White Paper
Micro Focus Enterprise Analyzer

基幹システムアプリケーション資産の可視化･最適化ソリューション

プログラミング言語 COBOL は、メインフレーム時代からの長い歴史を持つ基幹システムの中で企業のビジネスの根幹を支え続けてきました。 現在も新たなビジネスを創出するアプリケーションの新規開発でも幅広く活用されています。
一方、長年のビジネス変化に対応し続けてきた COBOL アプリケーションは、度重なる変更・保守の結果、複雑さを増し、それに伴う弊害を発生することがあります。 もっとも深刻な弊害は、仕様書が存在しない・または存在していても現状に合ってなく、これらを把握するために工数をかける必要があるという問題で、この状態では些細な変更ですら重大なデグレードを引き起こす危険性があります。 そこまで深刻ではなくとも、複数の観点の異なる改修を施された結果のプログラムには、様々な品質上・性能上の問題点が潜在化することがあります。

MICRO FOCUS

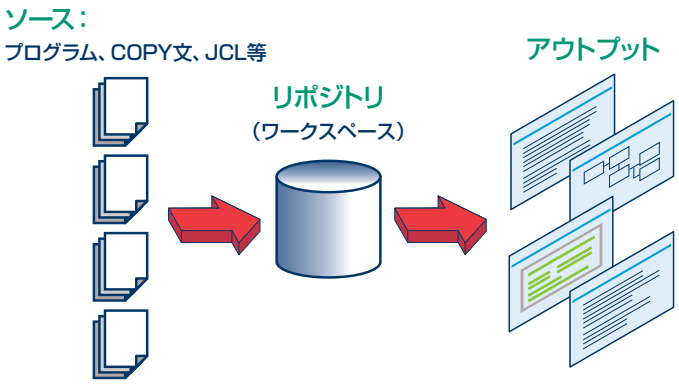
目次

Enterprise Analyzer を活用した COBOL アプリケーションのブラッシュアップ

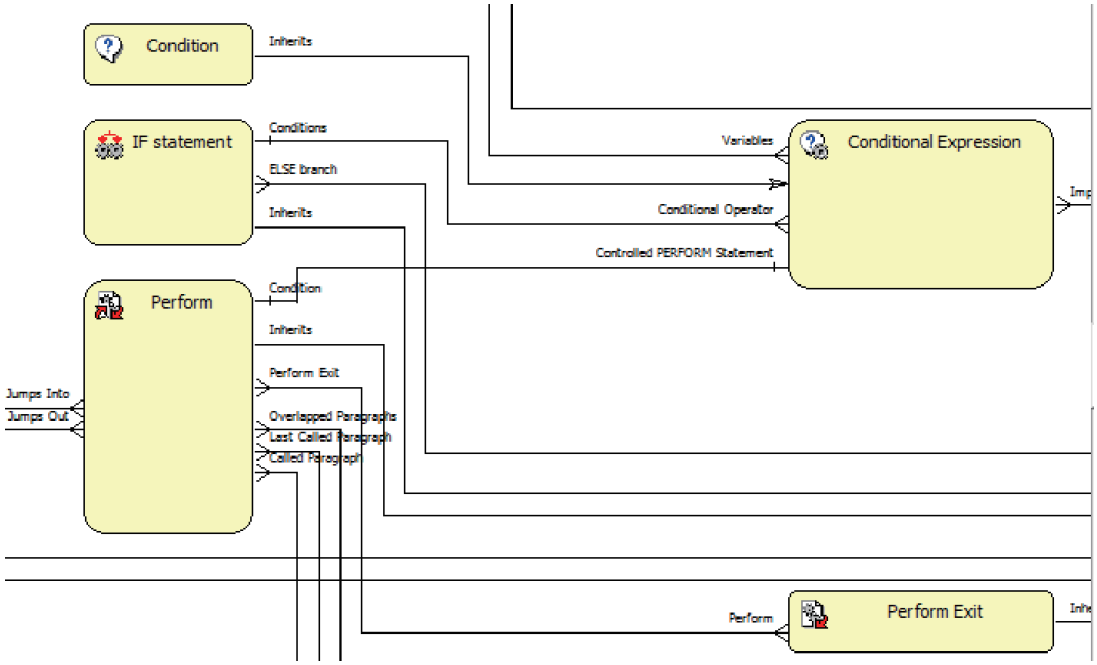
Micro Focus Enterprise Analyzer は、COBOL コードのリバースエンジニアリングツールとして、特にメインフレームアプリケーション対応で長い実績がある製品です。 多種多様な機能を提供する中で、潜在的な品質・性能上の問題点を自動検出する機能も装備しています。これを活用することによって現役で活躍しているアプリケーションのコードをブラッシュアップし、将来に渡ってさらに活用して行くための準備を行うことができます。 本書ではこのような点を中心にした Enterprise Analyzer の活用シナリオを紹介します。

1. Enterprise Analyzer のアーキテクチャ

Enterprise Analyzer は COBOL, PL/I などのプログラムソースのみならず、JCL、CICS リソース定義、IMS システム生成マクロなどを総合的に構文解析し、その関連を正確にリポジトリ化します。このリポジトリを利用して、一覧表、グラフ、チャートなど様々な形式のアウトプットを生成します。

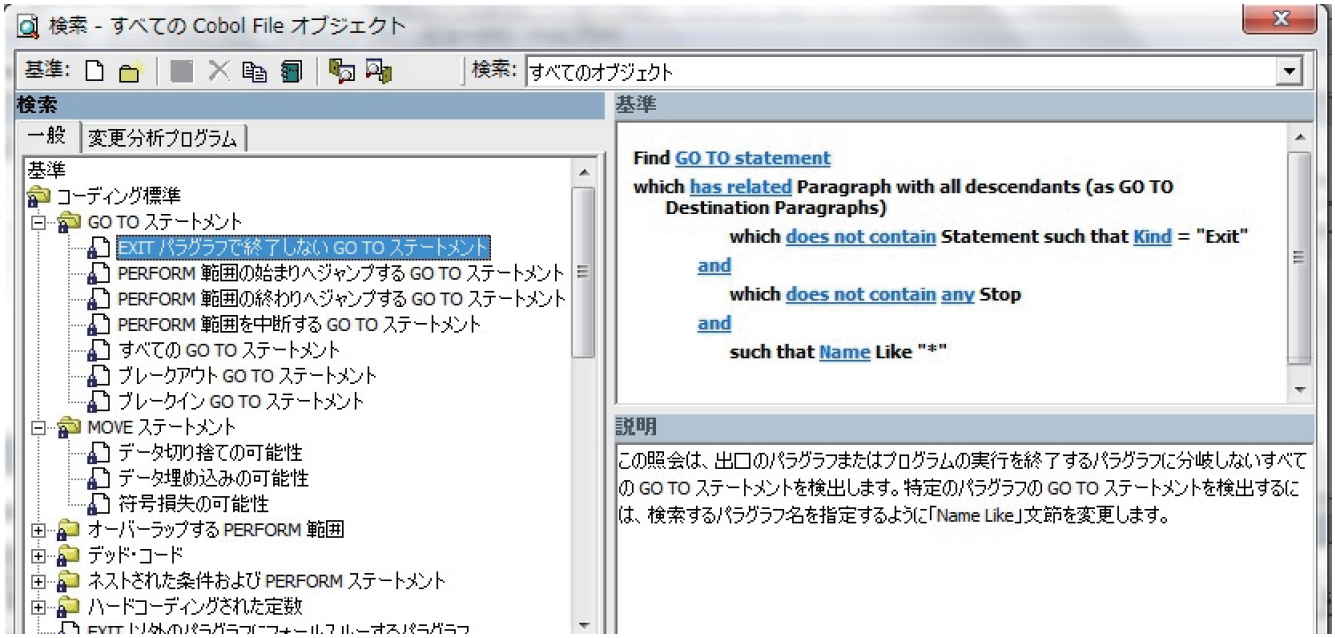


リポジトリのデータモデルは Enterprise Analyzer が規定するエンティティ・リレーションシップ型のスキーマとなっており、完全に公開されています。以下にその一部分の抜粋を示します。



アウトプットは様々なカスタマイズ手段が用意されており、目的と用途に応じて最適な形に整理された情報を得ることができます。特にClipperと呼ばれるクエリー機能は、COBOL構文を構文解析した結果のデータベースに対して直接照会することができるため、膨大な量のソースコードの中からある固有のパターンを自動検出するのに役立ちます。

出荷時設定で提供済みの豊富なクエリーのライブラリが用意されていますが、以下のように固有の構文でスキーマに対する複雑な照会条件を柔軟に記述して行くことができます。作成されたクエリーはエクスポート･インポート機能を使用して他のEnterprise Analyzerユーザーに展開して行くこともできます。



Clipperクエリー

以下に基本的なクエリー機能を示します；

- 項目名、プログラム名、ファイル名のパターンマッチ
- 項目・プログラム・ファイルの使用箇所と使用モード（照会・更新）
- 項目の長さ、十進桁数（整数部・小数部）、USAGE種別
- 節・段落名のパターンマッチと、GO TO・PERFORMでの使用箇所

Clipperが自動検出した問題箇所は、HTMLやExcelシートの形式でレポートとして保存することができ、Enterprise Analyzerを使用しない方に配布可能な文書として再利用することができます。以下の章でこのクエリー機能を活用して抽出することができるプログラムの問題箇所の例を解説します。

2. 品質向上

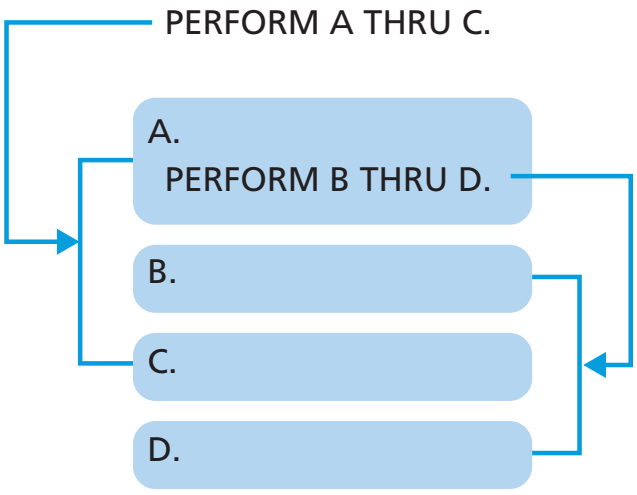
プログラムの品質問題にはいくつかのカテゴリーがあります。不正なプログラミングによってアプリケーションの出力結果に誤りがあるようなケースは問題が顕在しますので発見されやすいものです。次に、COBOL 言語仕様上正しくないプログラミングであるにもかかわらず結果的には出力結果には問題が無いというケースもあります。そのような箇所は、現状結果的に問題がなくても、プログラムの改修・再コンパイル・移植などのタイミングで問題が顕在化する可能性があります。以下のようなケースがこれに該当します。

●変数を初期化せずに使用する

COBOL では VALUE 句を書かないデータ項目の初期値は言語として規定されていません。COBOL コンパイラの実装によって規定されることがあり、Micro Focus COBOL ではデフォルトで空白（X’ 20’ ）で初期化されます。このようなコンパイラの種類に依存する仕様を仮定したプログラミングは、現在は問題なく稼動していても将来の再コンパイルや移植で問題が顕在化する可能性があり、品質上好ましくありません。

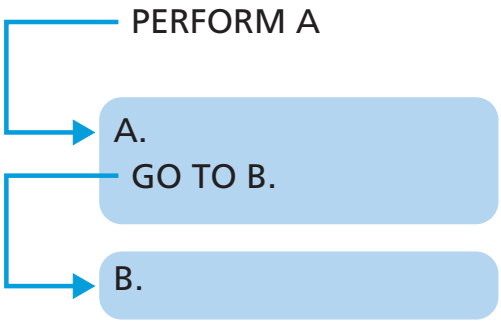
●複数の PERFORM 文でその実行対象範囲に重なりがある

COBOL の PERFROM A THRU B 文法は、引き続く複数の節・段落をまとめて PERFORM するものです。この実行対象範囲が重複するような複数の PERFORM 文があると、その動作は言語仕様上規定されておらず、コンパイラの実装によって異なります。



● GO TO などによって明示的に終了しない PERFORM 文

COBOL では PERFORM 文による手続き実行は実行範囲の最後の文で明示的に終了しなければならない規則となっています。PERFORM 文による実行範囲の中から GO TO 文で外部へ分岐することは許容されていますが、必ず再び実行範囲に戻って終了しなければなりません。終了しないまま処理を続行していると COBOL ランタイムはまだ PERFORM 文を実行中の状態で処理を行うためプログラマの想定したのと異なる動作になることがあります。



● FILE SECTION ㊥に書かれた VALUE 句

COBOL では VALUE 句の記述は WORKING-STORAGE でのみ意味があり、FILE SECTION では書かれても無視される規則となっています。しかしコンパイラによってはこれを有効とするものがあり、これを仮定して記述されたプログラムはたまたま動作しているに過ぎません。
また、言語仕様上不正ではないがプログラミング作法上問題があり、将来にわたる保守作業の品質に悪影響を与えうるものとして以下のようなケースがあります。

●決して到達することの無い実行文（デッドコード）

STOP RUN 文の直後に書かれた実行文が代表的なデッドコードですが、そのほかにも例えば以下のような書き方でデッドコードは発生します。

```
PERFORM A
PERFORM B
STOP RUN.
```

上の例では段落 C はその直前の段落 B からのフォールスルーで実行される可能性があるように見えますが、段落 B が PERFORM 文でしか実行されていないためフォールスルーがありえません。
デッドコードは、出口ラベルのような作法として記述する場合に必然性を持って発生することがありますが、そうではなく単にプログラム修正を重ねた結果として発生するケースが多く、これはテストカバレッジの達成や保守性の観点から好ましくありません。

●使用しないデータの宣言（デッドデータ）

デッドデータもプログラム修正を重ねた結果既に使用されなくなったデータ項目が削除されずに残っている状態で発生します。保守性観点のみならず実行時に消費するメモリも発生していますので好ましくありません。ただし、汎用的な COPY メンバーの中で宣言された項目のうちの一部だけを使用するといったケースは問題とはなりません。

●深すぎる入れ子の制御構造

IF 文などの条件文の入れ子が深すぎるプログラムは可読性を低下させ、バグ混入の原因となります。一般的なコーディング規約では 5 階層以上の場合は PERFORM 文などで外出しにすべきであると言われています。また、複雑なデシジョンテーブルは IF 文で実現せずに EVALUATE 文を駆使すると整理された形で記述することができます。

3. 性能向上

どのようなプログラミング言語でも、ある同じ結果を得るためのプログラミングの方法は複数存在します。このとき、COBOL 言語の特性、または弊社のコンパイラの特性として実行性能の良い方法と悪い方法があります。このような性能の悪い方法でプログラミングされた部分は、実行結果に問題がないため多くは顕在化しません。しかし、このような箇所を改善することでアプリケーションの処理速度を高速化できる可能性があります。劇的な改良ではなくとも、もともと長時間かかっているバッチ処理などの場合には効果が目に見えるものとなり得ます。

以下のようなケースが悪い方法に該当します。

●表参照の添え字付けにゾーン十進項目を使用
ゾーン十進形式は算術演算や表参照の添え字付けで使用するにはもっとも効率が悪い形式です。添え字付けは INDEXED BY 句で宣言した指標を使用するのがもっとも高速です。特に意味が無ければ指標以外の項目を使用するべきではありません。

●内部使用の作業用数字項目にゾーン十進項目を使用
カウンタ変数や複合的な計算処理の一時結果を保持するだけの目的で宣言された作業用数字項目は、画面や帳票に送られることがないためゾーン十進で宣言する意味はありません。最も効率的な COMP-5 項目などを使用するべきです。

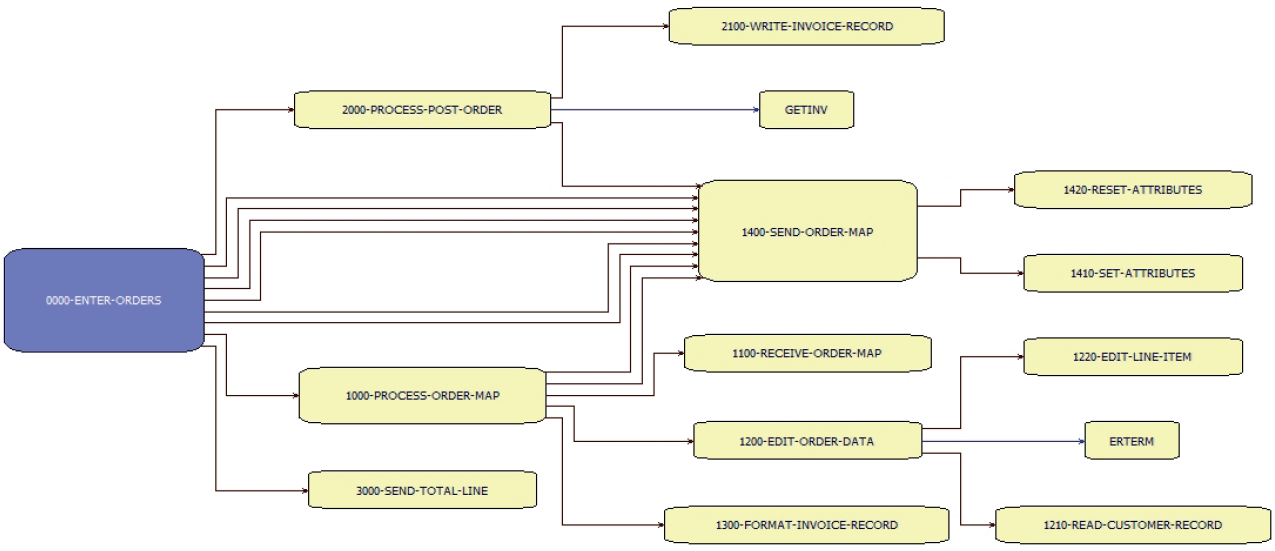
●基本項目の初期化に INITIALIZE 文を使用
INITIALIZE 文は集団項目の初期化を 1 文で実行するときの可読性・保守性を高めるものですが、実行性能は必ずしも良くありません。特に対象が基本項目である場合は INITIALIZE 文を使用するメリットはないため MOVE 文による初期化を使用するべきです。

●集団項目中のバイナリデータ項目が語境界に整列されていない
ほとんどの CPU ではバイナリデータのレジスタへのロード･ストアは 4 バイトまたは 8 バイト境界に整列されているときが最も効率的です。また、RISC マシンでは整列されていないデータのロード・ストアは許容されていません。このため C 言語などではコンパイラによる自動的な整列が行われますが、COBOL はデフォルトではそれを行いません。語境界に整列されていないバイナリ項目に対する操作ではコンパイラが余分なコードを生成しており、効率が低下します。

● Intel プラットフォームで COMP（ビッグエンディアン）を使用している
COBOL では COMP 項目はプラットフォームに依存せずにビッグエンディアン表現となっており、リトルエンディアンがネイティブの Intel CPU では効率が低下します。特に理由が無い限り COMP-5 を使用するべきです。

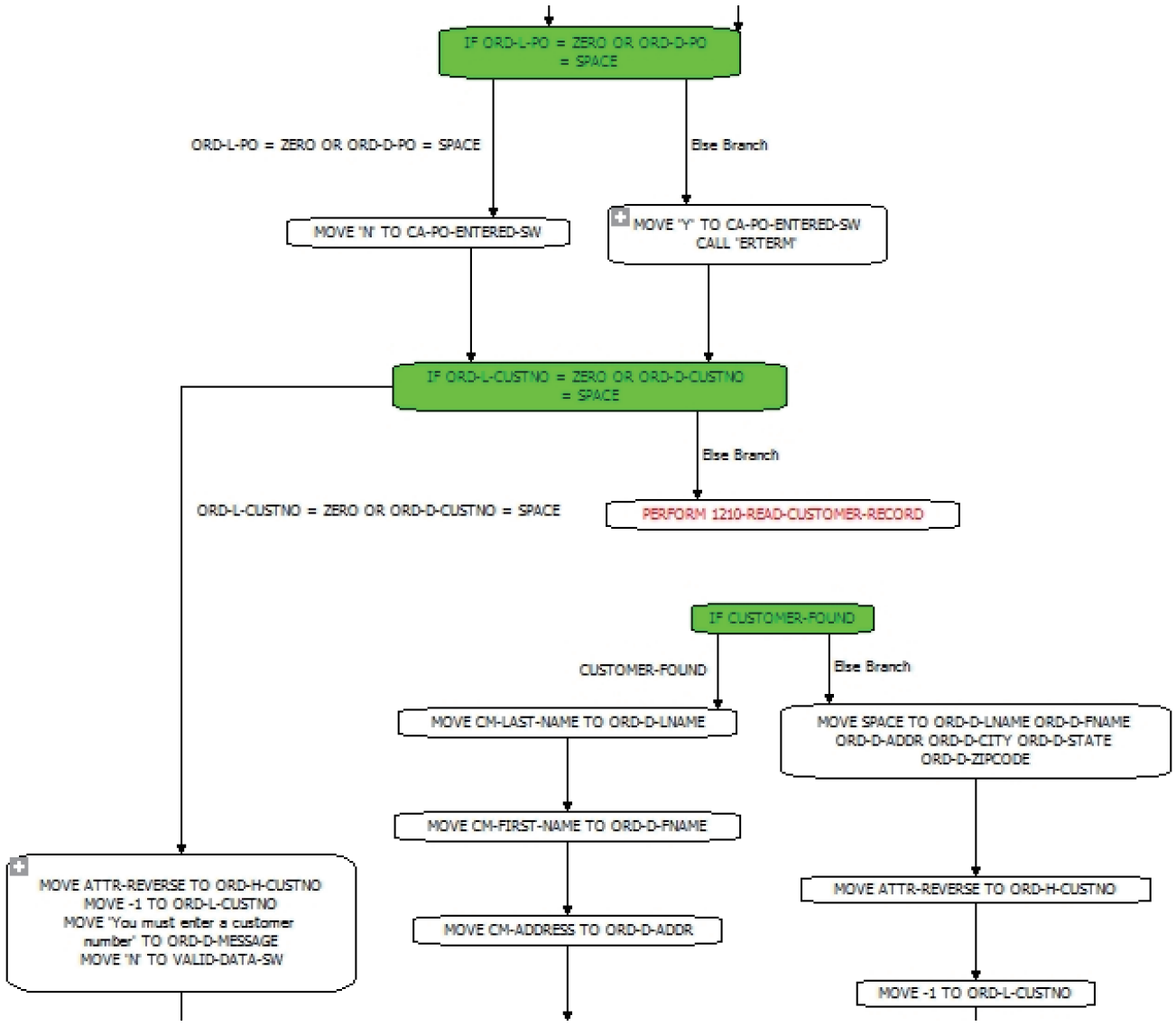
4. 可視化
ここまでは、「潜在的な」問題箇所を自動検出するという点に着目して事例を挙げてきました。このほかにも Enterprise Analyzer は広範な機能を提供しており、リバースエンジニアリングツールの主目的である可視化でも活用することができます。可視化に対するニーズは、仕様書が整備されていないアプリケーションの保守で現れ、Enterprise Analyzer が以下のような場面で役立ちます。

1) プログラムフロー
プログラムの手続き部の中での PERFORM 文や GO TO 文を経由した処理のフローをグラフィカルに図示します。



2) フローチャート

現在着目している節・段落に対してその中の実行文の分岐構造をフローチャート化して表示します。



3) CRUD レポート

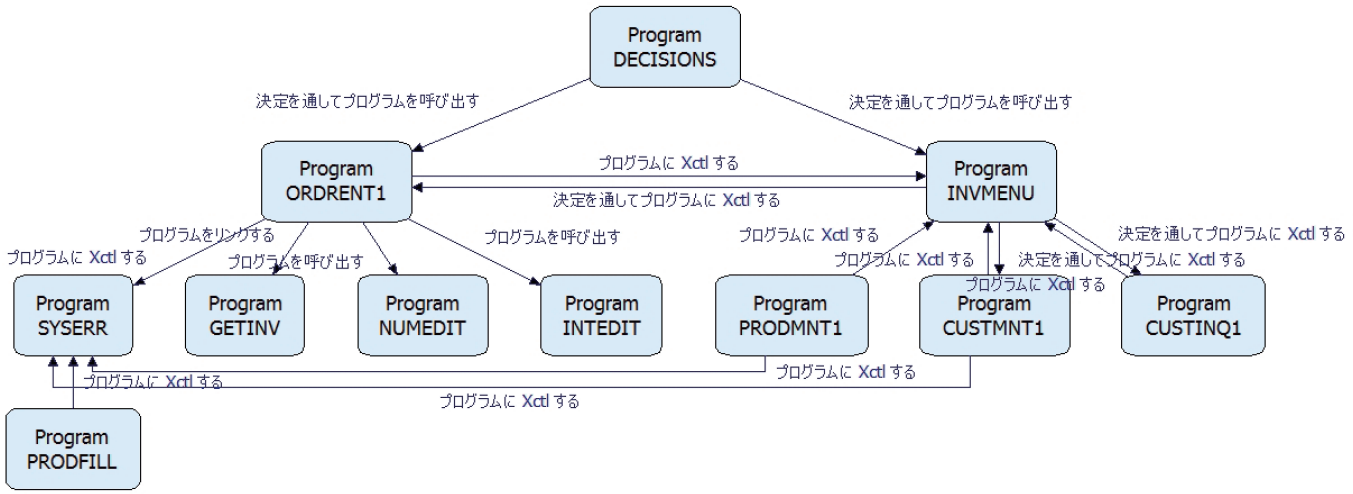
プログラムとファイル・データベーステーブルの間のクロス参照関係を一覧表示します。特にデータのライフサイクルを示すため Create(作成) / Read(読み取り) / Update(更新) / Delete(削除) のオペレーションを区別して表示しますので、単にアクセスしているだけの関係ではなく、関係の種別に応じた検索が可能となります。

CRUD レポート - IBMDEMO (IBMDEMO)

プログラム	ファイル名	データ・ストア	データ	タイプ	作成	読み取り	更新	削除
AR7100	AR7100.CBL	DD01.CUST.MASTER	CUSTMAS	File		+	+	
AR7100	AR7100.CBL	DD01.CUST.MASTER.SORTED	CUSTMAS	File		+	+	
AR7100	AR7100.CBL	DD01.CUST.MASTER.SORTED	UPDATES	File		+		
AR7100	AR7100.CBL	DD01.PROD.MASTER	UPDATES	File		+		
AR7200	AR7200.CBL	DD01.PROD.FORECAST.MASTER	FCSTMSTR	File		+	+	
AR7200	AR7200.CBL	DD01.PROD.GENERAL.UPDATES.SORTED	UPDATES	File		+		
AR7200	AR7200.CBL	DD01.PROD.SECURITY.MASTER	FCSTSEC	File		+	+	
AR7300	AR7300.CBL	DD01.PROD.FORECAST.MASTER	FCSTMSTR	File		+	+	
AR7300	AR7300.CBL	DD01.PROD.MASTER	CATLOGMS	File		+	+	
AR7300	AR7300.CBL	DD01.PROD.OUTHOURS.UPDATES.SORTED	UPDATES	File		+		
CUSTINQ1	CUSTINQ1.CCP	DD01.CUST.MASTER	CUSTMAS	File		+		
CUSTMNT1	CUSTMNT1.CCP	DD01.CUST.MASTER	CUSTMAS	File	+	+	+	+
DECISIONS	DECISIONS.CBL		DATA.TXT	File		+		
GETINV	GETINV.CCP	DD01.INV.CONTROL	INVCTL	File		+	+	
INTEDIT	INTEDIT.CBL		PS_ID_C	Table		+		
ORDRENT1	ORDRENT1.CCP	DD01.CUST.MASTER	CUSTMAS	File		+		
ORDRENT1	ORDRENT1.CCP	DD01.INVOICE.MASTER	INVOICE	File	+			
ORDRENT1	ORDRENT1.CCP	DD01.PROD.MASTER	PRODUCT	File		+		

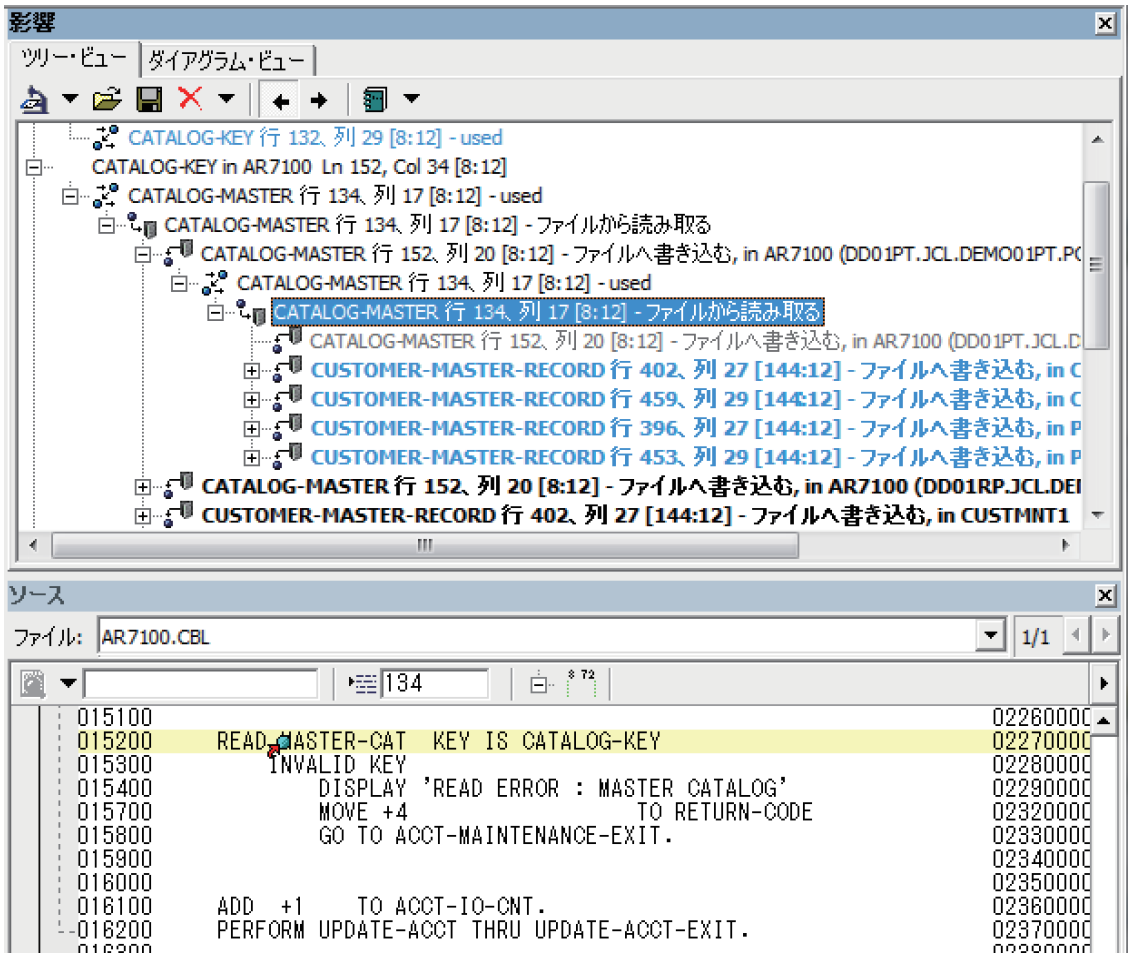
4) 関係ダイアグラム

プログラムとプログラムの間の呼び出し関係をグラフィカルに関連つけて表示します。CALL 文による明示的な呼び出しのみならず、CICS コマンドなどを経由した間接的な呼び出しも把握します。このダイアグラムはこのほかにも JCL からプログラムの起動や、プログラムからのデータアクセスなど用途に応じてきめ細かに表示内容をカスタム化することができます。



5) 影響分析

現在着目している項目がどのような経路を経て画面･ファイル･データベースに出力されるか（順方向分析）または画面･ファイル･データベースから入力されるか（逆方向分析）を順次検索して表示します。出力結果に不正が確認された際にその結果がどのような処理を経由して出力されたのかを効率的にトレースすることができ、目視による作業と比較して著しい生産性と作業品質の向上を得ることができます。



5. まとめ

ここまで、COBOL ソースコードのブラッシュアップに Enterprise Analyzer を活用する方法について例示してきました。

このように、現在重大な問題は無く稼動しているアプリケーションであっても、潜在的な問題を含んでいる可能性があります。安定稼動しているものに敢えて手を入れるリスクを負う必要はありません。しかし、プラットフォーム移行、サーバーリプレースなどのタイミングでこのような作業を計画されてみてはいかがでしょうか。長いライフサイクルを持つ COBOL プログラムであればこそ、このような施策によってさらに今後も安定的にビジネスを支え続けて行くことができるようになります。