ソフトウェア工学 87－1
（1992. 9. 29）

構造化分析／設計の方法論に基づいた プログラム理解支援ツール

四野見 秀明，藤井 邦和，牧野 正士，津田 和幸

日本アイ・ビー・エム 東京基礎研究所
東京都千代田区三番町 5 番地 11

あらまし

RE/Cycle は，COBOL，PL/I，JCL で書かれたプログラムの保守における理解を支援するツールである．既存ソフトウエアの中には，仕様書が存在しないもの，存在してもプログラムとの対応が取れなくなっているものが多い．本論文では，プログラムから構造化分析／設計で用いられている各種ダイヤグラムを生成し，ハイパーテキスト風のインタフェースにより，プログラム理解を支援することを提案する．また，プログラム資産の中には，構造化設計されていないもの，長年の修正の繰り返しの後，構造が崩れたものがかなり存在している．それらの理解のために自動的にプログラムを再構造化し，構造化された形式上でプログラム理解を行なうことも提案する．

和文キーワード リバース・エンジニアリング，構造化分析／設計，プログラム理解，ソフトウェア保守，再構造化

A Program-Understanding Tool Based on the Structured Analysis/Structured Design Methodology

Hideaki SHINOMI, Kunikazu FUJII, Seishi MAKINO, Kazuyuki TSUDA

IBM Research, Tokyo Research Laboratory
5-11, Sanbancho, Chiyoda-ku, Tokyo

Abstract

RE/Cycle is a tool that helps software maintainers to understand programs written in COBOL, PL/I, and JCL. Many existing programs do not have specifications, which are indispensable for program understanding. Even when specifications exist, they do not always correspond to the programs that they purport to describe. RE/Cycle automatically generates from programs the diagrams (specifications corresponding to programs) used in the Structured Analysis/Structured Design (SA/SD) methodology, and makes HyperText-like links between them. This paper proposes a method for program understanding based on automatic SA/SD diagram generation and HyperText-like links. It also proposes a method for program understanding based on automatic restructuring.

英文 key words Reverse Engineering, SA/SD, Program Understanding, Software Maintenance, Restructuring

1 はじめに

ソフトウェアに費やされる経費の中で，保守が占める割合が，全体の80%にもおよぶといわれている．その保守作業の中で半分はプログラム理解に費やされているという[1]．つまり，プログラムの理解を助けることが，ソフトウェア生産性向上への効果的なアプローチといえる．

一方，各種CASEツールの普及により，構造化分析／設計の方法論に基づいたダイヤグラムを用いて，ソフトウェアを設計することが一般的になりつつある．初めからCASEツールを用いて設計されたソフトウェアの場合は，設計の過程で書かれたダイヤグラム等が仕様書として残されていて，保守作業の際のプログラム理解にも役立っている．しかし，現実には，既存のプログラムのほとんどは，CASEツールを用いて設計されてはおらず，ツールの利用可能な形式での仕様書は残っていない．そこで，プログラムから仕様書を逆生成するというリバース・エンジニアリングが，注目されている[3]．

本論文では，既存のプログラムから構造化分析／設計で用いられるダイヤグラムを逆生成し，その上でプログラム理解に有効な情報を効果的に表示することにより，プログラムの正確な理解を支援する方法論を提案する．まず，2節で，プログラム理解の現状と提案するプログラム理解支援についてを述べる．次に，3節では，提案を実現したRE/Cycleについて述べ，4節で，その適用事例を紹介し，最後に，5節で今後の課題をまとめる．

2 プログラム理解

本節では，保守現場で実際に行なわれているプログラム理解と，それを踏まえたプログラム理解支援について述べる．

2.1 保守におけるプログラム理解と問題点

社内でプログラムの保守を行なっている部門への調査の結果，CASEツールで設計されていない既存のプログラムに対しては，以下のような手順で保守が行なわれていることが分った．また，幾つかの問題点が上げられた．

保守の手順として，以下のプロセスが行なわれる．プログラムの修正の必要性が発生した時，まず初めに，システムの構成に関する仕様書とバッチの入出力処理を表現する図（無い場合にはその場で手作成する）により，アプリケーション・システムの理解を行なう．その結果，修正すべきプログラムを発見すると，まず，プログラムの内部構造に関する仕様書により，サブルーチンの構成やインタフェースに関する理解を進める．その際，データに関する仕様書も必要に応じて参照される．詳細を検討すべきサブルーチンの当りがつけられたら，その内部の詳細なロジックとデータに関する仕様書，そして，プログラムを見ながら詳細な理解を行なっていく．これらの理解の流れと参照関係を表現したのが図1である．図中，実線の矢印が保守する者の視点の移動であり，点線の矢印が参照を表現する．このプロセスで用いられる仕様書は，ほとんどが手書きで作成されたものであり，仕様書とプログラムの整合性が取れなくなるという事態も生じている．また，各仕様書間の関係づけは，プログラム，サブルーチン，変数等の名前を頼りに紙の仕様書をめくらなければならないので，手間のかかる仕事になっている．

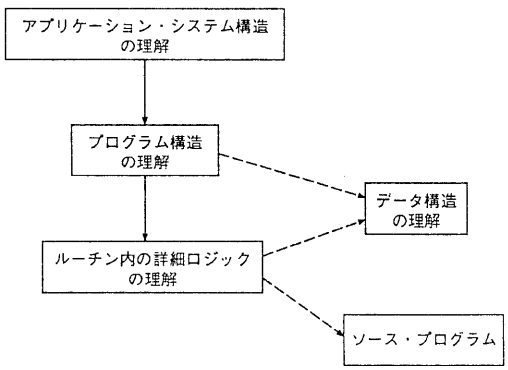

図1: プログラム理解の流れ

COBOL，PL/Iで書かれたプログラムには，構造化プログラミングの概念の無いうちに書かれたものや，初めは構造化されて書かれていても，長年，修正を繰り返す内にGOTO文が多用され，構造が破れてしまい，そのままでは非常に理解し難いものになってしまっているものが多いのも現状である．構造化されていないプログラムの制御の理解を容易にする手段が必要とされる．

また，プログラムの制御構造の理解も重要であるが，より困難なのは，データの流れの理解であるという指摘もあった．構造化されていれば，プログラムの制御構造は，ソース・コード上に明確に表現されているから比較的容易に理解できるが，データの流れを把握するのは，依然骨の折れる仕事である．例えば，あるサブルーチンを理解しようとする場合を考えよう．データの入出力が引数として実現されていれば比較的把握し易いが，大域変数に対する代入参照によりデータが渡されている時，通り得る全ての制御フローを追いながらデータ代入参照をチェックしなければならず．プログラム理解にとってはかなりの負担となっているという．データの流れを追う

手段の提供により，プログラム理解に対するかなりの支援となる．

2.2 プログラム理解支援

2.1節で述べた調査結果に基づいて，以下のようなプログラム理解支援が効果的であると考えた．

1図における理解の流れを観察すると，構造化分析／設計で行なわれている [4]，そして，それに伴う CASE ツールで採用されているトップダウンの設計過程が分かる．構造化設計では，アプリケーション・システムを実行プロセスとデータ・ファイルの関係として設計する．ある程度まで設計を詳細化したのち，実行プロセスの中身のプログラムの構造をプロセスの呼びだし関係の図として設計し，そのプロセスの詳細ロジックは，疑似コードの形式で設計する．そして，最終的にプログラムのコーディングとなるわけである．そこで，図1において，四角で表現された，仕様書もしくは理解すべき対象に相当するダイヤグラムを，ソース・プログラムから生成しする．そして，図1における矢印に相当する関係をハイパーテキスト風のリンクとして実現することにより，プログラム理解のために必要な関連情報を有機的に結びつけ，プログラム理解を支援することを提案する．ダイヤグラムは，構造化分析／設計に基づく CASE ツール等で利用されている馴染み深いものであり，リポジトリ等を経由して，フォワードの CASE ツールへ，ダイヤグラムの情報を渡すことも可能である．一度，フォワードの CASE ツールにダイヤグラムの情報が与えられれば，その環境でプログラムの修正，再開発等が行なえるようになる．

構造化されていないプログラムに対する理解支援の手段として，自動的なプログラム再構造化の機能が有効であると考えた．GOTO 文によって書かれた制御を IF-THEN-ELSE や LOOP の構造に自動変換した形式で，プログラム・ロジックを読んだ方が理解し易くなる．ここでは，無理矢理すべての GOTO 文を無くすことだけを目指すのではなく，プログラムの理解を容易にするためのという観点の再構造化が必要とされる．

また，データ・フロー解析を行ない，その結果を効果的に提示することで，プログラム理解の際のデータの流れを追う作業を支援することが有効であると考えた．具体的には，以下を支援することとした．

- 注目するデータの値が何処で決定され，次何処で使用されるかの把握（データに関する変更の影響の把握）
- サブルーチンで如何なるデータが処理され，結果として返されるかの把握（サブルーチン間のデータの流れの把握）

これらは，それぞれ，影響分析機能，ストラクチャ・チャート上のデータ・カップルの表示として実現されている．

3 RE/Cycle

本節では，RE/Cycle においてなされているプログラムの解析，解析結果の表示方法，そして，ツールを用いた際のプログラム理解について述べる．

3.1 システム構成

図2に RE/Cycle のシステム構成を示した．JCL からはシステム・フロー・チャートが生成される．COBOL，PL/I からは，構文解析から，プログラムの内部表現である代数表現が生成され，代数表現の再構造化，データ・フロー解析の後に，ストラクチャ・チャート，アクション・ダイヤグラム，データ・ストラクチャ・チャートが生成される．なお，再構造化とデータ・フロー解析は，オプションでありこれを外しても，すべてのダイヤグラムは，（一部機能は制限されることもあるが）表示可能である．

3.2 プログラムの内部表現

COBOL，PL/I のプログラムの制御構造の内部表現としては，言語に依存しない表現として代数表現（Algebraic expression [5][6]）を採用している．代数表現は，プログラムのコントロール・フローと，ステートメントやブロックの構造情報を表現する．代数表現を内部表現として用いることにより，RE/Cycle における再構造化機能を含む，いくつかの機能の実現における言語依存部分を最小限にとどめることができた．

[5] で提案された代数表現では，構造化されたプログラムしか表現することが出来なかったが，RE/Cycle では，拡張を行ない GOTO 文等の構造化から外れたステートメントに相当するオペレータを導入することで，構造化されていないプログラムについても表現可能にしている．本論文では，代数表現とその拡張についての詳しい説明は行なわない．

3.3 再構造化

再構造化では，GOTO 文で書かれた制御構造を自動的に，IF-THEN-ELSE や LOOP 等の構造に，プログラムのロジックを変えずに変換する．再構造化された後のロジックは，アクション・ダイヤグラム (3.5.3 参照) 上に表示される．再構造化された後は，GOTO 文に来る度にラベルを探しながらプログラムの制御を追う必要がなく，上から下へとすなおに，プログラムを読んでいけるので理解が容易になる．特に，GOTO 文によりループ構造が

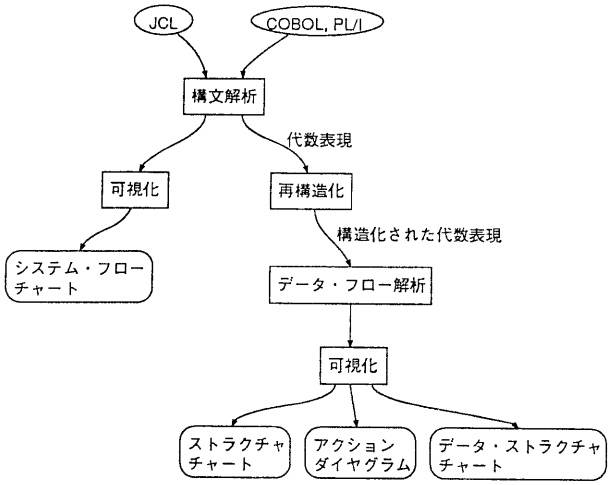

図 2: システム構成

作られている場合など，再構造化後は，明示的にLOOPとして表示されることにより理解しやすくなる．

RE/Cycle の再構造化機能の特徴は，オリジナルのプログラムの構造を出来るだけ保存することで，オリジナルのプログラムを見慣れているユーザが理解しやすい制御構造に変えることにある．COBOL については，既に自動的な再構造化を行なうツールは存在するが，ユーザからもいくつかの問題点が指摘されている．それらのコメントを踏まえ以下を実現することを目標とした．

- プログラムの構造を極端に変え過ぎない．
- 人工的なスイッチ変数の使用を極力さける．

これらを実現するために，いくつかの工夫を行なったが，その一つとして，サブルーチンをまたがるGOTO文は，あえてGOTO文のまま残した．サブルーチン間のGOTO文は，ほとんどの場合エラー処理であり，明示的に残した方がプログラムを理解し易い．そのかわり，アクション・ダイヤグラムのユーザ・インタフェースとして，GOTO文の飛び先をハイパーテキスト風に表示できる機能を付加することにした．また，ループ脱出を，従来の人工的なスイッチ変数への代入，参照を用いて表現するかわりに，アクション・ダイヤグラム上に，特殊なLEAVE文を導入することによって表すことにした．その他，構造化アルゴリズムをサブルーチンを極力生成しないものとした．

現在のRE/Cycleでは，再構造化機能はプログラムの制御の理解を容易にするためのものであり，構造化後のロジックをアクション・ダイヤグラム中で見るために用いられ，コンパイル可能なプログラムを生成するまでは行なっていない．

3.4 データ・フロー解析

データ・フロー解析では，COBOL，PL/Iのプログラム中の代入文やサブルーチン呼びだし等の解析により，変数の値が何処で定義（Define）され，何処で使用（Use）されているかの情報を静的に分かる範囲で，プログラムから抽出することを行なう．ここで言う「静的」とは，プログラム実行時にしか分からないもの以外は全てサポートするいうことを意味する．データ・フローの解析結果は，影響分析（3.5.5参照）の実行と，データ・カップル（3.5.2参照）を表示するために用いられる．

3.5 プログラム情報の可視化

ここでは，RE/Cycleによって生成表示される各種ダイヤグラムと影響分析機能について述べる．本節で，用いられているダイヤグラムの例は，すべてRE/Cycle COBOL版の出力結果である．

3.5.1 システム・フロー・チャート

システム・フロー・チャート（図3参照）はJCLのプログラムの解析から，データ・ファイルと実行モジュール間のデータの流れを表示する．これにより，アプリケーション・システムの構造の理解が出来る．

ディスクの形で表現されているのがデータ・ファイル，四角が実行モデュール，カードがJCL内で設定されるパ

ラメータであり，データの流れは矢印で表現されている．また，モジュールからモジュールに引かれた線は，実行によるリターン・コードによって，次のモジュールが実行されるか否かが決まるという実行の依存関係を表現している．図3の例では，実行モジュールが2つあり，上側のモジュールは，パラメータ2つとファイル6つを入力とし，8つのファイルにデータを出力している．また，上側のモデユールから下側へと実行の依存関係があることが分かる．

システム・フロー・チャート上で，実行モジュール上をクリックすることで，そのプログラムについての，ストラクチャ・チャートが生成表示される．

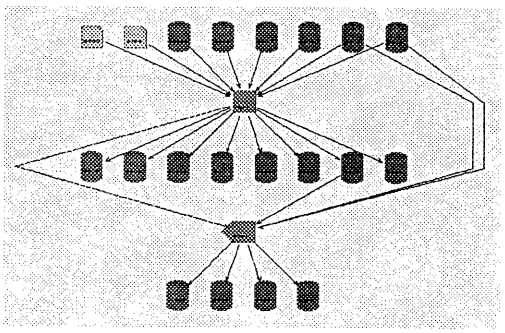

図 3: システム・フロー・チャート

3.5.2 ストラクチャ・チャート

ストラクチャ・チャート（図4参照）は，COBOLもしくはPL/Iプログラム中のサブルーチンの呼びだし関係と，それに伴うデータの流れを表示する．これにより，個々のサブルーチンのインターフエースと，プログラムの構造の理解を進めることが出来る．

図中，四角はサブルーチンを示しており，内部ルーチン，外部ルーチン，再構造化によって作られたルーチンは，色の違いにより表現される．矢印は呼び出し関係を表現しており，矢印についた黒いダイヤモンドは，呼び出しが生じるかどうかは，条件によることを示している．矢印の中程についた小さな楕円は，呼び出しが繰り返されることを表している．呼び出しを表す矢印の横に，小さな円のついた小さな矢印（⇩）は，データ・カップルと呼ばれ，サブルーチン・コールに伴うデータの流れを示す．データ・カップルを選択すると，流れるデータの変数名を知ることができる．これらデータの流れは，サブールーチン・コールの引数によるものと，大域変数への代入／参照によって生ずるものがあり，前述のデータ・フロー解析の結果により表示される．

ストラクチャ・チャート上で，内部ルーチンに相当する四角をクリックすることて，相当するルーチンに対するアクション・ダイヤグラムが表示される．

3.5.3 アクション・ダイヤグラム

アクション・ダイヤグラム（図5参照）は，ルーチン内の詳細なロジックを表示する．また，ステートメントを注釈で置き換えてゆくことにより，疑似コード形式の仕様書へと変換していくことができる．

表示の仕方は，基本的には，CASEツールで広く使用されるアクション・ダイヤグラムと同じである．図中，左側に表示されている鉤括弧（[）状のものは，ルーチン，IF-THEN-ELSE，LOOP等のスコープを表し，楕円状のものは，サブルーチン呼び出しに相当する．RE/Cycleにおいて，アクション・ダイヤグラム内に表示された再構造化をしたロジック上でプログラム理解を進め，最終的に，オリジナルのプログラムのどの箇所に相当するかを見たい場合，ソース・プログラムを別ウインドウに表示（ソース・コード・ブラウザ）し，アクション・ダイヤグラムのステートメントをマウスでクリックすることにより，相当するプログラムの箇所を表示できる．図中，2行目の「変数の初期化」と表示されているが，これは，もともと5行程度の代入文を注釈機能により，一行の注釈として置き換えたものである．また，RE/Cycleでは，変数名と日本語の対応表を予め用意することにより，アクション・ダイヤグラム中で変数名の代わりに，日本語の記述を表示する日本語辞書機能も有する．これは，注釈を行なって行く際に，非常に助けとなる．

アクション・ダイヤグラム上で，変数名をマウスでクリックすることにより，データ・ストラクチャ・チャートが，生成表示される．

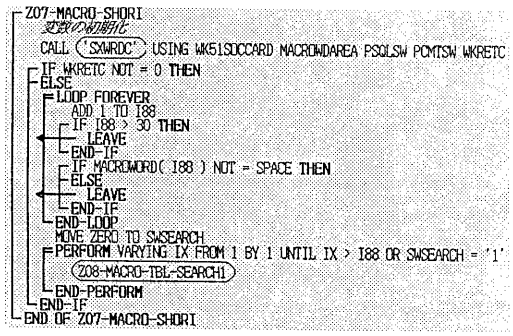

図 5: アクション・ダイヤグラム

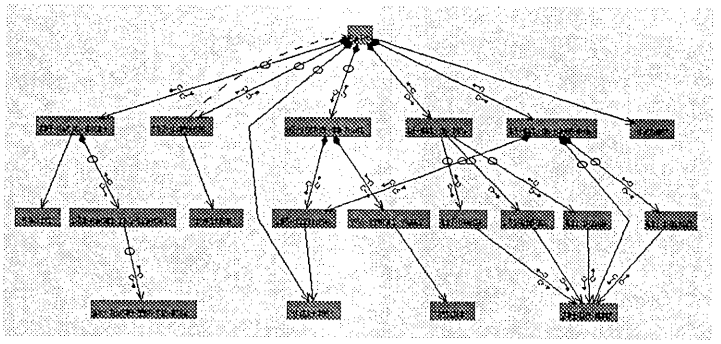

図 4: ストラクチャ・チャート

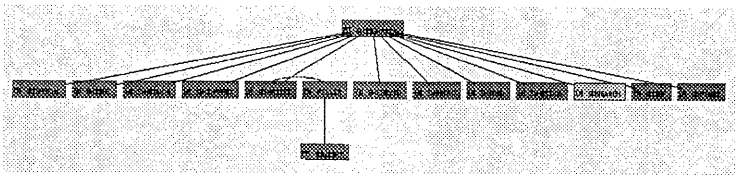

図 6: データ・ストラクチャ・チャート

3.5.4 データ・ストラクチャ・チャート

データ・ストラクチャ・チャート（図6参照）は，データ構造の論理的な側面を図に表現しており，主にデータの階層関係を表現している。図中、四角はデータ項目であり、直線で結ばれた上下は、階層の上下関係に相当する。配列は通常のデータと箱の形を変えることで区別している。Jackson 法で用いられるデータ構造ダイヤグラムと同じ情報を表現しているとも言える。また，一つのデータ領域を同じ名前で定義し直している場合（COBOL の REDEFINE，PL/I では DEFINED) は，矢印付きの円弧で表し，図上で明示している．

3.5.5 影響分析機能

RE/Cycle は，プログラム内の詳細な影響分析を行なう機能を有する．ここで言う影響分析とは，プログラム内のある変数の値に関する意味の変更を行なった場合，その変更の影響の及ぶ範囲を効果的に提示する機能を意味している．

影響分析機能は，アクション・ダイヤグラム中の変数，またはデータ・カップルの変数について起動され，その変数の値が何処で定義（Define），もしくは，何処で使用（Use）しているか，また，その変数に対する何らかの変更により，影響を受け得る全ての箇所をアクション・ダイヤグラム，ストラクチャ・チャート等上で明示する機能を持つ．例えば（図7参照），アクション・ダイヤグラム中のある代入文の中で，変数 X にある値が代入（Define）されているとしよう．その代入された値が，次に何処で使用（Use）されているかを知りたければ，その変数をマウスで選択して，Use 発見の操作をすることにより，その値が使用される（Use）箇所の変数 X は，アクション・ダイヤグラム中で赤い反転により明示される．それと同時に，ストラクチャ・チャート上でその箇所を含むルーチンは，四角の色が緑から赤に変化することにより明示される．そこで発見された Use 変数 X の値が，その箇所での代入文により，他の変数 Y に代入されている場合，変数 X に対する何らかの変更は，この変数 Y への代入文により，影響が伝播されることになる．これらの影響の伝播を全て表示することも可能である．その場合，アクション・ダイヤグラムとストラクチャ・チャート上で，影響の生じ得る箇所は，全て赤の反転で表示されることになる．それにより，プログラム修正の時に，チェックすべき必要最小限の箇所を知ることができる．

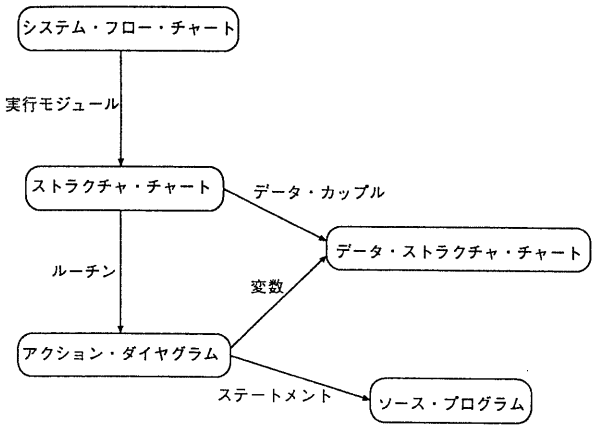

図 8: RE/Cycle における各ダイヤグラム間のリンク

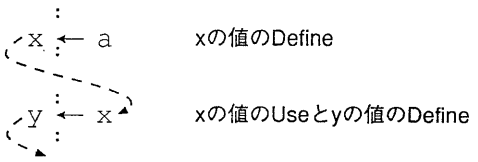

図 7: Define/Use 連鎖

3.6 RE/Cycle を用いたプログラム理解

2節で述べた保守におけるプログラム理解の流れを自然に実現するために，図1における関連付けは，RE/Cycle では図8のように，ツールにより生成されるダイヤグラム間のリンクとして実現されている．図1中の四角で表された理解すべき対象は，図8では，それぞれ，角の取れた四角のなかに書かれたダイヤグラムに相当している．矢印の横に書かれたものは，ダイヤグラム中のオブジェクトを示しており，そのオブジェクトをマウスでクリックしたときに，矢印の先のダイヤグラムが，表示されることを示している．

アプリケーションの構造を理解するために，JCL を解析して得られるデータ・ファイルと実行モジュール間のデータの流れを表現するシステム・フロー・チャートが用いられ，特定の実行モジュールを選択することにより，相当するプログラムのサブルーチンの呼びだし関係，インタフェースを表現するストラクチャ・チャートが，生成表示される．ルーチン間のデータの流れを表現するデータ・カップル中の変数を選択することにより，そのデータ構造をデータ・ストラクチャ・チャートとして表示，インタフェースのより深い理解を支援する．ストラクチャ・チャートにより，プログラム構造を理解し，調査すべきサブルーチンが分かったところで，それを選択し，アクション・ダイヤグラムを生成表示することで，ルーチン内の詳細ロジックの理解を進める．その際，必要に応じて変数に関するデータ構造を参照したり、影響分析機能により、データが如何に使われていくかの理解を深める。また，必要があるときは，アクション・ダイヤグラム中のステートメントに対応するソース・プログラム中の箇所を参照する．このような流れで、RE/Cycle を使用した時のワークステーション上の画面を図9に示す。ダイヤグラム上のオブジェクトを選択して、生成された各ダイヤグラムのウインドウが表示されている。

こうして，プログラムの理解を進めながら，理解の経過や結果を各ダイヤグラム中のオブジェクトに対する注釈として残してゆき，後の保守の際のプログラム理解の有益な情報とする．

4 適用事例

RE/Cycle の有用性を評価するために，某信託銀行で実際の業務プログラムをサンプルとして，業務システムの保守をされている担当者数人に実験を行っていただいた．

実験は次の二つの観点から行なわれた．

- 通常の保守作業の中で，プログラム理解のために RE/Cycle を使用する
- 構造化されていないプログラムの再構造化処理に RE/Cycle を使用する

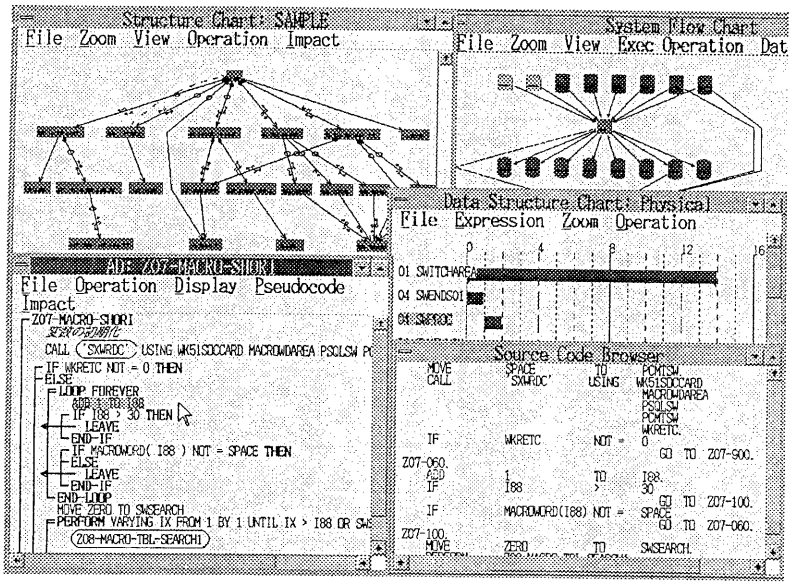

図 9: RE/Cycle 使用時の画面

使用したプログラムは構造化されていない PL/I プログラム 4 本（これらは，アセンブラで書かれたものを変換ツールを利用して，PL/I に変換したものであり，多数の GOTO 文を含む俗に言うスパゲッティなコーディングになっている）と構造化されている PL/I プログラム 5 本である．合計 9 本のプログラムに対して，プログラムを理解した後，構造化や修正作業などを行った．

まず，プログラム理解ツールとしての RE/Cycle の評価として次の結果が得られた．

- 難解なプログラムを理解する場合には，ツールは有効である．特に経験の浅い若手ほど効果が大きい．逆に，単純なプログラムの場合にはツールを使用しても使用しなくても生産性はさほど変わらない．
- データ項目の桁数を修正するような場合，影響分析機能によりチェックすべき箇所が限定されるので有効である．また，データの使われ方を理解するのにも有効である．
- 構造化されていないプログラムの理解には，再構造化機能が有効であるが，構造化されたアクション・ダイアグラムで理解した後，保守対象である元ソースを見て修正をしなければならないのは，二重感がある．

次に，再構造化ツールとしての評価を以下にまとめる．なお，ここでの再構造化とは，構造化されたアクション・ダイヤグラムを見ながら，元プログラムを手作業で構造化する作業である．

- ツールを使用することで，難解なプログラムの再構造化処理を経験の浅い若手でもベテラン並の時間で完了できた．
- ツールを使用することで，構造化されたコードの品質を作業員の経験度に依らず一定の水準にすることが可能になる．
- ツールによる変換だけでは限界があり，更に理解しやすいコードにするためには，手作業による後修正が必要であることが指摘された．

以上の評価により，RE/Cycle は複雑なプログラムほど効果が大きく，その使用により，経験の浅い若手がベテラン並の生産性を達成することを可能にすることが明らかになった．また，構造化したコンパイル可能（実行可能）なコード生成の機能への要求は，アセンブラから変換された PL/I プログラムのみならず，通常の PL/I プログラムに対しても大きいことが分かった．

また，今回作業していただいた作業員の方は，日常の作業ではリスティングを使用し，机上で解析や修正を行っており，ウィンドウ・システムを使用するのは初めてあったので，RE/Cycle のようなインタラクティブな環境に

慣れていくことにより，ツールを使用した作業の生産性が，さらに向上することが期待される．

5 おわりに

現在，RE/Cycle では，ホスト・コンピュータから，COBOL，PL/I，JCL のプログラムをダウンロードし，解析，表示すべてを IBM PS/55 OS/2 バージョン 1.3 上で行なっている．

これまで，プログラム理解を支援するツールとして，RE/Cycle のプロトタイプを作り，保守現場での使用を通してフィードバックを得てきた．それらに基づき，いくつかの RE/Cycle の拡張が考えられる．

研究の当初は，PL/I は COBOL に比べて新しい言語であることから，ほとんどのプログラムは，構造化プログラミングされているのではないかと考えていたのであるが，銀行，保険，証券会社に対する調査の結果，現状では，構造化されていない PL/I プログラムの資産がかなりあり，保守の障害になっていることが分かった．PL/I プログラムにおいても，再構造化機能が有効なプログラムがかなり存在することが分かり，再構造化されたコンパイル可能なコードを生成することに対する要求が大きいことが明らかになった．つまり，4節で述べたアセンブラから変換された PL/I という特殊ケースのみならず，もともと PL/I で書かれたプログラムも再構造化して実行可能なコードを生成し，その後は，構造化されたプログラムを保守していきたいという要求である．よって，再構造化後のロジックをアクション・ダイヤグラム上で理解し，修正箇所を発見した後，オリジナル・プログラムの相当する箇所を表示修正していくという現在の RE/Cycle が持っている機能（プログラムの自動的な変換を好まないユーザも多く，その場合はこの機能が有効である）に加えて，コンパイル可能なコードを生成する機能を付加していく予定である．

現在，ほとんどの CASE ツールでは，プログラムの上流をデータ・フロー・ダイヤグラムを用いて設計していく．プログラムからデータ・フロー・ダイヤグラムを生成できれば，より上流の仕様に対するリバース・エンジニアリングが可能になる RE/Cycle が現在行っているデータ・フロー解析を基にして，実行モジュール間やサブルーチン間のデータの流れをデータ・フロー・ダイヤグラムの形式で表示し，ユーザとのインタラクションにより，より抽象化されたデータ・フロー・ダイヤグラムへと変換していくことを目指したい．

参考文献

[1] R. K. Fjeldstad and W. T. Hamlen, *Tutorial on Software Maintenance.*

[2] 保守地獄脱出へ挑戦始まる，日経コンピュータ，1990/7/30.

[3] *Special Issues on Maintenance, Reverse Engineering and Design Recovery*, IEEE Software, Jan., 1990.

[4] J. Martin and C. McClure, *Structured Techniques*, Prentice-Hall.

[5] A. Cimitile, U. deCarlini, G. Cantone, *Programs, Graphs and Metrics*, DIS-CSCI Technical Report No. 78, University of Naples, 1988.

[6] P. Benedusi, A. Cimitile, and U. de Carlini, *Reverse Engineering Processes, Design Recovery and Structure Charts*, CRIAI Technical Report, May 1990, Italy.