# プログラム解析を提供するAPIの実現とその適用

四野見 秀明　玉井 哲雄

既存のプログラム解析ツール内部の C 言語の構造体やアルゴリズムを関数呼び出しとしてラップすることで，通常のプログラム解析結果のみならず，プログラムスライシング技術におけるデータ／制御依存関係の解析結果を抽出できる API を実現した．その API を利用すれば，静的スライスを利用したプログラム解析ツールなどを容易に構築することが出来る．本論文では，プログラム解析結果のデータモデル，API 設計と実現方法，そして，その API の適用例として，データ／制御依存関係に基づく解析結果を帳票化したツールについて述べる．そのツールは，システム再構築の現場でのシステム理解に実際に使用された．

## 1 はじめに

近年，Web 技術の発達により，社内の情報共有や，インターネット経由の情報発信が行われるようになった．現状では，それらを実現するシステムをフロントエンドからバックエンドまですべて新規開発することはほとんどない．既存のいわゆるレガシーシステムを，バックエンドはそのままでユーザインターフェースのみを Web ブラウザ対応させるという要求が高い [14]．また，これまで COBOL，C，PL/I で構築された既存システムを Enterprise JavaBeans を含めた Java 2 Platform, Enterprise Edition (J2EE）[17] 技術での再構築を開始している会社も出始めている．それらを称して，レガシー現代化（Legacy Modernization）[6] [18] [8] と呼ばれる．世界中のビジネスデータの 70%は，COBOL で書かれたアプリケーションで処理されているという調査結果 [1] もあり，レガシー現代化はむしろ今後本格化することが予想されている．本論文で提案するプログラム解析結果を提供する API は，レガシーシステムの一部をそのまま利用する際の呼び出しインターフェースの分析や，再利用時に現実的に必要となる機能追加の支援や，J2EE での再構築の際の既存システム理解に役立つツールの開発を容易にする．

四野見らは，COBOL，PL/I 対象の保守支援ツール『悟』（SATORU: Source code Analysis TOol for program Understanding）の研究開発を行って来た [15] [16]．そのプログラム解析結果をデータモデルとしてアクセスすることを可能にする API を，データ構造とアルゴリズムを隠蔽することにより実現した．この API を使用することにより，COBOL，PL/I プログラムについて，クロスリファレンス情報のみならず，プログラムスライシング技術 [19] [12] [13] におけるデータ依存関係（Data Dependence），制御依存関係（Control Dependence）[2] をも含めた，プログラム解析結果を参照することが可能になり，プログラムスライシング技術の現実的な適用を可能にする．

この API を利用して，再構築のためのシステム理

---

Realization of Program Analysis API and Its Application
Hideaki SHINOMI, 日本アイ・ビー・エム株式会社ソフトウェア開発研究所, Software Development Laboratory, IBM Japan, Ltd.
Tetsuo TAMAI, 東京大学大学院総合文化研究科, Graduate School of Arts and Sciences, the University of Tokyo.
コンピュータソフトウェア, Vol.x, No.x(200x), pp.x–xx.
xxxx 年 x 月 x 日受付.

解に利用される帳票生成ツールが開発され，実際のシステム再構築現場で利用された．適用例として，そのツールについても述べる．

## 2 背景

ある保険会社で COBOL で書かれたシステムを再構築する際に，データ項目名の一貫性を実現するため，既存のデータ項目の使用法を理解したいという要求があった．そこで，プログラムの出力個所から入力方向にデータの流れを溯って追跡し，各変数の値を計算した文，その実行に関与した条件文などを帳票として出力する機能を持った用語相関ツールが提案された．単独のプログラム（コンパイル単位）内のみならず，外部呼出しを経由した多数のプログラムにまたがる変数の値の代入関係追跡も必要とされた．

技術的な側面でのツール実現性の検討を依頼され検討した結果，帳票として出力されるべき情報は，プログラムスライシング技術 [19] におけるデータ依存関係と制御依存関係 [2] に基づく解析を行うことにより生成可能であることが分かった．それらは，四野見らが研究開発を行ってきた『悟』が，プログラム解析結果として保持している情報である [16].『悟』内部に持つプログラム解析結果を，他のツールからも容易に使用できるようにする必要性が生じた．

## 3 課題と解決方法

当初,『悟』の解析結果を関係データベースに保存することも検討したが，データフロー，制御フローを含む大変詳細なグラフの関係を表形式で関係データベースに保存することは，パフォーマンスの観点から現実的ではないと判断した．今回の解析結果を利用するツールは，一度に 2000 本ものプログラムを解析対象とした帳票生成を想定し，莫大な計算量を必要とすることからパフォーマンスを重要視する必要があった．C のデータ隠蔽という観点で，C++での実現も検討したが，やはりパフォーマンスの配慮と，API 利用者側が C でも C++でも解析結果を利用できるという利点を考慮し，C の関数として隠蔽を実現することにした．しかし，C の関数として実現しつつも，C++や Java のクラスとして容易にラップ可能なオブジェクト指向風の設計を行った．

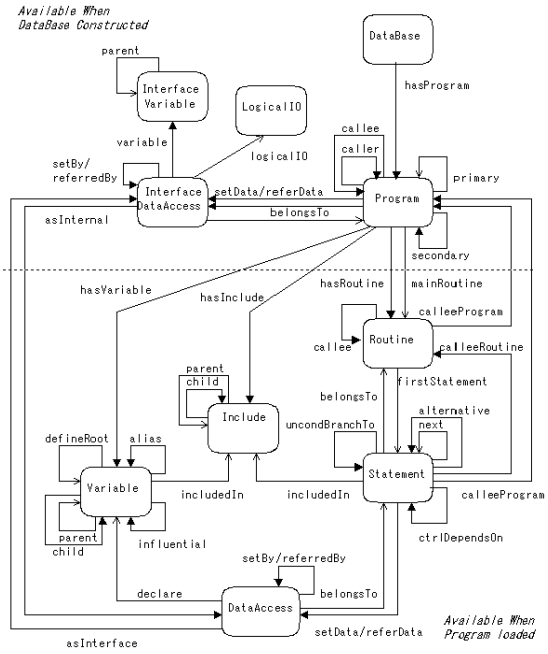


**図 1 実体関連図**

既存ツールの内部データ構造を C の関数としてカプセル化することにより，データ構造の変更は API 使用者側には影響を与えないようになった．また，C++では直接的に，Java では JNI（Java Native Interface）[7] を利用することで，容易にクラス化できるように設計されている．

## 4 データモデルに基づく API

プログラムの解析結果を言語に依存しないモデルとして提供する API を構築することにより，その API を利用して開発したツールは，元々『悟』が解析可能な COBOL と PL/I の双方に対して有効になった．ここでは，実現した API におけるプログラム解析結果のデータモデルと，関数形式について述べる．

### 4.1 実体関連図

この API を使用することにより，プログラムは図 1 のようにモデル化されて見えることになる．図中の箱は実体（オブジェクト）であり，矢印は関連である．三角が塗りつぶされた矢印は一対多の関連，塗りつぶ

されていない矢印は一対一の関連を示している．図中点線の下はプログラム内，上はプログラム間の情報のモデルである．実体は ID の振られたオブジェクトに相当し，関連は関数呼び出しの際の引数と返り値のオブジェクト ID 間の関連である．C++や Java でラップした際には，実体はオブジェクト，関連はメソッド名に相当する．

例えば，図 1 中，その Program 中に含まれている Routine を検索し，その中の特定の Routine について，含まれている Statement を検索することが可能である．DataAccess オブジェクトはプログラムのソースコード中の実行部（代入文，計算式，条件部など）中に現れる変数に相当し，setBy/referredBy がその変数に対する代入／参照個所への関係（Define/Use）である．データ依存関係は，その関係と，その変数がどの Statement に含まれているかという関係 belongsTo の組み合わせで容易に実現される．制御依存関係は図中の ctrlDependsOn が相当する．それらの関係はプログラム依存グラフ（PDG: Program Dependence Graph）[2] に相当する．プログラム間のデータ依存関係に関しても同様であり，InterfaceDataAccess がプログラム呼び出しの際のパラメータや外部変数に相当する．

### 4.2 データフローの追跡

COBOL，PL/I のプログラムでは，データ構造の再定義によるエイリアスや，データ構造ごとの代入が頻繁に行われ，保守や再構築現場でプログラムを理解する際のデータの流れの追跡を困難にしている [16]．例えば，プログラムにおいて，データの階層の下位の変数に対して代入された値が，上位レベル（例えばデータ構造全体ード）に相当する変数として参照され，そのレベルで別のデータ構造へ代入されることがある．また，そのデータ構造が再定義されていて，別の名前，別のデータ構造で参照されるなどである．

そのような場合でも正確にデータフローを追跡できるために，代入／参照を追跡するための API 呼び出しにおいては，対象とする変数全体だけではなく，その変数メモリー領域の一部を指定して追跡することも可能としている．その際には，対象とする部分についてのその変数の領域のはじめからのオフセットと対象とする領域の大きさをビット値として指定することになる．API を通して返される代入／参照の対象の変数に関しても，その変数領域中のメモリー上の対象となる領域がオフセットと大きさの組み合わせという形で返される．それに続く代入／参照の追跡にも返された領域に関するオフセットと大きさを，API のパラメータとして渡すことにより，正確なデータフローの追跡を可能にする．

再定義には，データ宣言において行われるデータ構造の再定義（COBOL の REDEFINES，PL/I の DEFINED や明示的な領域指定による BASED の利用）と，静的な解析が難しいポインタ変数による再定義（ポインタエイリアス）がある．しかし，COBOL では言語仕様上ポインタはあるが，現実的には使用されていない．本研究のプログラム解析では，ソースプログラム中でポインタ変数が使用されている場合には構文解析時に警告を出力するようになっている．しかし，銀行，保険会社を中心とする数十社の COBOL プログラムを解析した経験に基づいても，警告が出力されたことがない．COBOL が対象とするビジネスアプリケーションは，ほとんどの場合，静的なプログラム解析で対応できるといえる．PL/I におけるはポインタ変数への代入による動的な領域再定義に関しても，多くの場合代入が一箇所であり再定義対象が静的に特定できる．本来，DEFINED で明示的に再定義すべきところをポインタ変数を利用しているケースが多い．そのような場合は，DEFINED と同様に再定義として解析し，それ以外は警告を出力するようになっている．その警告が，そのプログラムが静的解析で対応できる性質かを見極める目安となる．

### 4.3 関数形式

プログラムの構成要素である実体（例えば，ステートメント，変数など）について，その実体間の関連と，各実体の属性（例えば，名前，大きさなど）の情報を関数呼び出しにより検索できるように設計されている．以下では，オブジェクト指向との対応が容易なように，実体をオブジェクトと呼び替え，オブジェクト指向の用語を用いて説明する．

オブジェクトの各インスタンスには ID が割り当てられ，メッセージ受け取り側のインスタンスとして関数呼び出しの際の第 1 引数に渡される．関数名はメッセージの受取り側オブジェクト名と，メッセージ（メソド）名の連結で出来ている．対象となるオブジェクトに対してある関連名で関連付けられているオブジェクトが一つのみの場合，対象となるオブジェクトのある属性名に対する値が一つのみである場合, 関数プロトタイプは以下の形式となる．

```
<オブジェクト ID|属性値>
    オブジェクト名メソド名 (<オブジェクト ID>, ..);
```

例えば，以下のような関数プロトタイプとなる．

```
ROUTINEID ProgramMainRoutine(PROGRAMID);
PSZ ProgramName(PROGRAMID);
```

前者は，プログラムの ID を引数にとり，その中のメインルーチンの ID を返している．後者は，プログラムの ID を引数にとり，その名前を文字列で返している．ここで，PSZ とは文字列のタイプである．

対象となるオブジェクトに関連付けられるオブジェクトや属性値が複数になる場合は，以下の形式となる．

```
<オブジェクト ID|属性値>
    オブジェクト名メソド名 (<オブジェクト ID>,
                            .. ,
                        <ステータスへのポインタ>);
```

例えば，以下のような関数プロトタイプとなる．

```
VARIABLEID ProgramHasVariable(PROGRAMID,
                              STATUS*);
```

最終パラメータには，関数呼出しごとに書換えられるステータスを保持する領域へのポインタを渡し，複数回この関数を起動することで，対応する複数個の ID を得ることができる．返される値がなくなると，関数は NULL を返す．複数個の ID や属性をリスト等の構造で返さないのはパフォーマンスを極力下げないためである．例えば，プログラム中の変数やステートメントを返すような関数の場合，返される項目の数が COBOL，PL/I のレガシープログラムでは時には何百，何千となるケースもあるからである．

### 4.4 C++，Java のクラス化

本 API は，オブジェクトに対するメッセージ送信という概念に基づき設計されている．したがって，実体をクラス，C の関数呼び出しを，C++ ならメンバ関数，Java ならメソッド呼び出しに対応づけ実装

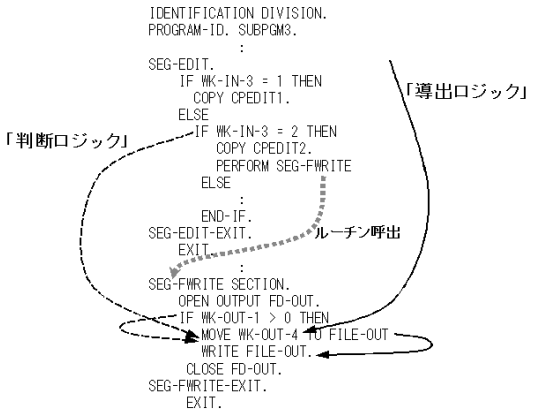


**図 2 判断／導出ロジック**

することで容易にクラス化が可能である．具体的な実装としては，各実体の ID の値を保持するために，C++ ならクラスのデータメンバ，Java ならフィールドを定義し，各々メンバ関数，メソッドの実装の中で相当する API 関数を呼ぶだけでよい．C++ならメンバ関数の定義に inline 関数を使用すれば，関数呼び出しによるパフォーマンスの低下を気にする必要がなくなる．

## 5 API 適用例としての用語相関ツール

既に述べたように，本 API を使用して用語相関ツールが開発された．用語相関ツールでは,「導出ロジック」,「判断ロジック」という用語が使われる.「導出ロジック」は，プログラムスライシング技術におけるデータ依存関係と一致すると考えてよい.「判断ロジック」は，制御依存関係とは若干異なっている．制御依存関係が，注目するステートメントが実行されるかどうかを直接左右する分岐ステートメントへの関係であるに対して,「判断ロジック」は注目するステートメントの実行に関与するすべての分岐ステートメントに相当する．図 2 中,「導出ロジック」は実線の矢印,「判断ロジック」は点線矢印で示した．矢印の方向は，データ依存関係，制御依存関係の説明に通常使われるデータや制御の流れに沿った方向で書いている．しかし，実際は「導出ロジック」,「判断ロジック」共に，矢印の先から矢印の元への逆方向に参照される．図中の「判断ロジック」をたどると，注

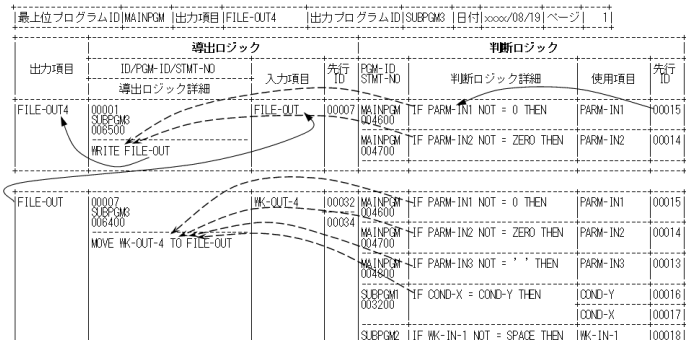


**図 3 用語相関表**

目しているステートメントの実行に直接影響する IF だけでなく，ルーチン呼び出しの元の IF も「判断ロジック」とされていることがわかる．

用語相関ツールが生成する表は，上記「導出ロジック」と「判断ロジック」を忠実に帳票化したものである．図 3 にプログラムから生成された用語相関表の一部を示し，その上に「導出ロジック」「判断ロジック」の関係をそれぞれ矢印で示した．顧客からの要請は，あくまで紙に出力するための帳票の生成であった．表では各ステートメントに ID が振られ，その ID をたどることにより関連する項目を追跡できるようになっている．しかし，その追跡は膨大な紙の出力をページをめくりながら行う必要がある．そこで，紙への出力を想定したテキストファイルによる表に，ID の部分をクリックすることにより対象個所を参照でき，追跡を容易にするための HTML のタグを挿入するユーティリティも提供した．用語相関ツールは，再構築のためのシステム理解を目的に，現場で実際に利用された．

## 6 評価

本 API を利用して，用語相関ツールは，その仕様が定義された時点でソフトウェア会社に外注され，設計実装された．ツールの開発者へは，API 自体と仕様とその使用法が与えられ,『悟』内部のデータ構造や実装は公開されていない．現場で稼動している数千本単位の COBOL プログラムを読み込んで，プログラム内，プログラム間のデータ依存関係と制御依存関係に基づく解析結果を帳票として生成するシステムが内部設計，実装，テストを含めて 18 人月で開発された．開発期間は 5ヶ月で，その間平均 3 人から 4 人が開発に係わった．本 API なしで，このような人月で開発を行うことは不可能であったと思われる．

また，元々『悟』自体は COBOL と PL/I をサポートしており，本 API は両方の言語仕様をサポートする形の抽象的なモデルになっている．用語相関ツールは，COBOL 用に開発したものであるが，実際に現場で稼動している PL/I のプログラムに対しても起動を試みた．何の修正も無くそのまま COBOL に対してと同じ帳票を生成されることが確認された．これにより，本 API が COBOL と PL/I の言語仕様を抽象化したモデルを実現していることが実証された．

## 7 関連研究

COBOL プログラムに対する解析結果の抽出に関しては，REFINE/COBOL があり，その中では Refine Object Repository [10] がを利用されている．Refine Object Repository の利用により，拡張された抽象構文木に基づく情報を抽出することが可能になっているが，制御フローやデータフローの情報までは提供してくれない．また，情報の検索プログラムは Refine 固有の言語により実装する必要がある．本 API は，C で実装することにより，COBOL と PL/I のプログラムに対して制御フロー，データフロー情報を含めたプログラム解析情報を抽出することを可能にしている．

Sapid [3] [11] は，細粒度リポジトリに基づく C 言語を対象とする CASE ツール・プラットフォームである．プログラム解析結果のためのモデル I-model

が提案され，制御とデータの依存関係を扱う SDA（Sapid Dependency Analyzer）という API 関数群を持つ．SDA における制御依存関係は，制御流れグラフ（CFG: Control Flow Graph）で表され，いわゆるプログラム依存グラフ（PDG）[2] における制御依存辺を直接表現するものでは無い．ただし，CFG から容易に PDG における制御依存関係を抽出できると思われる．データ依存関係解析は，関数内に限定され，関数をまたがる情報は提供していない [11]．本研究における解析では，C 言語の関数に相当する手続き（ルーチン）内のデータフロー解析結果をサマライズした情報を生成する．その情報を利用し，手続き呼び出しに伴うデータ依存関係も API を通じて抽出できるようになっている．本 API を利用した用語相関ツールも，それを利用し複数プログラムにまたがるデータ依存関係をも帳票化している．また，本 API は，あくまでプログラムを解析することを目的としているが，Sapid はソースプログラムに対する変更操作までも対象にしていることも特徴といえるだろう．

Decomposition Slice [4] [5] は，出力変数に注目し，その出力を生み出すプログラムスライスの集合である．用語相関ツールは，出力変数に注目し，それに係わるデータ依存関係，制御依存関係に基づく情報を全て帳票にしていることに相当する．本 API を利用することにより，Decomposition Slice と等価な情報を容易に帳票化することを実現している．

## 8 おわりに

レガシー現代化が本格化するにしたがって，COBOL，PL/I で書かれたシステムを，UML を利用した設計等で再構築する要求も高まっている．本 API を利用すれば，既存プログラムの解析結果に基づき XMI (XML Metadata Interchange) [9] のフォーマットに従った XML データを生成するツールを開発可能である．それは，モデリングツール等の開発環境の入力となり，現状の設計把握に役立ち，また再設計の基礎となりうる．モデリングツールへの入力のためは，解析結果からの抽象化も必要となるであろう．

## 参考文献


[1] Aberdeen Group, Inc.: Legacy Applications: From Cost Management to Transformation, March 2003.

[2] Ferrante, J., Ottenstein, K. J., and Warren, J. D.: The Program Dependence Graph and Its Use in Optimization, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 9,No. 3(1987), pp. 319–349.

[3] 福安直樹, 山本晋一郎, 阿草清滋: 細粒度リポジトリに基づいた CASE ツール・プラットフォーム Sapid, **情報処理学会論文誌**, Vol. 39,No. 6(1998), pp. 1990–1998.

[4] Gallagher, K. B. and Lyle, J. R.: Using Program Slicing in Software Maintenance, *IEEE Transactions on Software Engineering*, Vol. 17,No. 8(1991), pp. 751–761.

[5] Gallagher, K.: Visual Impact Analysis, *Proceedings of the International Conference on Software Maintenance*, November 1996, pp. 52–58.

[6] IBM Corp.: *VSE/ESA Software Newsletter, First/Second Quarter, 1999*, 1999.

[7] Liang, S.: *The Java Native Interface: Programmer's Guide and Specification (Java Series)*, Addison-Wesley, 1999.

[8] Micro Focus: Micro Focus EnterpriseLink, 2003. http://www.microfocus.co.jp/products/entlink.asp.

[9] Object Management Group: *OMG XML Metadata Interchange (XMI) Specification*, Jan. 2002.

[10] Reasoning Systems, Inc.: *REFINE User's Guide*, Palo Alto, California, 1992.

[11] Sapid Homepage: http://www.sapid.org/.

[12] 下村隆夫: Program Slicing 技術とテスト, デバッグ, 保守への応用, **情報処理**, Vol. 33,No. 9(1993), pp. 1078–1086.

[13] 下村隆夫: **プログラムスライシング技術と応用**, 共立出版, 1995.

[14] 四野見秀明: Web ベース開発におけるレガシーシステム利用技術, **ウィンターワークショップ・イン・伊豆論文集**, 情報処理学会, 2002, pp. 67–68.

[15] 四野見秀明, 藤井邦和, 牧野正士, 津田和幸: 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, **情報処理学会ソフトウェア工学研究会報告**, Vol. 92,No. 79(1992), pp. 1–9.

[16] 四野見秀明, 藤井邦和, 高橋真由美: データフロー解析に基づくプログラム保守支援, **情報処理学会第** *54* **回全国大会論文集**, Vol. 1(1997), pp. 323–324.

[17] Sun Microsystems, Inc.: *Java 2 Platform, Enterprise Edition (J2EE)*.

[18] Wahli, U., Agatsuma, M., Barosa, R., Hekkenberg, G., McGoogan, B., and Winoto, I.: *Legacy Modernization with WebSphere Studio Enterprise Developer*, IBM Corp., 2002. IBM RedBook.

[19] Weiser, M.: Program Slicing, *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4 (1984), pp. 352–357.