバル変数とする.

16) CALL文のような引数つき呼び出しは行わない.

17) 実行条件が同じ計算は1つのブロック内で行う．言い換えれば，見やすさのためにむやみにブロック分けしない.

18) 数値条件による繰り返し処理(TYPE 4)を含まない.

制約1〜3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4〜10はワーニエ法に従ってプログラミングされていることを示している.制約11〜17は分析を効率的に行うために設けた.制約18は数値条件の繰り返しを表現するには等関係式仕様の拡張が必要とされるため課した.なお,我々は処理構造からプログラムを理解する過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.

19) 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19が課せられていると理解における推定の検証が行えて有効である.

2.5 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，右辺は左辺項目の計算的意味を定義するための項目からなる算術式や条件式からなる等式である．ここで，各項目Xは，添え字や"_"を用いてその項目が存在するファイルFの識別子で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)のように修飾されている.この添え字Fを項目Xのファイル修飾子と呼ぶ．さらに，項目は次に形式的に述べるように，"."を用いてそれが属する実体あるいは関連の識別子で修飾される.

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.

$$\mathrm{group\_F}(X, x)=\{f \mid \exists f \in F, X(f)=x\} \quad (1)$$

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．従って，一般的には，$X_F.I_G$はファイルG内の各レコードgの識別子Iの各値iに対するファイルF内のグループgroup_F(I, i)内の各レコードfの項目Xの値リストの内容をiの昇順に並べたリストとして定義される.このような"."1つによる，実体Iと属性Xの修飾を一段修飾という.

$$X_F.I_G=\{X(f) \mid \exists g \in G, \exists f \in \mathrm{group\_F}(I, I(g))\} \quad (2)$$

ファイル処理に対する本研究の根底となる考え方は,『更新あるいは出力される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で表される実体$I0_F$=値i0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，ファイルF1が実体I1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$I0_{F1}$=値i0とこのファイル内で関連するす

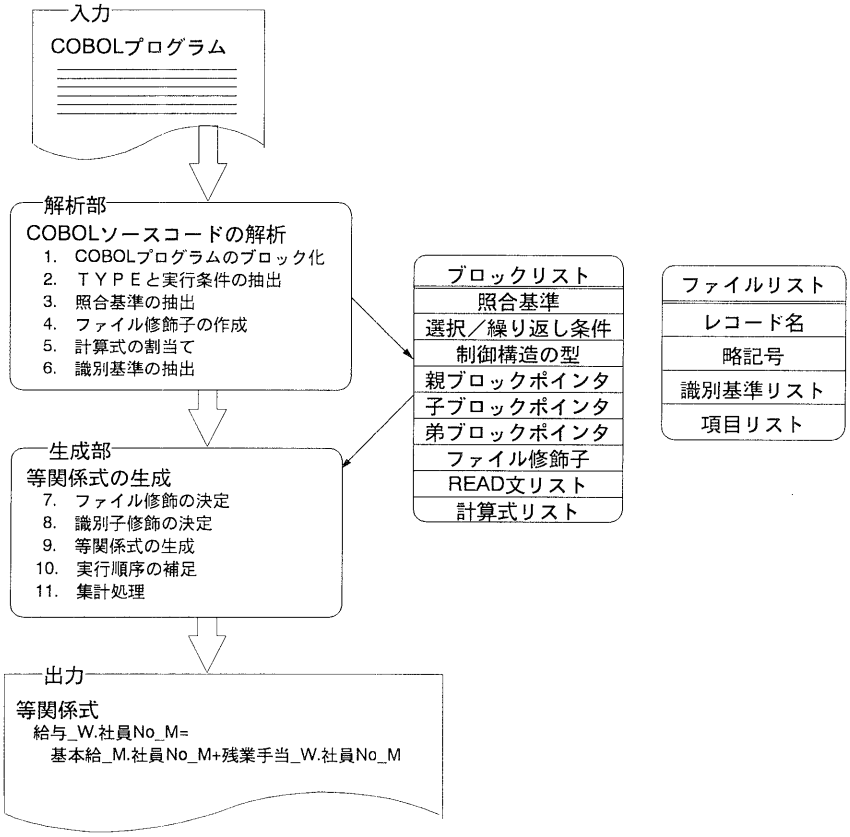


図2 COREにおける処理の流れ
Fig. 2 Analysis and generation steps in CORE.

べての実体I1の属性$X1_{F1}.\langle I0, I1\rangle_{F1}$などに，関数gを適用させて求まる』ということである．

$$X_L.I0_F=g(X0_{G0}.I0_{F0}, X1_{F1}.\langle I0, I1\rangle_{F1}, \cdots) \quad (3)$$

この式(3)を等関係式と呼び，左辺の項目の計算的意味を右辺の条件式を含む算術式で定義している[9]．

等関係式には条件式は含まれるが繰り返し式は含まれない．これは等関係式の各項目が，(2)に示すように，1つの値ではなく値の順序集合(リスト)であることに関係している．すなわち，等関係式(3)は識別子$I0_F$の各値に対するX_Lの各値を，その左辺で計算される値で与えることを表している．このことは一般にファイル処理プログラム内の1つの繰り返し処理ブロック内では1つの属性に対する計算式は1通りであることに対応している．このことは非常に重要で，後で分かるようにプログラムの意味を等関係式仕様を用いて適切に表せることを意味している．

2.6 照合ファイル

ある実体の属性（例：社員の残業単価と残業時間など)が異なるファイルに記録され，『ファイルに当該レコードがあるかないか』という事実が，このレコードで表される実体の状態（例：残業レコードがファイルZにあることが，残業をしたかしなかったか)を表現することがよくある．この状況は照合ファイルを用いて表現できる．照合ファイルF1&F2[I]とは，2つのファイルF1とF2の各レコードを両ファイルに共通の識別子Iを照合キーにして，その値が同じレコード同士を連結した仮想的なレコードからなるファイルである．なお，1つの照合に参加するファイルの数は共通のキーを持てば3つ以上でもよい．

$$F1\&F2[I]=\{f1\hat{}f2 \mid \exists f2\in F2,\ \exists f1\in F1,\ I(f1)=I(f2)\}^{*} \quad (4)$$

また照合ファイルF1&!F2[I]や!F1&F2[I]は，実体Iを表すレコードがファイルF1やF2だけにあって他方にないファイルである．

$$F1\&!F2[I]=\{f1 \mid \forall f2\in F2,\ \exists f1\in F1,\ I(f1)\neq I(f2)\} \quad (5)$$

* f1^f2はレコードf1の次にf2を繋げたレコードである．

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ムでは，この条件は照合基準と識別基準の比較条件がほとんどである．先に述べたように，同一ブロック内の計算式の左辺項目はこの照合基準で識別される実体の属性であると考えられるので，これらの項目は照合基準によって“.”修飾される．従って，プログラムをブロック構造に展開すること，ブロックの照合基準を求めることなどが必要であることが分かる．

これらのことから，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，解説にあたっては事例として**図3**に示すような給与計算問題を用いた．この問題は，各社員ごとの集計と部ごとの集計という2重の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への入力事例として用いる．

3.1 COBOLプログラムのブロック化

特定の条件が満たされた時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．

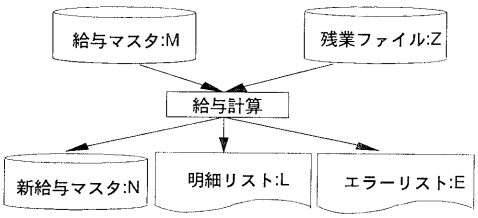
ブロックはPERFORM文で他のブロックを呼び出すことができる．この場合，**図5**に示すように，呼び出すブロックを親ブロック，呼び出されるブロックを子ブロックという．また同じ親の子ブロック同士内で，呼び出す順で先のものを兄ブロック，後のものを弟ブロックという．さらに，親，親の親，さらにその親などを先祖ブロックといい，逆に，子，子の子，さらにその子などを子孫ブロックという．

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，ブロックごとにブロックに

［処理概要］

残業ファイルにより、給与マスタを更新する。同時に明細リストとエラーリストも出力する。

［プロセスフロー］



［処理条件］

1．基本給に当月の残業手当を加えて給与累計を求める。残業には、通常、深夜、早朝の3種類あり、残業区分が0、1、2で区別される。

2．残業手当は下記の計算式で求め、社員ごとに集計する。
通常残業の時 (残業区分 "0") －＞ 残業単価＊残業時間
深夜残業の時 (残業区分 "1") －＞ 残業単価＊残業時間＊1.5
早朝残業の時 (残業区分 "2") －＞ 残業単価＊残業時間＊1.2

3．残業レコードにエラーが含まれる時は、エラーメッセージを添えてエラーリストに打出す。残業区分が0、1、2以外の時は、"残業区分エラー" を打出す。通常残業は5時間以内深夜残業は2時間以内、早朝残業は3時間以内とする。この範囲を超える時、"残業時間エラー" を打出す。給与マスタに存在しない社員Noを持つ残業レコードに対しては、"残業部のみ" や "残業社員のみ" を打出す。

図3 例題としての給与計算問題
Fig. 3 Sample salary calculation problem.

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，弟ブロックの先頭へのポインタ，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，ファイルごと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．

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TYPE 4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．

例：事例に対しては，図6の下段に示したようになる．なお，C{Bはブロック Bを条件Cの時実行するの意味である．また，C↓{Bはブロック Bを条件Cが満足されるまで繰り返す意味である．ここで，C2～C5はキー項目比較であり，C1とC6～C11は数値比較

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  給与問題.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT  給与マスタ      ASSIGN F-DISK.
  SELECT  残業ファイル    ASSIGN F-DISK.
  SELECT  新給与マスタ    ASSIGN F-DISK.
  SELECT  明細リスト      ASSIGN F-DISK.
  SELECT  エラーリスト    ASSIGN F-DISK.

DATA DIVISION.
FILE SECTION.
FD    給与マスタ.
01    マスタレコード.
      02  部社員Ｎｏ.
          03  部Ｎｏ     PIC X(1).
          03  社員Ｎｏ   PIC X(4).
      02  社員名         PIC X(10).
      02  基本給         PIC 9(10).
      02  単価           PIC 9(5).
      02  支給累計       PIC 9(10).

FD    残業ファイル.
01    残業レコード.
      02  部社員Ｎｏ.
          03  部Ｎｏ     PIC X(1).
          03  社員Ｎｏ   PIC X(4).
      02  区分           PIC 9(1).
      02  時間           PIC 9(4).

FD    新給与マスタ.01    新マスタレコード.
      02  部社員Ｎｏ.
          03  部Ｎｏ     PIC X(1).
          03  社員Ｎｏ   PIC X(4).
      02  社員名         PIC X(10).
      02  基本給         PIC 9(10).
      02  単価           PIC 9(5).
      02  支給累計       PIC 9(10).

FD    明細リスト.
01    明細レコード１.
      02  部社員Ｎｏ.
          03  部Ｎｏ     PIC X(1).
          03  社員Ｎｏ   PIC X(4).
      02  社員名         PIC X(10).
      02  給与           PIC Z(10).
      02  残業額合計     PIC Z(10).
      02  支給累計       PIC Z(10).
01    明細レコード２.
      02  FILLER         PIC X(5).
      02  部Ｎｏ         PIC X(1).
      02  FILLER         PIC X(4).
      02  部合計         PIC Z(10).
      02  FILLER         PIC X(5).
      02  部平均         PIC Z(10).
      02  FILLER         PIC X(10).

FD    エラーリスト.
01    エラーレコード.
      02  部社員Ｎｏ.
          03  部Ｎｏ     PIC X(1).
          03  社員Ｎｏ   PIC X(4).
      02  エラーメッセージ  PIC X(50).

WORKING-STORAGE SECTION.
01    作業領域.
      02  部社員Ｎｏ.
          03  部Ｎｏ     PIC X(1).
          03  社員Ｎｏ   PIC X(4).
      02  給与           PIC 9(10).
      02  支給累計       PIC 9(10).
      02  部合計         PIC 9(10).
      02  部平均         PIC 9(10).
      02  部社員数       PIC 9(3).
      02  残業額合計     PIC 9(10).
      02  残業額         PIC 9(10).

PROCEDURE DIVISION.
初期処理.
    OPEN  INPUT   給与マスタ.
    OPEN  INPUT   残業ファイル.
    OPEN  OUTPUT  新給与マスタ.
    OPEN  OUTPUT  明細リスト.
    OPEN  OUTPUT  エラーリスト.

    READ  給与マスタ
      AT END MOVE 'E' TO 部Ｎｏ OF 給与マスタ.
    READ  残業ファイル
      AT END MOVE 'E' TO 部Ｎｏ OF 残業ファイル.

 PERFORM 部処理
    UNTIL（部Ｎｏ OF 給与マスタ = 'E' AND 部Ｎｏ OF 残業ファイル = 'E'）.

終了処理.
    CLOSE 給与マスタ.
    CLOSE 新給与マスタ.
    CLOSE 残業ファイル.
    CLOSE 明細リスト.
 CLOSE エラーリスト.

    STOP RUN.

部処理.
    IF 部Ｎｏ OF 給与マスタ ＜ 部Ｎｏ OF 残業ファイル
       THEN MOVE 部Ｎｏ OF 給与マスタ TO 部Ｎｏ OF 作業領域
       ELSE MOVE 部Ｎｏ OF 残業ファイル TO 部Ｎｏ OF 作業領域.
    IF 部Ｎｏ OF 給与マスタ = 部Ｎｏ OF 作業領域
      THEN PERFORM マスタ部有処理
      ELSE PERFORM マスタ部無処理.

マスタ部有処理.
    MOVE 0 TO 部合計 OF 作業領域.
    MOVE 0 TO 部平均 OF 作業領域.
    MOVE 0 TO 部社員数 OF 作業領域 .
    PERFORM 社員処理
      UNTIL（部Ｎｏ OF 給与マスタ = 'E' AND 部Ｎｏ OF 残業ファイル = 'E'）
         OR（部Ｎｏ OF 作業領域 NOT = 部Ｎｏ OF 給与マスタ）
    COMPUTE 部平均 OF 作業領域 = 部合計 OF 作業領域 / 部社員数 OF 作業領域.
    MOVE 部Ｎｏ OF 作業領域 TO 部Ｎｏ OF 明細レコード２.
    MOVE 部合計 OF 作業領域 TO 部合計 OF 明細レコード２.
    MOVE 部平均 OF 作業領域 TO 部平均 OF 明細レコード２.
    WRITE 明細レコード２ OF 明細リスト.

マスタ部無処理.
    MOVE 部社員Ｎｏ OF 残業ファイル TO 部社員Ｎｏ OF エラーリスト.
    MOVE '残業部のみ' TO エラーメッセージ OF エラーリスト.
    WRITE エラーレコード OF エラーリスト.
    IF（部Ｎｏ OF 残業ファイル NOT = 'E'）
      THEN READ 残業ファイル AT END MOVE 'E' TO 部Ｎｏ OF 残業ファイル.

社員処理.
    IF（部社員Ｎｏ OF 給与マスタ ＜ 部社員Ｎｏ OF 残業ファイル）
      THEN MOVE 部社員Ｎｏ OF 給与マスタ TO 部社員Ｎｏ OF 作業領域
      ELSE MOVE 部社員Ｎｏ OF 残業ファイル TO 部社員Ｎｏ OF 作業領域.
    IF（部社員Ｎｏ OF 給与マスタ = 部社員Ｎｏ OF 作業領域）
      THEN PERFORM マスタ社員有処理
      ELSE PERFORM マスタ社員無処理.


マスタ社員有処理.
    MOVE 0 TO 残業額合計 OF 作業領域.
    IF（部社員Ｎｏ OF 作業領域 = 部社員Ｎｏ OF 残業ファイル）
        THEN PERFORM 残業社員有処理
      ELSE PERFORM 残業社員無処理.
    COMPUTE 給与 OF 作業領域 =
                    基本給 OF 給与マスタ ＊ 残業額合計 OF 作業領域.
    COMPUTE 部社員数 OF 作業領域 = 部社員数 OF 作業領域 ＋ 1.
    COMPUTE 支給累計 OF 新給与マスタ =
                    支給累計 OF 給与マスタ ＋ 給与 OF 作業領域.

    MOVE 部社員Ｎｏ OF 給与マスタ TO 部社員Ｎｏ OF 新給与マスタ.
    MOVE 社員名 OF 給与マスタ TO 社員名 OF 新給与マスタ.
    MOVE 基本給 OF 給与マスタ TO 基本給 OF 新給与マスタ.
    MOVE 単価 OF 給与マスタ TO 単価 OF 新給与マスタ.
    WRITE 新マスタレコード OF 新給与マスタ.

    MOVE 部社員Ｎｏ OF 給与マスタ TO 部社員Ｎｏ OF 明細レコード１.
    MOVE 社員名 OF 給与マスタ TO 社員名 OF 明細レコード１.
    MOVE 給与 OF 作業領域 TO 給与 OF 明細レコード１.
    MOVE 残業額合計 OF 作業領域 TO 残業額合計 OF 明細レコード１
    MOVE 支給累計 OF 新給与マスタ TO 支給累計 OF 明細レコード１.
    WRITE 明細レコード１ OF 明細リスト.
    COMPUTE 部合計 OF 作業領域 = 部合計 OF 作業領域 ＋ 給与 OF 作業領域.

    IF（部Ｎｏ OF 給与マスタ NOT = 'E'）
      THEN READ 給与マスタ AT END MOVE 'E' TO 部Ｎｏ OF 給与マスタ.

マスタ社員無処理.
    MOVE 部社員Ｎｏ OF 残業ファイル TO 部社員Ｎｏ OF エラーリスト.
    MOVE '残業社員のみ' TO エラーメッセージ OF エラーリスト.
    WRITE エラーレコード OF エラーリスト.
    IF（部Ｎｏ OF 残業ファイル NOT = 'E'）
      THEN READ 残業ファイル AT END MOVE 'E' TO 部Ｎｏ OF 残業ファイル.

残業社員有処理.
    PERFORM 残業レコード処理
      UNTIL（部Ｎｏ OF 給与マスタ = 'E' AND 部Ｎｏ OF 残業ファイル = 'E'）
         OR（部Ｎｏ OF 作業領域 NOT = 部Ｎｏ OF 給与マスタ）
         OR（部社員Ｎｏ OF 作業領域 NOT = 部社員Ｎｏ OF 残業ファイル）.

残業レコード処理.
    MOVE 0 TO 残業額.
    PERFORM 残業レコード検査.
    COMPUTE 残業額合計 OF 作業領域 = 残業額合計 OF 作業領域 ＋ 残業額.
    IF（部Ｎｏ OF 残業ファイル NOT = 'E'）
      THEN READ 残業ファイル
           AT END MOVE 'E' TO 部Ｎｏ OF 残業ファイル.


残業社員無処理.
    MOVE 0 TO 残業額合計 OF 作業領域.


残業レコード検査.
    IF（区分 OF 残業ファイル NOT = 0）
       AND（区分 OF 残業ファイル NOT = 1）
       AND（区分 OF 残業ファイル NOT = 2）
      THEN MOVE 部社員Ｎｏ OF 残業ファイル TO 部社員Ｎｏ OF エラーリスト
           MOVE '残業区分エラー' TO エラーメッセージ OF エラーリスト
           WRITE エラーレコード OF エラーリスト.

    IF（（区分 OF 残業ファイル = 0）AND（時間 OF 残業ファイル ＞ 5））
       OR（（区分 OF 残業ファイル = 1）AND（時間 OF 残業ファイル ＞ 2））
       OR（（区分 OF 残業ファイル = 2）AND（時間 OF 残業ファイル ＞ 3））
      THEN MOVE 部社員Ｎｏ OF 残業ファイル TO 部社員Ｎｏ OF エラーリスト
           MOVE '残業時間エラー' TO エラーメッセージ OF エラーリスト
           WRITE エラーレコード OF エラーリスト.

    PERFORM 残業額計算.


残業額計算.
    IF（（区分 OF 残業ファイル = 0）AND（時間 OF 残業ファイル NOT ＞ 5））
       COMPUTE 残業額 = 単価 OF 給与マスタ ＊ 時間 OF 残業ファイル.
    IF（（区分 OF 残業ファイル = 1）AND（時間 OF 残業ファイル NOT ＞ 2））
       COMPUTE 残業額 = 単価 OF 給与マスタ ＊ 時間 OF 残業ファイル ＊ 1.5.
    IF（（区分 OF 残業ファイル = 2）AND（時間 OF 残業ファイル NOT ＞ 3））
       COMPUTE 残業額 = 単価 OF 給与マスタ ＊ 時間 OF 残業ファイル ＊ 1.2.
```

図4　給与計算問題のCOBOLプログラム
Fig. 4　Cobol program for the sample salary calculation problem.

であり，その他はこれらの合成である．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．

TYPE 0：①,⑪,⑭

TYPE 1：③,④,⑥,⑦,⑧,⑨

TYPE 2：⑫,⑬,⑮,⑯,⑰

TYPE 3：②,⑤,⑩

TYPE 4：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-STORAGE 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）

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選択処理や繰り返し処理の条件に使用されている．

照合基準は，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，あるブロックで設定されると，次にその子孫ブロックで別の（より詳細レベルの）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．このようにして各ブロックごとに有効な照合基準が一意に決まる．

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

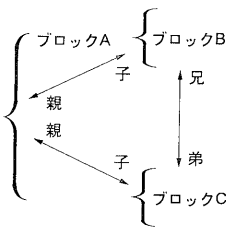


図5 ブロック間の関係
Fig. 5 Block relationships.

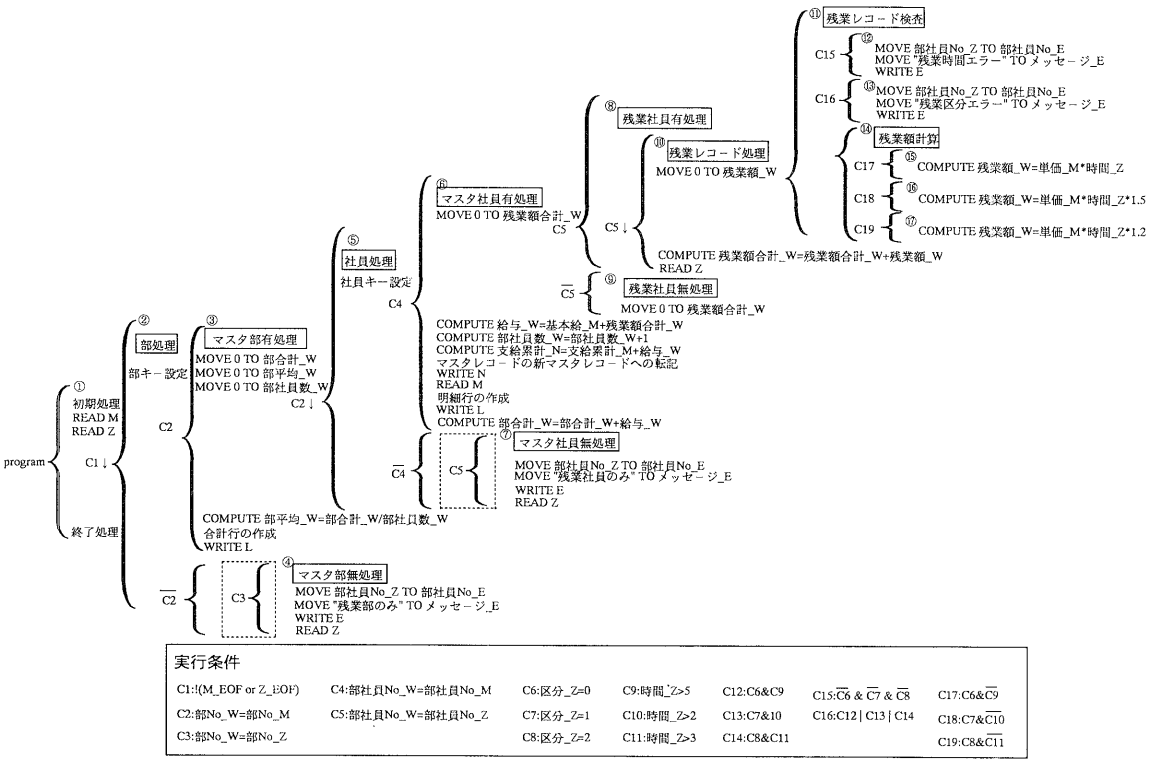


図6 事例のCOBOLプログラムのブロック構造への展開と実行条件
Fig. 6 The block structure and conditions of the sample program.

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TYPE 3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TYPE 3ブロックが処理する対象の照合基準を設定する．

しかし，時には照合基準を設定しないTYPE 3ブロック（例：ブロック⑩）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，プログラムの最外側のブロック（例：ブロック①）はすべ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．

例：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＝部Noのブロック：②～④
照合基準＝部社員Noのブロック：⑤～⑨
照合基準＝＠のブロック：⑩～⑰

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，処理対象がどんな識別子で修飾される実体なのか（例：部なのか社員なのか）が明らかになった．次に明らかにすべきは，この実体がどんな関連を持つ実体なのか（例：残業した社員なのかしなかった社員なのか）を明らかにすることである．ファイル処理では，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ことが多い．この時には，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とキー項目値を共有するレコードが他のファイルに存在するかしないかによって表現される．従って，各ブロックの照合基準で指定される実体がどんなファイル群にレコードを持つ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は次のようにして行う．まず，最外側のブロック（例：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．同様にして，以降のTYPE 3ブロックで（＠以外の）照合基準が設定されるたびにこの照合基準に対するファイル修飾子FLを作成し，初期値を設定する．以下，このブロックの子孫ブロックのファイル修飾子は，通常親ブロックのファイル修飾子を継承するが（例：ブロック②では部NoにするFL＝φ），それがTYPE 1ブロックの時には変更される．すなわち，TYPE 1ブロックの実行条件において照合基準（例：部No_W）と比較される識別基準（例：部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でこのブロックのファイル修飾子を求める（例：ブロック③では部NoにするFL＝&M）．なお，φに何らかのファイルが&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-を用いる．なお，この議論において＠が照合基準として設定された場合は，既に先祖ブロックでどのファイルに対する処理か決定されているはずであるから，ファイル修飾子をφで初期化せず，親ブロックのものを継承する．

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：ブロック①のREAD MとREAD Z）があれば，基本的にこれら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，ワーニエ法の1つである制約8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プログラミング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，2つのファイルM，Zにおいて，一方のファイルM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!MはZ&!Mと考えてよい．この推定は，このブロック内にファイルZ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Zの識別基準のTYPE 1ブロック（例：ブロック④の外側の点線で囲まれたC 3条件のブロック）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!MをZ&!Mとする．

例：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．

$に対するファイル修飾子
　＝-のブロック：①
部Noに対するファイル修飾子
　＝-のブロック：②
部Noに対するファイル修飾子
　＝Mのブロック：③
部Noに対するファイル修飾子

＝!M＝Z&!Mのブロック：④

部社員Noに対するファイル修飾子

＝-のブロック：⑤

部社員Noに対するファイル修飾子

＝Mのブロック：⑥

部社員Noに対するファイル修飾子

＝!M＝Z&!Mのブロック：⑦

部社員Noに対するファイル修飾子

＝M&Zのブロック：⑧

部社員Noに対するファイル修飾子

＝M&!Zのブロック：⑨

@に対するファイル修飾子

＝M&Zのブロック：⑩～⑰

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．なお，ここで制約8による識別基準の最小値による照合基準の作成のためのMOVE文は計算式としては抽出しない（例：図4の部処理節および社員処理節の先頭3行）．同様に，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読み込みなのかを含めて，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，TYPE 2ブロックの条件を構成する項目，をまとめて左辺項目という．またMOVE文のTO句の前の項目やCOMPUTE文の右辺を構成する項目をまとめて右辺項目という．

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を表すかを明らかにできる．この結果をファイルごとにファイルリストに記録しておく．なお，この際@を照合基準に持つTYPE 3ブロック内にREAD FなるファイルFの読み込みがあれば，ファイルFは識別子をすべて指定してもそれらの値を持つレコードが複数件あることを表しているので，このファイルの識別基準に@を加える．

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，データ構造とキーの順位を生成する．制約12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]のように生成する．なお，部社員Noは部Noと社員Noからなる複合キーであり，こ

```
［1］機能記述
部合計_W.部No_M = 0.部No_M.
部平均_W.部No_M = 0.部No_M.
部社員数_W.部No_M = 0.部No_M.
部平均_W.部No_M = 部合計_W.部No_M / 部社員数_W.部No_M.
部No-OF-明細レコード2_L.部No_M = 部No_W.部No_M.
部合計-OF-明細レコード2_L.部No_M = 部合計_W.部No_M.
部平均-OF-明細レコード2_L.部No_M = 部平均_W.部No_M.
WRITE(明細レコード2_L.部No_M).
明細レコード2_L.部No_M = FOLLOW(部No-OF-明細レコード2_L.部No_M^
  部合計-OF-明細レコード2_L.部No_M^部平均-OF-明細レコード2_L.部No_M).
部社員No_E.部No_!M&Z = 部社員No_Z.部No_!M&Z.
エラーメッセージ_E.部No_!M&Z = '残業部のみ'.部No_!M&Z.
WRITE(エラーレコード_E.部No_!M&Z).
エラーレコード_E.部No_!M&Z = FOLLOW(部社員No_E.部No_!M&Z^
  エラーメッセージ_E.部No_!M&Z).
残業額合計_W.部社員No_M = 0.部社員No_M.
給与_W.部社員No_M = 基本給_M.部社員No_M + 残業額合計_W.部社員No_M.
部社員数_W.部No_M = SUM(1.部社員No_M).
支給累計_N.部社員No_M = 支給累計_M.部社員No_M + 給与_W.部社員No_M.
部社員No_N.部社員No_M = 部社員No_M.部社員No_M.
社員名_N.部社員No_M = 社員名_M.部社員No_M.
基本給_N.部社員No_M = 基本給_M.部社員No_M.
単価_N.部社員No_M = 単価_M.部社員No_M.
WRITE(新マスタレコード_N.部社員No_M).
新マスタレコード_N.部社員No_M = FOLLOW(部社員No_N.部社員No_M^
  社員名_N.部社員No_M^基本給_N.部社員No_M^単価_N.部社員No_M^
  支給累計_N.部社員No_M).
部社員No-OF-明細レコード1_L.部社員No_M = 部社員No_M.部社員No_M.
社員名-OF-明細レコード1_L.部社員No_M = 社員名_M.部社員No_M.
給与-OF-明細レコード1_L.部社員No_M = 給与_W.部社員No_M.
残業額合計-OF-明細レコード1_L.部社員No_M = 残業額合計_W.部社員No_M.
支給累計-OF-明細レコード1_L.部社員No_M = 支給累計_N.部社員No_M.
WRITE(明細レコード1_L.部社員No_M).
明細レコード1_L.部社員No_M = FOLLOW(部社員No-OF-明細レコード1_L.部社員No_M^
  社員名-OF-明細レコード1_L.部社員No_M^給与-OF-明細レコード1_L.部社員No_M^
  残業額合計-OF-明細レコード1_L.部社員No_M^
  支給累計-OF-明細レコード1_L.部社員No_M).
明細レコード1_L.部社員No_M = FOLLOW(新マスタレコード_N.部社員No_M).
部合計_W.部No_M = SUM(給与_W.部社員No_M).
部社員No_E.部社員No_!M&Z = 部社員No_Z.部社員No_!M&Z.
エラーメッセージ_E.部社員No_!M&Z = '残業社員のみ'.部社員No_!M&Z.
WRITE(エラーレコード_E.部社員No_!M&Z).
エラーレコード_E.部社員No_!M&Z = FOLLOW(部社員No_E.部社員No_!M&Z^
  エラーメッセージ_E.部社員No_!M&Z).
残業額合計_W.部社員No_M&!Z = 0.部社員No_M&!Z.
残業額_W.@_M&Z = 0.@_M&Z.
残業額合計_W.部社員No_M&Z = SUM(残業額_W.@_M&Z).
IF ((区分_Z.@_M&Z != 0.@_M&Z)
 & (区分_Z.@_M&Z != 1.@_M&Z)
 & (区分_Z.@_M&Z != 2.@_M&Z)) {
  部社員No_E.@_M&Z = 部社員No_Z.@_M&Z.
  エラーメッセージ_E.@_M&Z = '残業区分エラー'.@_M&Z.
  WRITE(エラーレコード_E.@_M&Z).
  エラーレコード_E.@_M&Z = FOLLOW(部社員No_E.@_M&Z^エラーメッセージ_E.@_M&Z).
}
IF (((区分_Z.@_M&Z = 0.@_M&Z) & (時間_Z.@_M&Z > 5.@_M&Z))
 | ((区分_Z.@_M&Z = 1.@_M&Z) & (時間_Z.@_M&Z > 2.@_M&Z))
 | ((区分_Z.@_M&Z = 2.@_M&Z) & (時間_Z.@_M&Z > 3.@_M&Z))) {
  部社員No_E.@_M&Z = 部社員No_Z.@_M&Z.
  エラーメッセージ_E.@_M&Z = '残業時間エラー'.@_M&Z.
  WRITE(エラーレコード_E.@_M&Z).
  エラーレコード_E.@_M&Z = FOLLOW(部社員No_E.@_M&Z^エラーメッセージ_E.@_M&Z).
}
IF ((区分_Z.@_M&Z = 0.@_M&Z) & (時間_Z.@_M&Z <= 5.@_M&Z))
 残業額_W.@_M&Z = 単価_M.@_M&Z * 時間_Z.@_M&Z.
IF ((区分_Z.@_M&Z = 1.@_M&Z) & (時間_Z.@_M&Z <= 2.@_M&Z))
 残業額_W.@_M&Z = 単価_M.@_M&Z * 時間_Z.@_M&Z * 1.5.@_M&Z.
IF ((区分_Z.@_M&Z = 2.@_M&Z) & (時間_Z.@_M&Z <= 3.@_M&Z))
 残業額_W.@_M&Z = 単価_M.@_M&Z * 時間_Z.@_M&Z * 1.2.@_M&Z.

［2］識別子の順序記述
@<区分;
区分<部社員No;
部社員No=<部No,社員No>;
社員No<部No;

［3］レコード記述
給与マスタ;M;マスタレコード;＊部No;社員No;＊部社員No;社員名;基本給;単価;;支給累計;;;
残業ファイル;Z;残業レコード;＊部No;社員No;＊部社員No;＊区分;＊@;社員No;時間;;;
新給与マスタ;N;新マスタレコード;＊部No;社員No;＊部社員No;社員名;基本給;単価;支給累計;;;
明細リスト;L;明細レコード1;＊部No;社員No;＊部社員No;社員名;給与;残業額合計;支給累計;;
  明細レコード2;＊部No;部合計;部社員数;部平均;;;
エラーリスト;E;エラーレコード;＊部No;社員No;＊部社員No;エラーメッセージ;;;
作業領域;W;作業領域;部No;社員No;部社員No;給与;支給累計;部合計;部平均;部社員数;
  残業額合計;残業額;;;
```

図7 逆生成されたEOS仕様

Fig. 7 The EOS specification regenerated from the sample program.

のことを示すために，3行目に示すような等式を生成する．一方，各ファイルのレコードごとの構成項目(キー項目の前には＊を付ける)を図7の[3]のように，“ファイル名；ファイルID；{レコード名；{[＊]項目名；};}；”の形式で生成する．

3.7 項目のファイル修飾の決定

COBOLプログラム中のすべての項目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名やレコード名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-STORAGE 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-STORAGE 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，例えばR1とR2があり，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すべての項目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（例：部内の社員の給与合計）は当該ブロックの照合基準（例：社員）より大きな照合基準(例：部)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．なお，定数Cは一律に，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．

CORE/Mでは，項目D_Fを修飾する識別子の決定は，Fが入力ファイル，出力ファイル，作業領域のいずれかによって異なる．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祖ブロックの内で，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つけ，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：ブロック⑥の基本給_Mに対しては，ブロック⑥そのもののファイル修飾子にMがあるので，基本給_M.部社員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この項目を含む式より後ろにある子孫ブロックのうちで，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つけ，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：ブロック⑥の支給累計_Nに対しては，ブロック⑥そのものがNへのWRITE文を持つので，支給累計_N.部社員NO_M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7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とし，ブロックBあるいはその先祖ブロックのうちで，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′を見つける．ブロックBのファイル修飾子をFL，ブロックB′の照合基準をI′として，D_WをD_W.I′_FLに変換する．

例：ブロック⑨(ファイル修飾子＝M&!Z)の残業額合計_Wに対しては，親ブロック⑥（照合基準＝部社員NO)がそれを初期設定している．従って，残業額合計_W.部社員NO_M&!Zとなる．また，ブロック⑥（ファイル修飾子＝M）内の3つの給与_Wに対しては，ブロック⑥(照合基準＝部社員NO)そのものがそれを初期設定しているので，給与_W.部社員NO_Mとなる．また，ブロック⑥（ファイル修飾子＝M）の部合計_Wに対しては，先祖ブロック③（照合基準＝部NO）にその初期設定があるので，部合計_W.部NO_Mとなる．

なお，以上の判定で，このような直近のブロックが存在しなければ，制約5,6,7に違反するプログラミン

グなので理解できないとして，その項目の識別子を?_-とする．また，見つかったブロックのファイル修飾子がφの場合は－とする．

なお，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，通常その照合キーは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TYPE 3ブロックの照合基準とする（例：ブロック⑮の時間_Z.@_M&Z[部社員No]）．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，煩雑さを減らすために照合キーを省略する（例：ブロック⑩の残業額合計．部社員No_M&Z[部社員No]→残業額合計．部社員No_M&Z）．

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すべ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TYPE 2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TYPE 2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，TYPE 2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．

例：事例においては，例えば

ブロック⑥の COMPUTE部合計_W＝部合計_W＋給与_Mは

部合計_W.部NO_M＝部合計_W.部NO_M＋給与_M.部社員NO_Mに，

ブロック⑥の COMPUTE支給累計_M＝支給累計_M＋給与_Mは

支給累計_M.部社員NO_M＝支給累計_M.部社員NO_M＋給与_M.部社員NO_Mに，

ブロック⑩の COMPUTE残業額合計_W＝残業額合計_W＋残業額_Wは

残業額合計_W.部社員NO_M＝残業額合計_W.部社員NO_M＋残業額_W.@_M&Z[部社員NO]に，

ブロック⑯の COMPUTE残業額_W＝単価_M＊時間_Z＊1.5は

IF((区分_Z.@_M&Z[部社員NO]＝1.@_M&Z[部社員NO])

&(時間_Z.@M&Z[部社員NO]＜＝2.@_M&Z[部社員NO]))

残業額_W.@_M&Z[部社員NO]＝単価_M.@M&Z[部社員NO]

＊時間_Z.@_M&Z[部社員NO]＊1.5.@_M&Z[部社員NO].に，

それぞれ変換される．

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, …ごとに，R＝FOLLOW (D1^D2^…) なる等関係式を生成する．なお，FOLLOW文内の項目の識別子修飾は左辺の出力ファイルの項目と同じものを使う．また連続するWRITE (R1) とWRITE (R2) に対しては，R2＝FOLLOW (R1) を生成する．

例：ブロック⑥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.部社員No_M＝FOLLOW (部社員No_N.部社員No_M^社員名_N.部社員No_M^基本給_N.部社員No_M^単価_N.部社員No_M^支給累計_N.部社員No_M).

3.11 クリシェの意味変換

TYPE 3ブロックに加算式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などの初期設定式があれば，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，COMPUTE A＝A＋BをA＝SUM(B)に変換する．ただし，初期設定式は他のブロックでも必要とされていたりすることがあるので，消さないで置いておく．なお，このよう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も行う．

4. むすび

4.1 評価

CORE/Mに対して事例とした図4のCOBOLプロ

グラムを入力したところ，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：2行，[3]レコード記述：8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/Mに送りモジュール設計を行わせると，5個の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プログラムと同じ計算結果を得た．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，『①TYPE 1ブロックの選択条件はファイル修飾子の決定に，②TYPE 2ブロックの選択条件はIF条件への変換に，③TYPE 3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TYPE 4(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18が取り除ける．

[2] ワーニエ法からくる制約4〜10や分析の効率化からくる制約11〜17を満足しないプログラムも理解できるようにする．これには，プログラムの等価変換が有効であると考えられる．

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氏と川辺敬子さんに感謝します．

参考文献

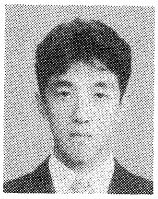
1) Adam, A. and Laurent, J. P.: LAURA—A System to Debug Student Programs, *Artif. Intell.*, Vol. 15, No. 1-2, pp. 75-122 (1980).
2) Arnold, R. S.: *Software Reengineering*, pp. 275-283, 520-541, IEEE Computer Society (1993).
3) Bohm, C. and Jacopini, G.: Flow Diagrams, Turning Machines And Languages 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*, 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*, Vol. 7, No. 1, pp. 13-17 (1990).
7) 原田 実：COBOLプログ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. J 71-D，No. 12，pp. 2555-2562 (1988).
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プロセス設計の自動化，電子情報通信学会論文誌D-I，Vol. J 77-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，Vol. J77-D-I，No. 9，pp. 635-645 (1994).
12) Johnson, W. E. and Soloway, E.: PROUST: Knowledge Based Program Understanding, *IEEE Trans. Softw. Eng.*, Vol. 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 (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)，pp. 23-40，オーム社 (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・プログラミング法則集，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. 6 th，No. Pt 2, pp. 503-506 (1992).

(平成6年 6 月 22 日受付)
(平成6年 12 月 5 日採録)

原田　実（正会員）

1975年東京大学理学部物理学科卒業．1980年東京大学理学系大学院博士課程修了．理学博士．同年(財)電力中央研究所担当研究員．1989年より青山学院大学理工学部経営工学科助教授．自動プログラミングシステム，ソフトウェアの要求理解や設計の自動化，オブジェクト指向CASE，株式投資エキスパートシステム，自律ロボットシステムなどの研究を行う．1986年（財）電力中央研究所経済研究所所長賞受賞．1992年人工知能学会第6回全国大会優秀論文賞受賞．訳書「ソフトウェアの構造化設計法」（日本コンピュータ協会），編著書「自動プログラミングハンドブック」，（オーム社），監修書「CASEのすべて」（オーム社）共著書「知的プログラミング」（オーム社）など．IEEE，ACM，AAAI，電子情報通信学会，人工知能学会，日本ロボット学会，OR学会，経営工学会各会員．



吉川　彰一

1991年青山学院大学理工学部経営工学科卒業．1993年同理工学研究科経営工学専攻修士課程修了．1993年より日本電気会社㈱C&C応用ソフトウェア事業本部CASE事業部製品技術部．

永井英一郎

1992年青山学院大学理工学部経営工学科卒業．1995年同理工学研究科経営工学専攻修士課程修了予定．1995年㈱東芝入社予定．