Vol.2012-SE-176 No.7
Vol.2012-EMB-25 No.7
2012/5/21

レガシーGUIアプリケーションからのDatalogを用いた画面遷移抽出

岩間 太[1,a)]　立石 孝彰[1,b)]

概要：C/C++言語で記述されたGUIアプリケーションのソースコードから，画面遷移情報を抽出するdatalogベースのコード解析器を構築した．そして，177個の実行可能ファイルから構成される大規模な実GUIアプリケーションへ適用し，36個の実行可能ファイルから遷移を抽出することに成功した．

キーワード：Datalog，レガシーシステム，GUIアプリケーション，画面遷移

Datalog-based Screen-Transition Extraction from Legacy GUI Application

Futoshi Iwama[1,a)]　Takaaki Tateishi[1,b)]

Abstract: We have implemented a Datalog-based program analyzer for extracting screen transitions from C/C++ source code of legacy GUI application. On a real GUI application that consists of total 177 programs, we have successfully extracted screen transtions from 36 programs.

Keywords: Datalog, Legacy Transformation, Screen transition, GUI Application

1. はじめに

MVCアーキテクチャに基づくネイティブGUIアプリケーション（Visual C++などで作成されたアプリケーション）を，Webアプリケーションとして再構築することが多い．このような種類のレガシートランスフォーメーションでは，そのアプリケーションにおけるモデル，ビュー，コントロールに相当するプログラムコードの理解と抽出の効率化が重要である．

1.1 レガシーGUIアプリケーションに対する現行分析

大規模なGUIアプリケーションの場合，限られた時間で，それらのソースコードを目視で理解・整理し，MVCに基づく設計モデルを構築するためには，このようなことを行えるだけの技量を持ったエンジニアを多く必要とする．この結果，現行アプリケーションの分析に対するコストが増大する．特に，すでに使われなくなったGUIライブラリを用いて作成されたC/C++ネイティブGUIアプリケーションの場合，そのGUIライブラリの特徴にも精通していなければならず，さらに現行アプリケーションの分析を困難にし，コスト増大の要因となる．

また，基幹業務システムのフロントエンドとなるようなGUIアプリケーションは，入力された値を基幹業務システムへ送信することが主な目的である．このため，このようなフロントエンドとしてのGUIアプリケーションでは，ビューとコントロールに相当する画面遷移（ウィンドウやフォームなどのGUIコンポーネントの表示・非表示の制御）とその遷移条件の理解が重要となる．我々は，このような基幹業務システムのフロントエンドとして動作する

[1] 日本IBM, 神奈川県大和市下鶴間 1623-14
IBM Japan, Shimoturuma 1623-14, Yamato, Kanagawa
[a)] gamma@jp.ibm.com
[b)] tate@jp.ibm.com

GUI アプリケーションを対象として，ソースコードから自動的に画面遷移を抽出することによって，現行システムにおける（フロントエンド側の）分析コストを抑制することにした．ここで，抽出された画面遷移は，Strusts のような Web アプリケーションプラットフォームへの入力となることを想定しており，これを簡単化し，**遷移元画面**，**遷移先画面**，**イベント**，**アクション**の 4 つ組を抽出することが我々の目的である．

1.2 ツール化に向けての課題

ところが，このような画面遷移を現実的な GUI アプリケーションから静的かつ自動抽出するためのツールを開発するためには，(1) 実装方法や使用される GUI ライブラリがアプリケーション毎に異なることを考慮しなければならず，また，(2) 大規模なソースコードに対しても解析を行える必要がある．

例えば，grep を用いて，関数名を利用して，ウィンドウが生成・表示される部分を特定することなどを行い，これを基点として，データフローやコールフローを目視によって（あるいは統合開発環境の機能を用いて）追跡することが一般的に行われている．そして，どのようなコールバック関数がどの GUI 部品から呼ばれたのかを調べることによって画面遷移を抽出することができる．しかし，ウィンドウや GUI 部品を生成・表示させる関数や，それらの親子関係などの定義方法は GUI ライブラリ毎に異なり，また，データフローやコールフローを追跡する場合においても，GUI 部品と関連付くコールバックを追跡する必要がある．目視による追跡では，エンジニアは GUI ライブラリの動作を理解しているため，これらのことを自然に行える．しかし，ツール化においては，このような GUI ライブラリに特有の知識を取り扱うことができるほど柔軟な拡張性がツールに必要である．

また，我々が対象とする GUI アプリケーションは，一つの実行ファイルだけから成るプログラムではなく，数百の実行ファイル（exe ファイル）と数千の DLL ファイルから成るような GUI アプリケーションである．このため，これらに対するソースコードの情報をディスク上で一元管理でき，静的解析が行いやすいソースコードリポジトリが必要となる．

1.3 ツール化のためのアプローチ

このような 2 つの解析ツールへの要求を満たすために，我々はソースコード中の個々の要素間の関係を datalog における関係へエンコードしてリポジトリへ格納し，datalog のルール（以降では単純にクエリと呼ぶ）を用いて画面遷移を抽出するアルゴリズムを記述することにした．ここで，ソースコード中の関係とは，主に，代入文における左辺と右辺の関係である．このアプローチにより，BDD によって datalog の関係を表現することで，リポジトリのサイズが抑制され，また，解析に必要なソースコード情報をメモリ上で保持することができるという利点を生む．

1.4 関連研究

文献 [1], [2], [3], [4] は，我々と同様に，datalog を用いたコード解析を行っている．これらの研究では，バイトコードへ変換後のプログラムを datalog の関係としてエンコードすることを通して，ポインタ解析やデッドロック検出などを行っている．つまり，強く正規化されたコード (より少ないパターンで定義されるコード形式) を対象にするため，クエリの記述が簡単になるが，ソースコード上の特徴や情報 (コードパターンなど) の多くは一般に失われる．一方で，必要に応じてポインタ解析などを用いるものの，我々の主な目的は，ソースコード理解である．このため，SSA (Single Static Assignment) 変換などを行った後のプログラムやコンパイル後のバイトコードを datalog の関係へエンコードするのではなく，AST (Abstract Syntax Tree) をエンコードすることに着目した．これによって，ライブラリの特徴や，事前に目視やインタビューによって知り得たソースコードの特徴を，datalog のルールとして記述できるようになり，解析の効率化や精度向上に繋げることを期待できる．

また，ソースコード理解という目的をより強く持った研究として，C 言語ソースコードからコード理解に有用な情報をリポジトリへと格納し利用するというアプローチの研究，例えば [5]，が挙げられる．[5] ではソフトウエア工学的に有用な情報をソースコードの解析により取り出しリポジトリに格納し，この情報を利用する API を準備することで，中流-下流における CASE ツールの作成基盤を与えている．また [6] ではリポジトリとして XML 文書を利用した細粒度なソフトウエア情報のリポジトリの実装を与えている．

1.5 本論文の貢献

本論文の貢献は次の通りである．

GUI ライブラリに対する datalog ルール　典型的な GUI ライブラリを用いたアプリケーションに対して，画面遷移情報を抽出するための datalog のクエリを与えた．

C/C++解析における AST を対象とした datalog の利用　ソースコードに対する知見を取り込みやすいプログラム解析ツールを構築することを目的に，datalog を用いた C/C++ソースコードの AST を対象とする解析の枠組みを与えた．

大規模な実 GUI アプリケーションへの適用実験　実レガシー GUI アプリケーションに対して適用実験を行った．

1.6 本論文の構成

2節では，解析対象として想定しているGUIアプリケーションについて記述する．また同時にレガシーGUIアプリケーションの解析において特に問題となる点を取り上げる．3節では，datalogに基づいた画面遷移抽出のためのプログラム解析器を2節で記述した枠組みのGUIアプリケーションに対して構築する．4節では，3節で構築した画面遷移抽出器を大規模レガシーGUIアプリケーションの1モジュールに試験適用した際の結果を報告し，その評価を行う．5節で本論文をまとめ，4節の適用と評価を通しての今後の課題について触れる．

2. レガシーGUIアプリケーションの画面遷移

本節では，本論文の手法が解析の対象とする，C/C++言語で記述されたGUIアプリケーションの基本的な枠組みを示す．また，レガシーGUIアプリケーションシステムからの画面遷移情報の抽出に関わる問題についてまとめる．記述した枠組み，ならびに，問題点は一般的なレガシーGUIアプリケーションにおいても成立しているものと考えられる．

特にGUIアプリケーション記述のためのライブラリ関数として以下を想定する．

- ***W*indow()**, ***Button*(window)**
  新しい画面オブジェクト，ボタンオブジェクトを作成するためのプリミティブである．***Button*** に関しては上記の命令により，ボタンオブジェクトが作成されwindowが指す画面上に配置されることとする．
- ***setAttr*(obj,"attr",value)**
  画面関係オブジェクトの属性に値を設定するためのプリミティブ．上記の命令により，`obj` が指すオブジェクトの属性 `attr` に値 `value` がセットされる．特に各画面関係オブジェクトはidとしてのname属性を持つ．
- ***find*(obj,"objName")**
  name属性に設定された値をキーとしてオブジェクトをルックアップするプリミティブである．上記の命令により，`obj` 上に配置され，かつ，name属性が `objName` のオブジェクトが取り出される．
- ***setCallBack*(obj,"event",fun)**
  イベント処理を処理するコールバック関数を画面関係のオブジェクトに対して設定するためのプリミティブ．上記の命令により，オブジェクト `obj` に対して，イベント `event` が生じた際のイベント処理関数として `fun` が登録される．
- ***show*(obj)**
  画面関係オブジェクトを表示させるためのプリミティブ．特に `obj` が画面オブジェクトの場合，この画面への画面遷移を生じさせる．

```
1  void SC031_create(Window *window){
2    window = new Window();
3    setAttr(window, "name", "SC031");
4    setAttr(window, "caption", "検索画面");
5    setAttr(window, "init","SC031_init");
6    Button btn1 = new Button(window);
7    Button btn2 = new Button(window);
8    setAttr(btn1,"name","search");
9    setAttr(btn1,"caption","検索実行");
10   setAttr(btn2,"name","preference");
11   setAttr(btn2,"caption","設定");
12   ...
13 }
```

図 1 ウィンドウオブジェクト作成のコード (form31.cpp) の一部

```
1  Window win31,win32,win33;
2  void SC031_init(Window obj) {
3    win31 = find(obj,"SC031");
4    win32 = find(obj,"SC032");
5    Button searchBtn = find(win31,"search");
6    setCallBack(searchBtn,"entered", initSearch);
7    ...
8  }
9  void initSearch(Button self) {
10   int b = process1(self);
11   if (b) process2(win32);
12   ...;
13 }
14 void process2(Window win) {
15   ...; show(win); ...;
16 }
```

図 2 ウィンドウオブジェクト作成用コード (formI.c) の一部

2.1 解析対象とするGUIアプリケーションの枠組み

通常，GUIアプリケーションは，画面，ならびに，画面上の部品（ボタンなど）を定義している画面関係オブジェクト（フォームオブジェクト）と，フォームオブジェクトに対する操作（表示，消去，イベントの設置）によって実現されている．フォームオブジェクトには画面オブジェクトや各種のコンテナオブジェク，ボタンオブジェクトがあり，コンテナやボタンはウィンドウや他のコンテナオブジェクト上に配置，表示することが可能である．この配置関係によりフォームオブジェクト間には親子関係が定義される．また，フォームオブジェクトに対する操作は，専用の関数（ライブラリ等として用意される関数）によって行われる．本論文では上記に示した形の関数を想定した．

例として，図1の画面作成関数のコード，ならびに，図2の画面ロード用のコードを考える[*1]

図1の関数 `SC031_create()` は，特定のウィンドウを作

[*1] このコードは機密保持のため、実際のコードとは異なる擬似コードを使用しています

成し，引数に与えられたポインタの先に格納する．2 行目において ***W*indow()** によって新しい画面オブジェクトを作成している．3 行目 ***setAttr***(window,"name","SC031") では，上記で作成した画面の属性 name に値"SC031"を設定し，5 行目ではロード時の初期化関数を指定している．6-7 行目では，画面上の部品オブジェクトを作成している．例えば，***Button***(window) により，ボタンオブジェクトが作成され window が指す上記の画面上に配置される．結果，図 1 の関数の実行によりボタンを 2 つ持つウィンドウオブジェクトが作成される．*2

図 2 は，画面 SC031 の初期化を行う関数を示している．1 行目で関係する画面を保持する大域変数を用意する．2 行目の関数 SC031_init がフォーム全体のオブジェクトを obj を引数として，画面 SC031 のロード時に呼び出される．3,4 行目ではライブラリ関数 ***find***() を使用して，画面オブジェクトを obj から名前をキーにして取り出している．例えば 3 行目では ***find***(obj,"SC031") により，画面 SC031 のオブジェクトが取り出される．その後，5 行目で，“検索実行ボタン” オブジェクトが取り出され，6 行目で，ライブラリ関数 ***setCallBack***() を使用して，このボタンのイベント処理を設定している．***setCallBack***(searchBtn,"entered",serachInit) により，情報検索ボタンに対して entered イベントが生じた場合，serchInit 関数が searchBtn オブジェクトを引数として呼び出されることとなる．9 行目以降は，上記の searchInit 関数の一部である．15 行目の ***show***(win32) の実行により，画面 SC031 から画面 SC032 へと画面遷移が生じる．

2.2 レガシー GUI アプリケーション解析の問題点

レガシーアプリケーションにおいては，アプリケーションに関する様々な情報が失われているためその解析が難しくなる．例えば，ソースコードへの更新が必ずしもドキュメントに反映されていないため，ドキュメント情報は必ずしも信用することが出来ない．また，ソースコードで用いられているライブラリのソースコードや，場合によってはアプリケーションのソースコードの一部を解析に用いることが不可能な場合も多い．同時に，大規模なシステムでは，ソースコードを記述する言語自体に独自拡張が施されているケースもあり，この場合，既存のソースコード解析器をそのまま利用することは困難となる．

以下に，この度のレガシー GUI アプリケーションの刷新プロジェクトにおいて遭遇した，GUI アプリケーションからの情報抽出における問題点をまとめた．

(1) 画面仕様書は信頼できず，画面遷移情報（発生イベント，イベント処理関数，画面遷移）はソースコードから取得する必要がある．
(2) C 言語に独自拡張を施した部分があり，汎用的なコンパイラでは完全にはパースやコンパイルができない部分が存在する．
(3) GUI ライブラリのソースコードを利用できない．
(4) GUI アプリケーション全体が複数の会社で分担して構築されており，各社担当部分においてソースコードの記述パターン（使用するライブラリ関数自体やその使用方法など）が異なっている．同時に，一部の他社専用のソースコードを利用できない．

上記 (1) により，ソースコードから画面遷移を抽出する必要が生じる．また (2)(3)(4) により，ソースコードが得られない部分の情報を外から独自のルールとして柔軟に織り込める形で，また，コードのパース結果が機械的に取得できる部分の情報のみを組合せることで解析を行える (モジュラーな) 解析器を実装する必要があることとなる．

3. Datalog を用いた画面遷移抽出

前節の問題点を考慮して，我々は，レガシー GUI アプリケーションのソースコードから画面遷移情報を抽出するプログラム解析器を datalog を用いて構築した．

このプログラム解析器の全体像を図 3 に図示する．まず「Makefile 解析器」により make ファイルが解析され，関連するソースコードが実行ファイルごとに纏められる，次に，「CDT パーサー」[7] を用いて実行プログラム単位ごとの抽象構文木が作成される．その後，「コード情報抽出器」において，AST を解析することでコードから基本的な情報が取り出され，BDD 形式の Relation データとして格納される．最後にこれらの Relation データの集合から画面遷移関係を取り出す datalog 形式のクエリを準備し，クエリ処理系に問い合わせを行うことで，画面遷移情報が取得される．

我々は datalog のクエリ処理のシステムとして bddbddb[8] を採用し利用した．この理由は bddbddb が処理対象データを BDD 形式で扱うことによりクエリ処理を高速に処理できること，また，大規模なプログラム解析の実績があるためである．

3.1 画面遷移抽出方法

2 節で与えられた枠組みの GUI アプリケーションに対して，画面遷移を抽出する方法を以下にまとめる．

(M1) イベント，アクションの抽出．これは関数呼びだし ***setCallBack***(x,"event",callBackFunc) で，callBackFunc が関数となるものを特定すればよい．イベント名が event でアクション（イベント処理関数）が callBackFunc となる．
(M2) 遷移元画面の抽出．(M1) の x がどの画面の部品オブジェクトかを特定することで行われる．具体的には，

*2 実際には，ボタンだけではなく，Text, Input, GroupBox 等といった様々な種類のフォームやコンテナが利用されるが省略する．

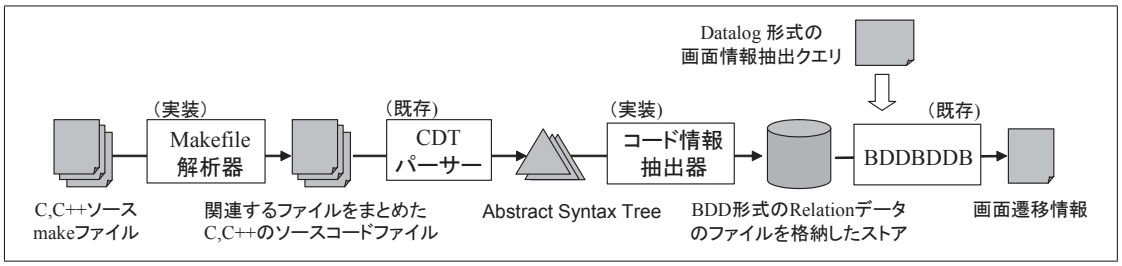

図 3 datalog を用いた画面遷移抽出のためのプログラム解析器の概要

(M2-a) `x'=`***find***`(win,"BtnName")`
(M2-b) `win'=`***find***`(obj,"ScName")`
(M2-c) `x'` の値が `x` に流れている
(M2-d) `win'` の値が `win` に流れている
という条件がすべて成立っている場合，(M1) のイベントが画面 `ScName` で起こることとなり，これが遷移元画面となる．

(M3) 遷移先画面の抽出．(M1) の関数 `callBackFunc` の実行を追うことで行われる．具体的には，
(M3-a) 関数 `callBackFunc` の呼出し時に関数 ***show***`(win2)` が実行される．
(M3-b) `win2'=`***find***`(obj2,"ScName2")`
(M3-c) `win2'` の値が `win2` に流れている
という条件が成立っている場合，(M1) のイベント処理で画面 `ScName2` への遷移が生じる．

この遷移抽出方法は，データフロー解析 (M2-c,M2-d,M3-c)，コントロールフロー解析 (M3-a) により実現できる．しかしながら，ライブラリや，その他一部のソースコードが機械的な解析に利用できない場合，必要な部分のコードの情報を柔軟に利用する必要がでてくる．また，大規模なアプリケーションの解析においては，事前の知識を使用して解析をショートカットすることが効率や精度の面において有効／必要である．そのため，我々は図 3 に示した形のプログラム解析器を構築し上記の画面遷移抽出を実装した．

3.2 C ソースコードからの基本情報の抽出

図 3 の「コード情報抽出器」において，ソースコードの AST から基本的な情報を抽出しストアに格納する．図 4 に定義したドメインを，図 5 にソースコードから抽出する基本的な関係の一部を掲載する．

記号	ドメイン	説明
V	変数のドメイン	名前とスコープから構成される
F	フィールド名のドメイン	コード中のフィールド名を表す
C	定数のドメイン	整数，文字，文字列など
T	型のドメイン	コード中の型,int,char など
L	ロケーションのドメイン	コード上の抽象的な場所を表す
N	名前のドメイン	変数の名前を表す

図 4 定義した datalog によるクエリにおけるドメイン

図 4 のドメインにおいて，変数ドメイン V の値はコード中の変数名とそのネームスコープでエンコードする．そのため，コード中で同じ名前を持つ変数でもネームスコープが異なれば異なった変数として表現される．例えば変数名 x，スコープ a に対して，$x@a$ などと表現されうる．ネームスコープの情報は CTD パーサーの機能を用いて取得した．また，L は (抽象的な) ロケーションを表し，ロケーションは各関係を抽出したコード上の位置関係を示す．ロケーションはコードファイルのパス名，各関係の抽出元部分の AST 上の位置から計算される．

図 5 に掲載したソースコードから抽出する Relation データについて簡単な例を挙げる．例えば，`x=1` から $AsgnC(x@a, 1, l)$ という関係が抽出され，`x=y` から $AsgnC(x@a, y@a, l)$ という関係が抽出される．また，`x=y.fld` から $ReadF(x@a, y@a, fld, l)$ が，`x.fld=y` から $WriteF(x@a, fld, y@a, l)$ が抽出される．`*x=y` からは $Ptr(x@a, y@a, l)$ が抽出される．`x=fun(y)` からは $Call(x@a, fun@a, l)$，$CallP(1, y, l)$ が抽出される (ロケーション l の一致により，対応する $Call, CallP$ が決定される)．`x=1;y=x` という部分コードに対して，$AsgnC(x, 1, int, l_1)$，$AsgnV(y, x, l_2)$ という関係が抽出され，同時に $Next(l_1, l_2)$ が抽出される．また `x=1` からは，$AsgnC(x@a, 1, l)$ とともに $Name(x, x@a)$ という関係が抽出される．

bddbddb が扱うことの出来るデータは整数データのみであり，各ドメインのデータはすべて整数でエンコードしなければならない．そのため上記の関係がもつデータ $x, x@a, fld, l$ などは，実際にはすべて整数であるが，論文においては，整数ではなく対応する文字列を用いて表記する．実装においては，各ドメインごとにエンコーダー，デコーダを準備した．

例えば図 2 の `win31=find(obj,"SC031")` という 3 行目の文からは，以下の Relation データが抽出される．

$$AsgnC(z_1, ''SC031'', string, l_1) \quad NextL(l_1, l_2)$$
$$Call(x_1, x_2, l_2) \quad CallP(1, x_3, l_2) \quad CallP(2, z_1, l_2)$$
$$Name(x_1, win31) \quad Name(x_2, find) \quad Name(x_3, obj)$$

上記において，x_1, x_2, x_3 等は Relation データにおける変数ドメインのデータであり，ソースコードの変数と同一で

関係名	関係の各項のドメイン	説明
$AsgnC$	$(x:V,\ c:C,\ t:T,\ l:L)$	場所 l において，変数 x に型 t の定数 c を代入
$AsgnV$	$(x:V,\ y:V,\ l:L)$	場所 l において，変数 x に変数 y の値をを代入
$ReadF$	$(x:V,\ y:V,\ g:F)$	場所 l において，変数 x に y のフィールド g の値を代入
$WriteF$	$(x:V,\ g:F,\ y:V)$	場所 l において，x のフィールド g に変数 y の値を代入
Ptr	$(x:V, y:V, l)$	場所 l において，ポインタ x が変数 y の値を指す
$Call$	$(x:V,\ f:V,\ l:L)$	場所 l において，変数 x に関数 f の呼出し結果を代入
$CallP$	$(n:I,\ x:V,\ l:L)$	場所 l における関数呼出しにおいて，変数 x の値が第 n 引数となる
$FunD$	$(f:V,\ l:L)$	場所 l において，関数 f が定義されている
$FunDP$	$(n:I,\ x:V,\ l:L)$	場所 l において定義されている関数の第 n 仮引数は変数 x である
$NextL$	$(l_1:L,\ l_2:L)$	同一関数内の制御フローにおいて，場所 l_1 の次の場所は l_2
$Name$	$(n:N,\ x:V)$	変数 x の変数名は n である

図 5 コードから抽出する関係の一部

はない（スコープ情報を伴ってエンコードされている）ことに注意．また，z_1, z_2 などは，コードの情報を準備された所定の関係データに変換するために用いられる変数でありソースコード中の変数とは対応しない．

また，図 2 の 1 行目 `void SC031_init(Form obj){..` からは以下の Relation データが抽出される

$FunD(f, l_3)$　$FunDP(1, x_4, l_3)$
$Name(SC031\_init, f)$　$Name(obj, x_4)$

この Relation データから，関数 `SC0031_init` の第一仮引数が `obj` といった主要情報を機械的に得ることが可能である．

レガシーコードの解析においては，独自の前処理や言語拡張により完全にはパースが出来ない場合も存在する．しかしながらパース不可能な部分が目的とする解析に影響しない限りにおいてはこの部分を無視することができる．ソースコードから基本情報を抽出・保存し，その後，この抽出情報を使って目的の解析を行うというアプローチでは，このような必要部分，もしくは解析可能部分のみに対応するすることが比較的容易に可能となる．

3.3 datalog による画面遷移抽出のためのクエリ

前節で得られた類の Relation データから画面遷移情報を抽出するクエリを datalog を用いて記述する．datalog では例えば，以下のような prolog-like な記法で推論を記述する．

```
NewRelation(x,z) :- Relation1(x,y), Relation2(y,z).
NewRelation(x,z) :- Relation3(x,z).
```

上記の記述では，Relation1 の第二項と Relation2 の第一項が同じとなるデータが存在した場合，該当する Relation1 の第一項と Relation2 の第二項からなる新しい関係 NewRelation を作成し，同時に Relation3 データをそのまま NewRelation のデータとする．

以下 2 節で用いられているライブラリに対して画面遷移を抽出する datalog のルールを，3.1 節の画面遷移抽出方法に基づいて bddbddb で記述する．

```
1  PageTransition(s:C,e:C,a:V,t:C) outputtuples.
2  PageTransition(s,e,a,t) :−
3      EventAction(x,e,a), SrcScreen(x,s), TgtScreen(a,t).
4
5  EventAction(x:V,e:C,a:N) outputtuples.
6  EventAction(x,e,a) :− CallP(_,setv,l), N.set(setv),
7        CallP(1,x,l), CallP(2,x2,l), CallP(3,cbf,l),
8        ValC(x2,e), FunV(cbf,a).
9
10 SrcScreen(x:V,s:C) outputtuples.
11 SrcScreen(x,s) :− FindV(xq,win,b), FindV(winq,obj,s),
12                   DFlow(xq,x), DFlow(winq,win).
13
14 TgtScreen(a:V,s:C) outputtuples.
15 TgtScreen(a,s) :− CFlow(a,acf,l), N.show(acf),
16                  CallP(l,1,win2), FindV(win2q,obj2,t),
17                  DFlow(win2q,win2).
18
19 FindV(x:V,y:V,c:C).
20 Find(x,y,c) :− CallP(_,findv,l), N.find(findv),
21                CallP(1,y,l), CallP(2,cv,l), ValC(cv,c).
```

図 6 画面遷移情報を抽出するクエリ

画面遷移情報の抽出を行うトップレベルのクエリを図 6 に示す．5-8 行目が (M1) に，10-12 行目が (M2) に，14-17 行目が (M3) に対応する．19-21 行目の `FindV(x,y,c)` は，`x=`***find***`(y,c)` の関係を意味している．これらのクエリを使用した 1-3 行目がトップレベルのクエリとなる．図 6 において，`N.setCallBack(x)`，`N.show(x)` は，変数 `x` が，それぞれ `setCallBack`，`show` という名前 (ドメイン `N` の値) を持つ変数であることを意味する Relation である．この Relation データはコード情報抽出時に生成する．

図 7 では，図 6 中で使用した取替え可能な解析モジュールを定義している．`ValC(x,c)`，`FuncV(x,f)` は，変数 `x` がそれぞれ `c` の定数，`f` の関数を格納していることを意味し

```
1  ValC(x:V,c:C).
2  ValC(x,c) :− AsgnC(x,c,_,_).
3  ValC(x,c) :− AsgnV(x,y,_), ValC(y,c).
4  ValC(x,c) :− FldValC(y,fld,c), ReadF(x,y,fld,_).
5  ValC(x,c) :− ValC(y,c), Call(_,f.l1), CallP(i,f,y,l1),
6               FunD(f,l2), FunDP(i,x,l2).
7  FldValC(x,fld,c) :− ValC(y,c), WriteF(x,fld,y,_).
8
9  DFlow(x:V,y:V).
10 FDFlowR(x:V, fld:F, y:V).
11 FDFlowW(x:V, y:V, fld:F).
12 DFlow(x,y) :− AsgnV(y,x,_).
13 DFlow(x,y) :− FDFlowW(x,z1,fld), DFlow(z1,z2),
14               FDFlowR(z2,fld,y).
15 DFlow(x,y) :− Call(_,f.l1), CallP(i,f,x,l1),
16               FunD(f,l2), FunDP(i,y,l2).
17 DFlow(x,y) :− DFlow(x,z), DFlow(z,y).
18 FDFlowR(x,fld,y) :− ReadF(y,x,fld,_).
19 FDFlowW(x,y,fld) :− WriteF(y,fld,x).
20
21 FunV(x:V, f:V).
22 DFlowPrj2(x) :− DFlowPrj(_,x).
23 FunV(x,f) :− FunD(f,_), f=x.
24 FunV(x,f) :− AsgnOp(_,x,_), !DFlowPrj2(x), x=f.
25 FunV(x,f) :− AsgnOp(_,x,_), DFlow(g,x), FunV(g,f).
26
27 CFlow(f:V,g:V,l:L).
28 CFlow(f,g,l) :− CallFun(f,g,l).
29 CFlow(f,g,l) :− CallFun(f,h,_), CallFun(h,g,l).
30 CallFun(f,g,l) :− InFunLoc(f,l), Call(gp,l), FunV(gp,g).
31 InFunLoc(f,l) :− FunD(f,l).
32 InFunLoc(f.l) :− InFunLoc(f,l2), NextL(l2,l).
```

図 7 各種のフロー解析 (flow-insensitive) を行うクエリ
(状況に応じて適切な実装と取替え可能な解析モジュール)

ており `CFlow(x,y,l)` は，関数 x からのコールフロー中において，関数 y がロケーション l で呼び出されていることを，`DFlow(x,y)` は変数 x の値が変数 y に流れていることを意味している．これらの関係は適切な精度のものを定義して利用することができる．必要に応じて精度の高いルールを記述することもできるが計算コストが高くなる．

図 7 においては，代入関係の順序などを考慮しない flow-insensitive なフロー解析を実装している．またポインタが指すメモリを経由した類の値の受け渡しも考慮していない (例えば `Ptr(x,v,l1)`,`Ptr(y,w,l2)`,`DFlow(v,w)` が成立つ場合，`DFlow(x,y)` などが推論できる)．対象のコードをレビューしてた上でこのレベルの解析で十分だと判断し，この実装に至った．

図 1,2 のコードに対して, クエリ `PageTransition` の結果として四組み (`"SC031"`,`"entered"`,`initSearch`,`"SC032"`) が得られる．

4. 適用と評価

レガシートランスフォーメーションのプロジェクトにおいて構築したプログラム解析器の試験適用を行った．

4.1 適用

対象アプリケーションの試験適用を行ったサブシステムの情報を図 8 に記載する．

試験適用対象サブシステムは，1504 個のファイル (.c ファイル，.cpp ファイル，.h ファイル) から構成されており，画面総数は 264 個である．このサブシステムに対して，図 3 の「Makefile 解析器」を適用することにより，177 個の実行可能プログラム＋ 490 個の DLL プログラムが得られた．各実行プログラムは一つの機能に対応しており，複数の画面によって実現されている．各実行可能プログラムに対して，前節の解析を用いて画面遷移情報の抽出を行った結果，36 個の実行画面について画面遷移情報が検出できた．

また，画面遷移情報の検出精度を調べるため，試験適用サブシステム中の，画面遷移が検出された一つの実行プログラムに対して，エンジニアと共に詳細なレビューを行い，コードの目視による画面遷移検出とプログラム解析器での画面遷移検出結果とを比較した．この結果を図 9 に示す．

サンプルとなった実行プログラムのソースコードをレビューした結果，関与する画面は 7 個であり，総計 19 個のイベントが設定されており，26 個の画面遷移情報 (遷移元画面，イベント，アクション，遷移先画面の四組みを指す) があると評価された．一方，プログラム解析による抽出では，レビュー結果の画面遷移情報を完全に含む 47 個の画面遷移情報が抽出された．False Positive (検出された遷移で実際の遷移でなかったものの割合) は (47-26)/47=0.44 より，44%であった．また，False Negative (実際の遷移で検出されなかった者の割合は)0%であった．

4.2 評価

図 8 において，検出されなかったプログラムを目視レビューにより調べたところ，必要な DLL プログラムがロードされていないことが判明した．これは，プログラム解析に必要なプログラムを集めてくる部分で想定していないパターンが複数あったことを意味する．結果，図 3 における「Makefile 解析器」あるいは，「コード情報抽出器」において，解析に必要なファイルのロードパターンの情報を柔軟に取り込める必要があることが理解された．

一方，画面遷移が検出された実行プログラムに対しては，高い精度で (サンプル調査カバー率 100%) で画面遷移情報が検出できていた．同サンプルプログラムに対して，余分に検出されていた画面遷移を調べたところ，すべて [前画面ボタン] に対する遷移の余分な検出であり，誤検出の理

対象モジュール	.c,.cpp,.hのファイル数	総画面数	生成した実行プログラム数	生成したDLLプログラム数	画面遷移検出実行ファイル数
モジュールA	1504個	264個	177個	490個	36個

図 8 レガシー GUI アプリケーションの試験適用対象サブシステムのデータ

サンプル実行プログラムB	画面数	イベント	画面遷移数	画面遷移検出 True Positive	画面遷移検出 False Positive	画面遷移検出 False Negative
目視レビュー結果	7個	19個	26個			
プログラム解析結果	7個	19個	47個	26/47 [55%]	(47-26)/47 [45%]	0/47 [0%]

図 9 実行プログラムの一サンプルに対するレビュー結果とプログラム解析結果の比較

由は，各画面の [前画面ボタン] に設定されるイベント処理関数がすべて同じ関数であるためであることがわかった．この関数内では，前画面がどこかという情報を値として保持しておくことで，その値による分岐によって遷移先画面を制御しているが，実装したプログラム解析では，値による実行の分岐を保守的に扱っており，このために生じる誤検出である．必要であれば，この部分に対応することは，図 7 の中のフロー解析をより精度の良いものとすることである程度可能である．

結論として，事前のコードレビューでの印象を踏まえて実装した図 7 のレベルでのプログラム解析を使用することで，必要なコードを揃えることの出来た実行プログラムに対しては，十分な精度で画面遷移情報が検出できたと考えられる．また，解析に適用できる形で必要なコードをまとめる処理に対しても固有のロードパターンなどを取り込めることが望ましいことが理解された．

5. まとめ

C/C++言語で記述された GUI アプリケーションのソースコードから，画面遷移情報を抽出する datalog ベースのコード解析器を構築した．また，大規模なレガシー GUI アプリケーションの一部に試験適用し，比較的高い精度で画面遷移が検出できることを確認できた．構築したプログラム解析器は，直接的には，コードの AST に対する解析を行う．そのため，ソースコードに対する知見やレビューの情報をクエリのルールとして織り込むことが可能である．(例えば，特定のレベルのフロー解析で十分であるとか，共用関数 `f` の第一引数が必ず `show` されるのであれば，`f` を `show` と見なすことで画面遷移抽出の解析をショートカットできるなど).

今後の課題としては，4.2 節で述べた制限の改善が挙げられる．また，図 6,7 等から理解されるように，datalog を用いたクエリは柔軟である一方，一般的なエンジニアには理解し難い．このため，エンジニアが持つ知見を datalog のルールとして蓄積するための方法が必要である．

参考文献

[1] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using bdds. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2003.

[2] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, Vol. 39, 2004.

[3] Monica S Lam, John Whaley, and Micheal C. Martine. Contextsensitive program analysis as database queries. In *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems,(PODS05)*, 2005. Invited Tutorial.

[4] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, 2009.

[5] 福安直樹, 山本晋一郎, 阿草清滋. 最粒度リポジトリに基づいた case ツール・プラットホーム Sapid. 情報処理学会論文誌, Vol. 39, No. 6, 1998.

[6] 吉田一, 山本晋一郎, 阿草清滋. Xml を用いた汎用的な細粒度ソフトウェアリポジトリの実装. 情報処理学会論文誌, Vol. 141, No. 106, 2003.

[7] Eclipse CDT, `http://www.eclipse.org/cdt/`.

[8] bddbddb, `http://bddbddb.sourceforge.net/`.