



Technical Description V7.3

日本語版

©2004 LDRA Group of companies.

LDRA believes the information within this publication to be accurate at its publication date. However, the information is subject to change without notice and should not be construed as a commitment by either LDRA Ltd. or any subsidiaries. LDRA assumes no responsibility for any errors that may appear in this publication.

Please note, this document describes features that may not be available for certain language and platform versions. Trademarks:

LDRA recognises all brandnames and trademarks of other companies used in this document as their property.

Dual System™- Trademark of LDRA Ltd. and Subsidiaries.

LCSAJ™- Trademark of LDRA Ltd. and Subsidiaries.

LDRA Testbed®- Registered Trademark of LDRA Ltd.

Qualsys™- Trademark of LDRA Ltd. and Subsidiaries.

TBrun™- Trademark of LDRA Ltd. and Subsidiaries.

TBset™- Trademark of LDRA Ltd. and Subsidiaries.

TBevolve™- Trademark of LDRA Ltd. and Subsidiaries.

TBsafe™- Trademark of LDRA Ltd. and Subsidiaries.

Testbed™- Trademark of LDRA Ltd. and Subsidiaries.

The LDRA logo™- Trademark of LDRA Ltd. and Subsidiaries.

The LDRA Quality logo™- Trademark of LDRA Ltd. and Subsidiaries.

LDRA Technical Description.

Produced by the LDRA Group of Companies.



FUJI SETSUBI

Static Analysis	6
メイン静的解析 –Main Static Analysis	6
解析範囲 Analysis Scope	6
ファイル包括 –File Inclusion	6
マクロ展開 –Macro Expansion	6
コードリフォーマット –Code Reformatting	6
コーディングルールチェック –Programming Standards Checking	7
MISRA C/ MISRA-C:2004 Checking (C/C++ only)	7
DERA C (C/C++ only)	7
複雑度解析 –Complexity Analysis	8
複雑度の尺度生成 –Complexity Metric Production	8
ノット解析 –Control Flow Knots	8
サイクロマティック複雑度 –Cyclomatic Complexity	8
到達可能性 –Reachability	8
ループ階層 –Looping Depth	8
LCSAJ –LCSAJ Density	9
コメント –Comments	9
Halstead –Halstead’s Metrics	9
構造化プログラミング検証 –Structuredness–Structured Programming Verification	9
オブジェクト指向 –Object Orientated Metrics (C++ only)	10
ファンイン/ファンアウト –Fan In/Fan Out	10
コードとデータのグラフィカル表示 –Code and Data Graphical Visualisation	11
コールグラフ –Callgraphs	11
ダイナミックコールグラフ –Dynamic Callgraph	11
フローグラフ表示 –Flowgraph Displays	11
ダイナミックフローグラフ機能 –Dynamic Flowgraphs	11
アクティブフローグラフ機能 –Active Flowgraph	11
バーチャート –Bar Charts	11
キビアット図 –Kiviat Diagram	11
品質レポート –Quality Report	12
メトリクスレポート –Metrics Report	13
静的解析と複雑度解析 まとめ –Static and Complexity Analysis Summary	13

スタティックデータフロー解析	— Static Data Flow Analysis	14
関数コール情報	— Procedure Call Information	14
データフロー異常	— Data Flow Anomalies	14
データフロー異常メッセージ	— Data Flow Anomaly Messages	15
関数インターフェイス解析	— Procedure Interface Analysis	15
関数の引数解析	— Procedure Parameter Analysis	15
グローバル変数解析	— Global Variable Analysis	16
関数値解析	— Function Value Analysis	16
グローバルデータフロー解析	— Global Data Flow Analysis	16
スタティックデータフロー解析 まとめ	— Static Data Flow Analysis Summary	16
クロスリファレンス	— Cross Reference	17
クロスリファレンス まとめ	— Cross Reference Summary	17
インフォメーションフロー解析	— Information Flow Analysis (part of TBsafe™)	18
データオブジェクト解析レポート	— Data Object Analysis Report	20
解析内容	— Contributing Analysis Phases	20
変数ドキュメント	— Variable Documentation	20
変数タイプレポート	— Variable Type Reporting	20
変数属性レポート	— Variable Attribute Reporting	21
構造化要素に対するドキュメント	— Structure Element Documentation	21
その他のドキュメント	— Other Documentation	21
コードドキュメンテーションレポート	— Code Documentation Reports (C/C++ only)	22
概要	— Overview	22
関数のパラメータとグローバルレポート	— Procedure Parameters and Globals Report	22
自動ヘッダコメント生成	— Automatic Header Comment Generator	22
クラス階層レポート	— Class Hierarchy Report (C++ only)	22
ユーザ定義タイプレポート	— User Defined Types Report	23
Instrumentation		24
概要	— Overview	24
ホスト/ターゲットテスト	— Host / Target Testing	24
セマンティック解析	— Exact Semantic Analysis (part of TBsafe™)	25
セマンティック解析 まとめ	— Exact Semantic Analysis Summary	25

Dynamic Analysis	26
動的解析 –Dynamic Coverage Analysis	26
LDRA Testbed コードカバレッジ解析機能 –LDRA Testbed Code Coverage Capabilities	27
ステートメントカバレッジ 100% –Statement Coverage (TER1)	27
ブランチ/デシジョンカバレッジ 100% –Branch/Decision Coverage (TER2)	27
LCSAJカバレッジ 100% –LCSAJ Coverage (TER3)	27
関数/関数コールカバレッジ 100% –Procedure/Function Call Coverage (P/F CALL)	28
ブランチコンディションカバレッジ 100% –Branch Condition Coverage (BCC)	28
ブランチコンディションコンビネーションカバレッジ 100% –Branch Condition Combination Coverage (BCCC)	28
モディファイドコンディション/デシジョンカバレッジ 100% –Modified Condition/Decision Coverage (MC/DC)	28
コードカバレッジ測定 –Measuring Code Coverage	28
カバレッジレポート –Coverage Reporting	29
動カバレッジ解析 まとめ –Dynamic Coverage Analysis summary	29
ダイナミックデータフローカバレッジ –Dynamic Data Flow Coverage	30
データセット解析 –Data Set Analysis	31
データセット解析 まとめ –Data Set Analysis Summary	31
プロファイル解析 –Profile Analysis	32
プロファイル解析 まとめ –Profile Analysis Summary	32
TBsafe™ –High Integrity Code Testing (DO-178B)	33
概要 –Overview	33
TBsafe要約 –TBsafe Summary	33
インフォメーションフロー解析 –Information Flow Analysis	33
セマンティック解析 –Exact Semantic Analysis	33
MC/DCカバレッジ –MC/DC Coverage	33
安全性への規約 –Safe Subsets	33
TBrun™ –Test Harness Generator	34
概要 –Overview	34
TBrunを使ったテスト生成 –Creating Tests with TBrun	34
リグレッションテスト –Regression Testing	35
スタブ –Stubbing	35
TBrun Features	36
ドライバプログラムの自動生成 –Automatically Generated Driver Program	36
スタブ生成 –Stub Creation	36
構造体/配列/共用体 –Structures/Array/Unions	36

クラスの扱い —Class Handling	37
テンプレートタイプの自動解決 —Automatic Resolution of Templated Type	37
例外処理 —Exception Handling	37
ポインタの扱い —Pointer Handling	37
自動生成とオブジェクトの再利用 —Automatic Creation & Object “Re-Use”	38
その他の自動処理される言語の機能	38
大規模システムでのユニットテスト —Unit Testing Large Systems	38
TBrun まとめ —TBrun Summary	38
TBevolve™	39
概要 —Overview	39

LDRA Testbed Tool Suite Technical Description



Static Analysis

メイン静的解析 – Main Static Analysis

LDRA Testbed は全ての解析の土台として、単ファイル、もしくはシステムレベルで静的解析を行います。

解析範囲 – Analysis Scope

LDRA Testbed は、独自の構文解析技術から、高度で柔軟性に優れた解析を提供しています。ツールによる制約は少なく、テストに集中する事ができるようになります。 LDRA Testbed で解析できるものは、以下になります。

- ・ ソースファイル
- ・ ソースコード、ローカルヘッダファイル
- ・ コンパイラヘッダファイル
- ・ 外部ライブラリのソースファイル
- ・ コンパイラによりプリプロセスされたファイル

```
/// This is a part of the Microsoft Foundation
/// Copyright (C) 1992-1997 Microsoft Corporat
/// All rights reserved.
///
/// This source code is only intended as a sup
/// Microsoft Foundation Classes Reference and
/// electronic documentation provided with the
/// See these sources for detailed informatio
/// Microsoft Foundation Classes product.

#include "stdafx.h"
#include "Scribble.h"

#include "MainFrm.h"
#include "ChildFrm.h"
#include "IpFrame.h"
#include "ScribDoc.h"
#include "ScribVw.h"
```

```
#define make_nname(x) sprintf(n##x, "name" #x )
/* Problems can arise
 * with unspecified
 * their use can cau

#define mac() 0
/* Parameterised macros used in
 could cause confusion and era

#if 0 UNDEFINED
int h = 1; /* Could cause unpredictable
 intention is not clear */
#endif

#ifdef mac
int j = 0;
#endif
/* Extraneous characters could cause confusion
```

LDRA Testbed は、下記どちらのレベルにも対応しています:

- ・ ユニットテスト: 単関数/機能 or ソースファイル or 関連しないソースファイル群
- ・ インテグレーション/システム/サブシステムテスト: 関連するソースファイル群

ファイル包括 – File Inclusion

LDRA Testbed は、ユーザによって指定されたヘッダファイルも取扱えます。(設定が非常に柔軟で自由度が高い)

マクロ展開 – Macro Expansion

LDRA Testbed には、マクロ展開機能(ユーザ設定可能な条件付きコンパイル設定)があり、各ターゲットごとのコンパイル環境でのテストが容易に実現できます。

コードリフォーマット – Code Reformatting

LDRA Testbed は、テスト対象コード(左図)のコピーをリフォーマット(要素毎に分解)します。リフォーマットされたコード(右図)は全測定において解析/利用されます。これにより、プロジェクトを通して全てのコードが一貫して系統だった方法で測定されます。解析結果は元のソースコード、リフォーマットされたコード、それら両方での参照が可能です。

```
/* TRIANGLE - an ANSI C triangle example */
#include <stdio.h>
void main ( )
{
    int i,j,k,match,ncases;
    char *message ;
    printf("Enter number of triangles \n");
    scanf ( "%d", &ncases ) ;
    /* start loop */
    while
    {
        ncases --;
        printf("nEnter sides of triangle\n");
        scanf ( "%d %d %d", &i , &j , &k );
        /*LDRAssert*/
        /* i>0 && j> 0 && k>0
        /*LDRAssert*/
        if
        {
            i >= j + k || j >= k + i || k >= i + j
        }
        {
            message = "not a triangle";
        }
        else
        {
            match = 0 ;
            if
            {
                (

```

```
17/*
18CFC TESTBED VERSION :
193FFILE UNDER TEST : "E:\C_CPP_Testbed\triangle.c"
204DATE OF ANALYSIS :
215P:
226F
237/* TRIANGLE - an ANSI C triangle example */
248#include <stdio.h>
259
2610PVOID
2711T main()
2812 {
2913F int
3014F i ,
3115F j ,
3216F k ,
3317F match ,
3418F ncases ;
3519F char *
3620F message ;
3721F printf ( "Enter number of triangles \n" ) ;
3822T scanf ( "%d" , &ncases ) ;
3923F /* start loop */
4024T while
4125T {

```

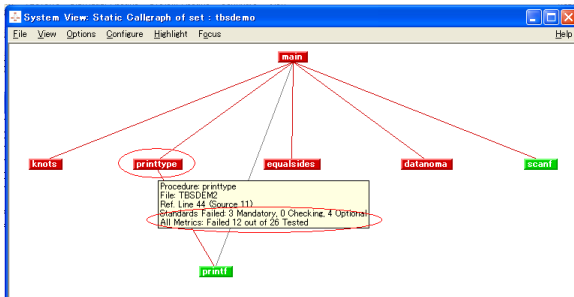
コーディングルールチェック – Programming Standards Checking

LDRA Testbed の静的解析機能により、ソースコード上のコーディング規約は自動で解析されます。

コーディング違反を検出する仕組みは、柔軟に定義できます：

- ユーザ定義フィルター：各種スタンダードのオン・オフ
- 各規約の定義(必須、任意など)の変更が可能
- 特定コード行や、特定コード領域への違反チェックを停止させることが可能(独自コメント文をコードに挿入)
- 依頼に応じて新しいコーディング規約へ対応可能(ルール追加など)

LDRA Testbed は、選択されたコーディング規約の全ての違反をテキストとグラフィカル表示への注釈でレポートします。



MISRA C / MISRA-C:2004 Checking (C/C++ only)

MISRA C と MISRA C: 2004 は、安全なシステム開発の実装基準として、コーディングルールやその他の指標を定めた国際的な標準規格です。

LDRA Testbed は、MISRA C チェック機能を持ち、準拠で求められる以下x3種全ての評価ができる唯一のツールです。

- コーディングルール
- 複雑度の尺度
- コードカバレッジ

LDRA Testbed は、MISRA のプログラミング標準に基づいて

- 必須 (Required)
- 忠告 (Advisory)

等をレポートします。

Number of Violations	LDRA Code	Required Standards	MISRA Code
0	T1 S	No brackets to loop body.	MISRA 59
59	S1 S	Use of comma operator.	MISRA 59
59	S1 S	Quality comparison of float.	MISRA 56
59	S1 S	Null statement found.	MISRA 56

DERA C (C/C++ only)

DERA Cは、セーフティクリティカルシステムに対する MISRA C 標準の拡張版です。

UK(英国) MOD(Military of Defence) でLDRAの協力を元に制定されました。

DERA C では、高度なソースコードデータフロー、インフォメーションフロー解析が求められ、

LDRAが唯一対応出来ています。 ※DERA : Defence Evaluation and Research Agency

複雑度解析 —Complexity Analysis

複雑度解析は関数ベースで、関数毎の基底構造を解析し、その結果は関数、ファイル、そして全システムに渡って分析されレポートされます。

複雑度の尺度生成 —Complexity Metric Production

LDRA Testbed は、以下の複雑度の尺度を生成し、ソフトウェア製品の品質(可読性、メンテナンス性、テスト容易性)を評価するために使用されます。

File	All Metrics	Clarity	Maintainability	Testability
Tbsdem3.c	85	71	91	100
Tbsdem2.c	76	57	73	93
Tbsdem1.c	89	79		93

79. 1. Declaration Comments/Exe. Lines
2. Comments in Headers
3. Total Comments

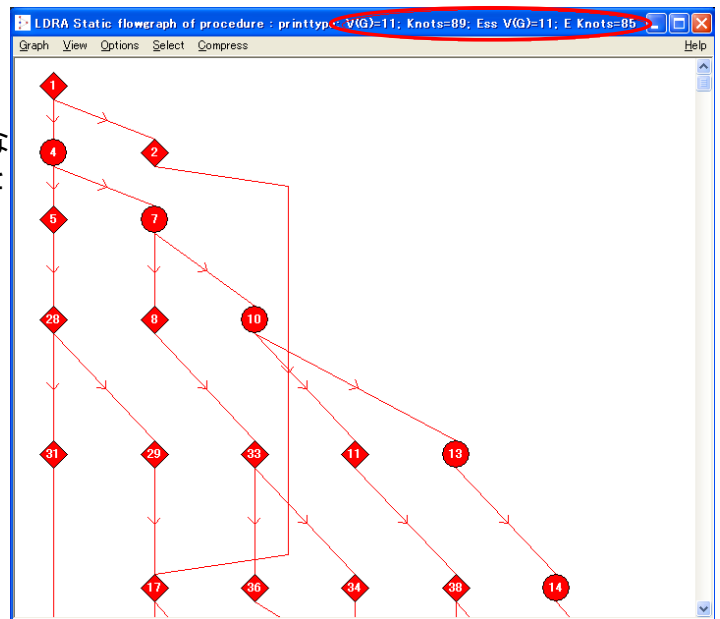
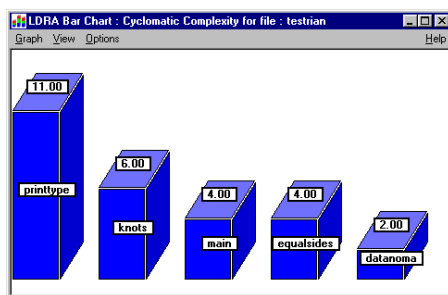
ノット解析 —Control Flow Knots

ノット(Knot metric)は、プログラムの実行制御構造に対する複雑さの一要因です。ノットにより支離滅裂な、つまりコード上あちらこちらへ飛んでいる部分を解析します。過度のノット値が測定された場合、プログラムの可読性を向上させるためにプログラムの再構成など複雑さを下げる工夫が求められるでしょう。

サイクロマティック複雑度 —Cyclomatic Complexity

サイクロマティックは、プログラムの制御フローグラフに着目した複雑度解析です。(サイクロマティック $V(G) = \text{グラフ中のリンク数} - \text{グラフ中のノード数} + 2 \times \text{不連続なグラフ数}$) 尺度として、モジュール毎で10を超えないことを推奨します。この解析結果は、モジュールを再設計すべきかどうかの判断に使用されます。

これは、有向グラフ(directed graph: 要素間の関係を示した図)のサイズの測定であり、つまりは複雑度の要因になります。



到達可能性 —Reachability

全ての実行行は、プログラムの開始から制御フローパスを通過して到達可能であるべきでしょう。

到達され得ないコードは、そのようなパスを持たないステートメントで構成されます。LDRA Testbed は、これらの行を "Unreachable" としてマークし抽出します。これらはプログラム実行に何の影響もないので、障害なく削除する事が出来ます。

ループ階層 —Looping Depth

制御フローループの最大の深さは、可読性、複雑度、コード効率の要因です。

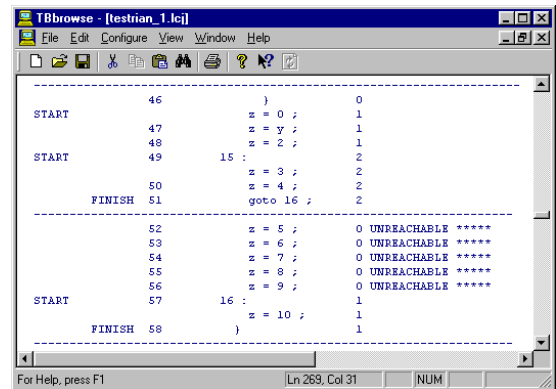
LCSAJ (Linear Code Sequence and Jump)

LCSAJ パス密度は、メンテナンス性の尺度です。

1行のコードを変更しようとした場合にどれだけのテストパス(LCSAJs) が影響を受けるかを知らせます。

1行のコードに対してLCSAJ密度が高いと、変更による全てのテストパスへの信頼度が低下します。

それゆえ多くのリグレッションテストが必要となります。

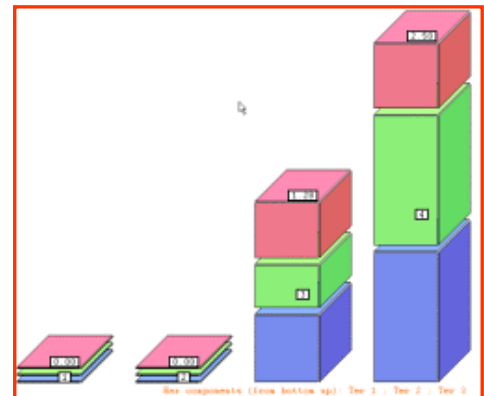


```
START 46      ) 0
47      z = 0 ; 1
48      z = y ; 1
49      z = 2 ; 1
START 49      1.5 : 2
50      z = 3 ; 2
51      z = 4 ; 2
FINISH 51     goto 1.6 ; 2
-----
52      z = 5 ; 0 UNREACHABLE *****
53      z = 6 ; 0 UNREACHABLE *****
54      z = 7 ; 0 UNREACHABLE *****
55      z = 8 ; 0 UNREACHABLE *****
56      z = 9 ; 0 UNREACHABLE *****
START 57      1.6 : 1
58      z = 10 ; 1
FINISH 58     ) 1
```

コメント — Comments

コメントによる可読性、メンテナンス性の評価。

- 関数宣言前のコメント行数(関数のヘッダー)
- コメント内の全空白行数
- 関数の宣言部分のコメント行数
- 関数の実行部分のコメント行数



Halstead — Halstead's Metrics

Maurice Halstead氏により提案されたプログラムサイズの尺度です。

- 全算術演算子 /Total Operators
- 全変数・定数 /Total Operands
- ユニークな算術演算子 /Unique Operators
- ユニークな変数・定数 /Unique Operands
- 語彙 /Vocabulary
- プログラム長 /Length
- 情報量 /Volume

構造化プログラミング検証 — Structured Programming Verification

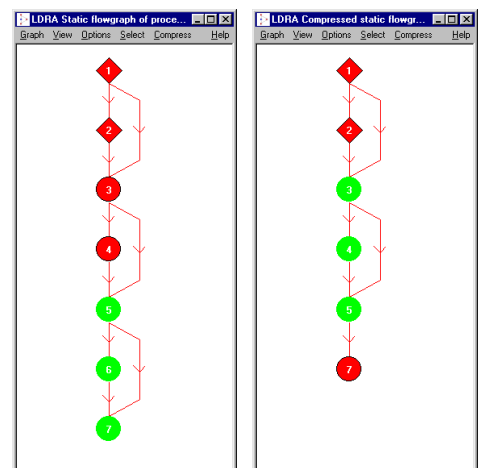
LDRA Testbed の構造化プログラミング検証(SPV)により、いかに適正にプログラムが構造化されているかを判定します。LDRA Testbed は、テンプレートを用いて、対象プログラム内の制御文(例:if then else、do while、for など)を評価します。

標準で用意されるテンプレート内の制御文は、削除・修正可能です。

新たな制御文も必要に応じて追加できます。

テンプレートを参照することで、(例えば、開発チーム内で)使用可能な制御文が確認できます。

使用可能な制御文を除いた結果、フローグラフが単一ノードに縮小されれば、プログラムは正しく構造化されていると言えます。



もし対象プログラムが正しく構造化されていない場合、Testbedは“Essential Cyclomatic Complexity” > 1 をレポートします。“Essential Cyclomatic Complexity” は、残差グラフ(残差=観測値 - 予測値)の式を適用して算出されます。Essential Knotsも、構造化されていないことの尺度として得られます。適切に構造化されたプログラムは、“Essential Knots”=0、“Essential Cyclomatic Complexity”=1 になります。

オブジェクト指向に対する尺度 – Object Orientated (OO) Metrics (C++ only)

LDRA Testbed は、業界標準である“Chidamber & Kemerer”を含む多くのOO尺度に対応しています。

ソースファイルレベルのOO尺度

- ソースファイル内の全クラス宣言数
- ソースファイル内の全オブジェクト宣言数

クラスレベルのOO尺度

- クラスタイプ (Prolog class, Abstract class, Handle/Envelope class with pointers to classes, Standard class)
- 宣言されたクラスのオブジェクト数
- クラスのインスタンス(宣言)としてのオブジェクト
- クラス内に宣言されたデータメンバ数
- クラスのメンバ数 (WMC)
- 派生クラスの数
- 基本クラスの数

Class	Type	Objects Created	Number of Data Members
CItem	S	4 (P)	4 (P)
CDispenser	S	0 (P)	1 (P)
CDrinksDispenser	S	1 (P)	4 (P)

[\[Top of Report \]](#)

Class Level OO Metrics - with Base Classes (dispense.cpp)

Class	Total Members	Depth of Inheritance
CItem	7 (P)	0 (P)
CDispenser	11 (P)	0 (P)
CDrinksDispenser	18 (P)	1 (P)

クラスレベルのOO尺度 – 基本クラス情報

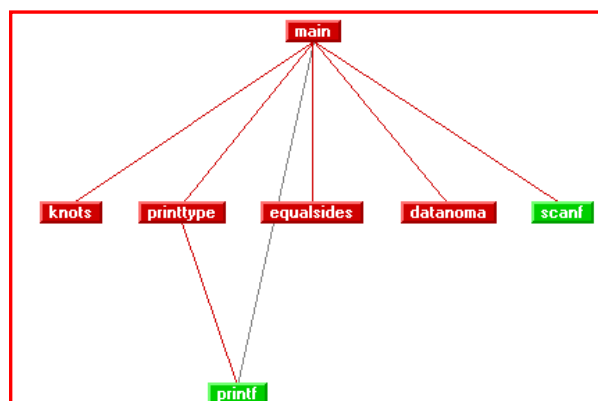
- 各クラス内の基本クラス数 – The total number of base classes for this class
- クラス内のデータメンバ数 – The total number of data members in the class
- クラスの全メンバ数 – The total number of all members of the class
- クラスの継承の深さ (DIT) (Chidamber & Kemerer)
- スタティックメンバ数 – The number of static members
- クラス外変数使用の数 – The number of out of class variable uses
- クラス外関数コールの数 – The number of out of class procedure calls
- 外部クラスメソッド使用数 – Number of external class methods used
- メソッド内のクラス外オブジェクトの数 – Number of out of class objects in methods

ファンイン/ファンアウト – Fan In/Fan Out

Fan In は、関数が他の関数からコールされる数です。

Fan Out は、他の関数へのコール数です。

コールグラフの複雑度の要因です。

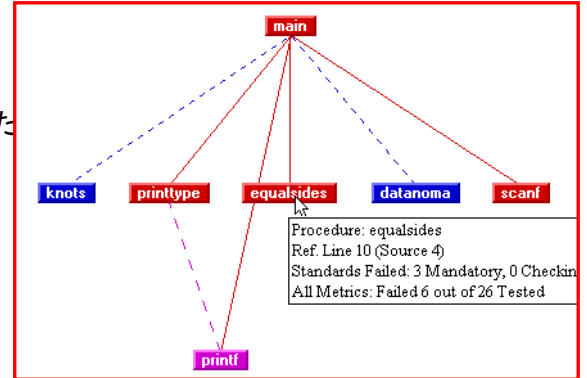


コードとデータのグラフィカル表示 – Code and Data Graphical Visualisation

主要な静的解析と複雑度解析は、統合されグラフィカルに表示されます。

コールグラフ – Callgraphs

関数間の階層関係を示します。ソースファイル、システム単位で表示され得ます。システムコールはグラフから追加、削除することができます。1つの関数に着目してグラフを見ることもできます。コールグラフから関数のフローグラフも表示でき、より深く掘り下げた解析も可能です。関数の引数、グローバル変数の使用有無などの情報も確認できます。

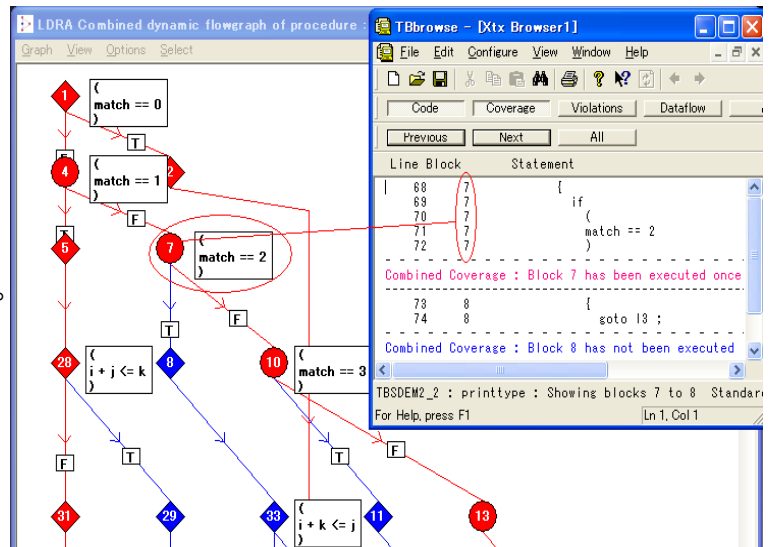


ダイナミックコールグラフ:

コールグラフにカバレッジ情報を重ね合わせて、動的カバレッジ結果を視覚的に色分け表示します(ノードとノード間を、一部実行でピンク、全て実行で赤、実行されない領域は青色で表示) 関数内のカバレッジに関して、ソースコードの色分けでも表示できます。

フローグラフ表示 – Flowgraph Displays

フローグラフは、各関数の制御フロー構造を分析して表示します。各ソースコード領域は、ノードとノード間の分岐パスで表現されます(右図)。フローグラフからソースコードや、分岐や関連情報などの注釈をグラフ上に展開して表示することが出来ます。また、丸のノードはコーディング規約違反がなく、ひし形ノードには違反が含まれることを表しています。



ダイナミックフローグラフ機能:

カバレッジ結果をノードとブランチで色分けして表示。実行済みで赤、実行されない場合は青色。

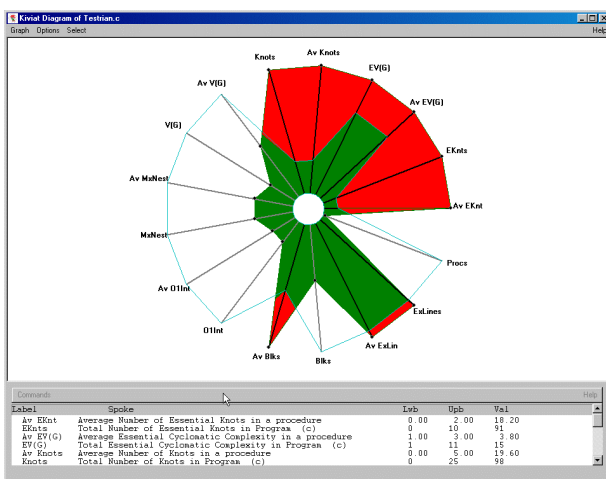
アクティブフローグラフ機能:

ダイナミックフロー表示の一種で、最後に実行したテストの制御フローを色分けして表示します。

バーチャート – Bar Charts

棒グラフにより開発者や管理者に、プログラムの品質、構造、テストレベルのプロファイルを直感的に提供します。

キビアット図 – Kiviat Diagram



キビアット図は様々な尺度を基にグラフ化し、ソースコードが基準に順守しているかを表示します。もし、各尺度においてキビアット図が緑色なら合格です。赤色部分が有れば、追加のテスト、あるいはコーディング規約に適合されるような修正が必要であることを示します。

品質レポート - Quality Report

Quality Reportは、解析されたソースコードの品質をおまとめ表示します。単ファイル、全システム、相関がないソースファイルの集まり(グループ)等をASCII or HTMLフォーマットでレポートします。レポートは、ファイルやシステム、グループのPass/Fail結果を生成し、詳細を表示します。

見出しは、レポートの構成、生成日時、解析範囲の詳細です。

LDRA Testbed [®] Quality Report

Set: scribble

Report Configuration

- Reporting Level: Summary Report
- Procedure Details: All PASSES
- Standards Violation Details: Show Violation Only

Report Production

- C++ LDRA Testbed Version: 5.2.1
- Report Produced On: 11/23/97 at 20:42:25
- Penalty File: c:\testbed\cpen.dat

Contributing Analysis Phases

- Static Analysis: Yes
- Complexity Analysis: Yes
- Static Dataflow: Yes

Pass/Fail 結果表示を判りやすく。

Overall Result: FAIL

HTML表示ではハイパーリンクを使って各関数ごとのレポートへジャンプします。

Quality Result	Procedure	Source File
Pass	Global Program	
Pass	CChildFrame::CChildFrame	CHILDFRM.CPP
FAIL C	CChildFrame::~CChildFrame	CHILDFRM.CPP
FAIL C	CChildFrame::PreCreateWindow	CHILDFRM.CPP
FAIL C	CChildFrame::AssertValid	CHILDFRM.CPP
FAIL C	CChildFrame::Dump	CHILDFRM.CPP
FAIL	CChildFrame::OnCreateClient	CHILDFRM.CPP
Pass	CInPlaceFrame::CInPlaceFrame	IPFRAME.CPP
FAIL C	CInPlaceFrame::~CInPlaceFrame	IPFRAME.CPP

解析範囲内の各関数は、Pass/Failリストで詳細化されます。

Procedure CScribbleDoc::CScribbleDoc (SCRIBDOC.CPP) - Pass

メトリクスレポート — Metrics Report

複雑度の尺度をレポートします。各メトリクスでファイルごと、関数ごとに分けて、値が品質基準をパスしているかをレポートします。レポート冒頭は測定された尺度のリストです。(右図)

各尺度は、品質基準を満たすと緑色、未到達だと赤色で表示されます。(下図参照)

List of Metrics to be Displayed

Complexity Metrics

- Knots, Cyclomatic Complexity, Essential Knots, Essential Procedure Structured (SPV).

Halsteads Metrics

- Total Operators, Total Operands, Unique Operators, Unique Vocabulary, Length, Volume.

Loop/Interval Analysis

- Number of Loops, Depth of Loop Nesting, Number of Op

Procedure	Cyclomatic	Essential	Ess. Cycl.	Structured	
	<u>Knots</u>	<u>Complexity</u>	<u>Knots</u>	<u>Complexity</u>	<u>Proc (SPV)</u>
equality	0 (P)	4 (P)	0 (P)	1 (P)	Yes (P)
printtype	89 (F)	11 (F)	85 (F)	11 (F)	No (F)
datanoma	1 (P)	2 (P)	0 (P)	1 (P)	Yes (P)
knots	7 (F)	6 (P)	6 (F)	5 (F)	No (F)

静的解析と複雑度解析 まとめ — Static and Complexity Analysis Summary

- コーディング規約チェック
- マクロ展開
- 複雑度の尺度
- 構造化プログラミングの検証
- 静的フローグラフ
- コールグラフ

静的解析、複雑度解析は、コーディング規約に基づいているか、ソフトウェアが適正に構造化されているか、複雑度、その他品質特性が構造化可能な品質モデルであるか、などを確かにするための情報を提供します。またこれらにより、相当なソフトウェアの欠陥を検出することができます。

静的解析は、ソフトウェア構造を文書化する基盤になっています。各関数や関数内部リンク(シングルファイルやシステムなど)への全制御フロー情報を生成し、ループ構造は明らかにされ、複雑度の尺度が生成されます。

スタティック・データフロー解析 — Static Data Flow Analysis

データフロー解析は、ソースファイルやシステムに対する、4種の情報を生成します。

- 関数コール情報
- データフロー異常
- 関数の引数解析
- グローバル変数解析

関数コール情報 — Procedure Call Information

プログラムの関数コール構造についての情報。プログラム内の各関数は順に解析され、他の関数とのコール (to/from) をリスト化します。もし関数がどの関数もコールしていなければ、それに相当したメッセージを表示 (右図) します。

再帰呼び出しも全て解析され、どこで再帰が起こっているか詳細を示します。

```
*****
 * Procedure Call Information *
*****

Call Lists:
-----
All calls are detailed. This includes calls internal to this file
as well as calls to system or other routines.

Called by Lists:
-----
Only calls by routines in this file are detailed.

*****
 * PROCEDURE equalsides *
*****

BETWEEN LINES 4 AND 10
DOES NOT CALL ANY PROCEDURES
IS CALLED BY THE FOLLOWING PROCEDURES
main

*****
 * PROCEDURE printtype *
*****

BETWEEN LINES 12 AND 37
CALLS THE FOLLOWING PROCEDURES
printf
```

データフロー異常 — Data Flow Anomalies

データフロー異常はエラーの要因になりがちな、プログラム内変数の一連の動作です。

例えば、プログラム内の変数 V に対して、次の3種の動作があります。

- V が定義 (Defined) される — 値の割り当てなど
- V が参照 (Referenced) される — 値の使用など
- V が未定義 (Undefined) になる — 値が破棄されるなど
(例えば、変数が範囲から外れた場合。変数が最初に宣言される時、未定義である。など)

これら3つの動作を、それぞれ D、R、Uとします。

これを基にして、3種のデータフロー異常が検知されます：

- 値が未定義の変数が参照される (UR anomaly)
- 定義されている変数が参照されること無しに再定義される (DD anomaly)
- 定義されている変数が参照されること無く未定義にされる (未使用) (DU anomaly)

以下の例を考察してみます：

```
1 procedure PROC is
2     t, x, y, z : integer;
3 begin
4     x := 1;
5     if y > 0 then
6         x := 2;
7     end if;
8     z := x + 1;
9 end PROC;
```

- 変数 X は4行目で定義され、6行目で再定義されています。DD異常 (DD anomaly)
- 変数 Y は、5行目で参照されますが、その参照前に値が割り当てられていない。UR異常 (UR anomaly)
- 変数 Z は、8行目で定義されますが、参照無しで未定義にされる (関数終了でスコープ外)。DU異常 (DU anomaly)
- 変数 t は、2行目で宣言されるが使用されることが無い。 (non-use)

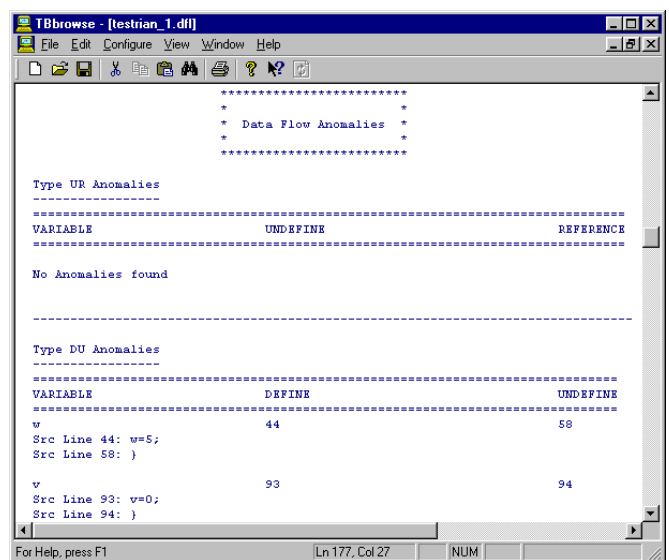
UR異常はプログラム上、正真正銘のエラーであるのに対し、DD、DU異常は疑わしい変数の使用で、必ずしもエラーとは限りません。上記プログラムでは、Xの用法は何も悪くありません。データフロー異常の検出には、プログラム上の全てのパスが実行されるものと仮定しています。決して実行されないパスに対しては異常として検出される可能性があります。例えば、もし上記プログラムの3行目の後に `y:= 0` が挿入されて2つ目の代入 (`x:=2`) は実行されなくなっても、変数Xに対するDDデータフロー異常はレポートされます。

データフロー異常のメッセージ – Data Flow Anomaly Messages

データフロー解析は、上記3種のエラーを検出して報告します。各タイプ別でグループ化され、UR、DU、DDの順でレポートされます。各メッセージには、変数名、行番号などが含まれます。

関数コールの結果として検出される異常があれば、そのことも報告します。さらに、宣言されているが使用されない変数も報告されます。

データフロー解析は、ソースファイル内の関数間や、システムをまたいで実施されます。



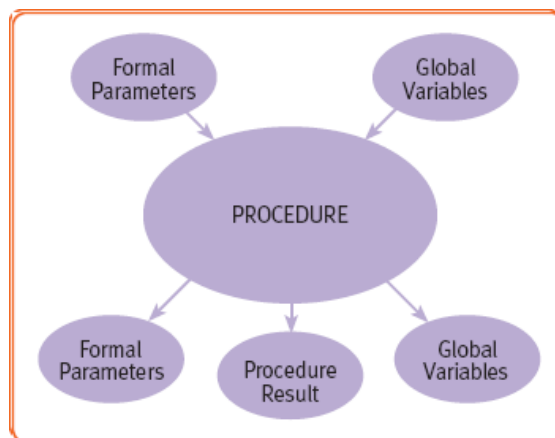
関数インターフェイス解析 – Procedure Interface Analysis

ソフトウェアの品質を左右する顕著な部分は、関数間の相互作用をコントロールし制限する関数インターフェイスです。LDRA Testbed は、全関数の引数、グローバル変数、関数ごとのローカル変数など全てを解析します。

関数の引数解析 – Procedure Parameter Analysis

各関数ごとに引数は解析され、そのタイプを以下のいずれかに分析します。

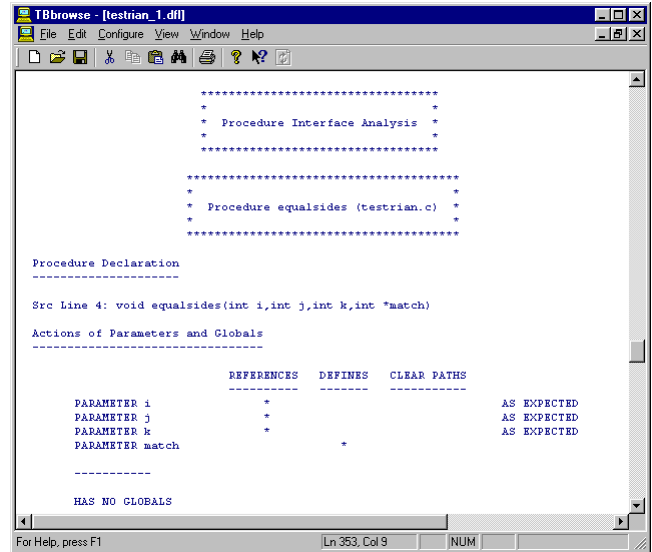
- 参照のみ (関数内で使用されるが変更される事はない)
- 定義のみ (値が割り当てられるが使用されない)
- 参照も、定義もされる
- 関数内で使用されない



もしアウトプットのパラメータが、あるパスに対してのみに値を設定する場合、定義されないパスがあることがレポートされます(clear paths)。これは、それに続くプロセスが適正な値を期待出来ないことを意味します。同様にインプットパラメータには、リファレンスされないパス(clear paths 決して使用されないパス)も存在するでしょう。これらの異常もレポートされます。

解析は関数の境界に対して実施されます。このように変数(別の関数や、その関数内へパスされるものや、パラメータとして他へ渡されるもの)は、各関数ごとにその用法で分類されます。再帰呼び出しにも、この解析が行われます。

異常な宣言もまた警告されます。従って、代入可能なパラメータが宣言されたにも関わらず、それへの代入が発生しなければ指摘されます。同様にパラメータが値を代入されているにも関わらず、その値が関数インターフェイス間で戻されない場合もレポートされます。



```
*****
* Procedure Interface Analysis *
*****

Procedure Declaration
-----
Src Line 4: void equalsides(int i,int j,int k,int *match)

Actions of Parameters and Globals
-----



|                 | REFERENCES | DEFINES | CLEAR PATHS | AS EXPECTED |
|-----------------|------------|---------|-------------|-------------|
| PARAMETER i     | +          |         |             | AS EXPECTED |
| PARAMETER j     | +          |         |             | AS EXPECTED |
| PARAMETER k     | +          |         |             | AS EXPECTED |
| PARAMETER match |            | +       |             |             |



-----
HAS NO GLOBALS
```

グローバル変数解析 - Global Variable Analysis

関数内で使用されるグローバル変数の解析。これは関数インターフェイス解析を補完するもので、同様のレポートが生成されます。

- 参照のみ(値は使用されるが、関数内で変更される事はない)
- 定義のみ(関数内で値が与えられるだけ)
- 関数内で参照され、定義もされる

関数値解析 - Function Value Analysis

関数は値を返します。通常戻り値は、リターン命令か関数名の指定で生成されます。不注意から戻り値を生成しないパスを関数内に生じてしまうことがあります。LDRA Testbed は、関数内の全パスをトレースし、そのようなパス(clear paths)をレポートします。これらはほとんどがエラーです。

グローバルデータフロー解析 - Global Data Flow Analysis

グローバルデータフロー解析は、各ソースファイルごと、全システムにまたがって行われます。

スタティックデータフロー解析 まとめ - Static Data Flow Analysis Summary

- 関数コール情報
- データフロー異常レポート
- 関数インターフェイス解析と異常レポート

データフロー解析で、変数用法のパターンが検査され、インターフェイスの詳細が全体に渡ってドキュメント化されます。あらゆる異常はハイライトされ、不具合を修正する為の参考になります。

TBsafeオプションであるインフォメーションフロー解析では、更に踏み込んだ(詳細な)レベルの解析を実現します。これは、自動ドキュメント生成、エラーや欠陥検出等に有力な機能です。

クロスリファレンス Cross Reference

ソースファイル、システム内で使用される全てのデータ項目の全面的なクロスリファレンスを行い、グローバル、ローカル、パラメータのどのタイプかをレポートします。また、解析されるプログラムの完全なコールツリーをテキスト形式で表示します。

クロスリファレンスでは完全なコールツリーが始めに生成されます。これは関数対関数ベースです。

- ある関数からコールされる他の全関数
- ある関数をコールする全関数

そして、ソースコード内で使用される全てのデータ項目の全面的なクロスリファレンスを行い、これもまた関数対関数ベースで構成されます。各関数で使用される全データ項目は、名前、属性、行番号です。

(<item name> <attribute code> <list of line numbers>)

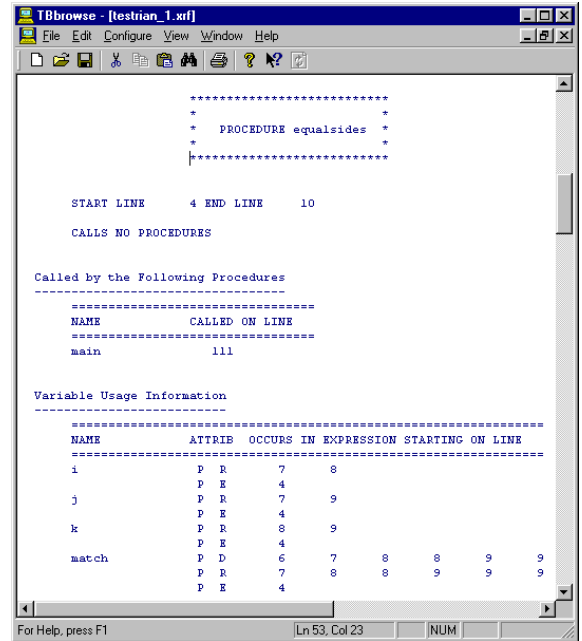
属性<attribute code> フィールドは以下のいずれかになります。

Code Meaning

- L ローカル変数(関数のスコープ内で宣言。他の場所では使用されない)
- G グローバル変数(他のプログラム内で宣言され、この関数内でグローバルとしてアクセスされる)
- P パラメータ(呼出し元関数からこの関数へ渡されるパラメータ) : 引数
- LG ローカルでありながらグローバルとして他の場所で使用(関数のスコープ内で宣言されるが、グローバル変数としてその他の関数内でアクセスされる) この場合、変数をアクセスする全関数と、使用している行が共にリストされます。:extern
- E 変数の宣言
- D 変数の定義
- R 変数の参照
- I 変数が入力として使用
- O 変数が出力として使用

クロスリファレンス まとめ Cross Reference Summary

クロスリファレンスは全ソースファイル間の変数使用をドキュメント化します。全システムに渡ってのクロスリファレンス解析も可能です。



インフォメーションフロー解析 (TBsafe™ オプション)

インフォメーションフロー解析(または変数依存関係分析 “variable dependency analysis”)は、プログラムの変数の相互依存性の解析です。LDRA Testbed は関数対関数ベースでこれらの依存性を解析します。

ある変数Bが変数Aを変更する要因になりえる場合、AはBに依存しています。中間媒介変数 (Intermediate variables) は、依存性のリストには現れません。入力変数のみです。例えば、もし変数BがCに依存する中間値であった場合、

```
B := C; A := B;
```

この変数AはCに依存しています(Bではなく)。以下、様々なタイプの依存性が分類されます。

強い依存 — Strongly dependent:

変数Aが定義される時、常に変数Bを前提とするような場合です。例えば、 $A = B + 1$ のような関数内のAへの割当てがBに依存する場合です。

弱い依存 — Weakly dependent:

変数Aは時々Bに依存、例えば関数内でAがBを参照して定義されるパスが少なくとも一つはあり、またBを参照しない場合もある場合。例: `if (condition) A = B + 1`

条件付依存 — Conditionally dependent:

変数Aは直接的にはBに依存しない、しかしBは実行フローパスによってはAに影響を及ぼす。例えば、`if (B > 0) A = 0`

更に以下2タイプの定義も確認します。

強い定義 — Strongly defined:

変数は必ず値を得る場合、強い定義とされる。例えば、関数内の全パスで変数の値が計算される。

弱い定義 — Weakly defined:

変数は必ず値を得るとは限らない場合、弱い定義とされる。例えば、関数内で少なくとも1つのパスで計算されない場合。

予測される依存情報をコメント文としてあらかじめコードに挿入しておくことで、インフォメーションフロー解析結果と比較され、想定内であるかの評価をすることが可能です。以下、コメント文の例です。(C, Ada)

```
--LDRA_INFOFLOW <output variable>[text]([[<input variable> {,<input variable>}]])
```

<output variable> は、出力変数名です。[text] はコメントで、<input variable> は出力変数に依存する入力変数名です。もしコメントが1行以上になる場合、2行目やそれ以降の行は、正当なコメントにしなければなりません。それぞれ別々のコメント行として記述します。

<Cの例>

```
/*LDRA_INFOFLOW j (i#sd i#sc)*/
```

i#sd: 変数j は、変数i に強く依存する。

i#sc: 変数j は、変数i に強く条件付依存する。

<Adaの例>

```
--LDRA_INFOFLOW xi (a,b,c)
```

```
--LDRA_INFOFLOW hiy depends on (y,z)
```

```
--LDRA_INFOFLOW DGSZ ( )
```

```
--LDRA_INFOFLOW G5Y depends on (P,Q,R)
```

```
--LDRA_INFOFLOW MANY depends on (ONE,TWO,  
--THREE,FOUR)
```

(インフォメーションフロー解析のためのサンプルコード)

```

TDbrowse
File Edit Configure View Window Help
program infoflow(input);
var a,b,c,d,e,f,g: integer;
begin
  read (a,c);
  if (c=1)
  then
  begin
    b:=a+7; d:=c+a; e:=a+1;
  end
  else
  begin
    d:=c+2; e:=1;
    if (c=2)
    then
    f:=a+1
    else
    f:=b+1;
    end;
  end;
end;
Ln 53, Col 23
NUM

```

デッドコード(アウトプット値に関わる事のないステートメントはインフォメーションフロー解析でレポートされます)

```

TDbrowse
File Edit Configure View Window Help
Strongly defined variables:
-----
Variable          Strongly directly      Weakly directly
                  Dependent              Dependent
-----
a                  directly depends on no other variables
c                  directly depends on no other variables
d                  c
e                  a

Variable          Strongly conditionally  Weakly conditionally
                  Dependent              Dependent
-----
a                  conditionally depends on no other variables
c                  conditionally depends on no other variables
d                  c
e                  c

Weakly defined variables:
-----
Variable          Strongly directly      Weakly directly
                  Dependent              Dependent
-----
b                  a
f                  a
                  b

Variable          Strongly conditionally  Weakly conditionally
                  Dependent              Dependent
-----
b                  c
f                  c
Ln 53, Col 23
NUM

```

データオブジェクト解析レポート Data Object Analysis Report

このレポートは、変数(定数も可)の全クロスリファレンスです。解析の範囲はフィルターで指定することも出来ます。LDRA Testbed のインフォメーションフロー解析オプションがあれば、各変数の前方及び後方の依存性もレポートします。レポートは、各変数がソースコードの何処で使用されるか等の詳細を表示します。

Variable	Alias	File	Procedure	Type	Attrib.	Used on lines...		
FALSE (P)		SCRIBDOC	CStroke::DrawStroke	G	R	212		
			CScribbleDoc::OnNewDocument	G	R	63		
			CScribbleDoc::InitDocument	G	R	144		
			CScribbleDoc::OnOpenDocument	G	R	118		
IDOK (P)		SCRIBDOC	CScribbleDoc::OnPenWidths	G	R	292		
PS_SOLID (P)		SCRIBDOC	CStroke::DrawStroke	G	R	209		
			CScribbleDoc::ReplacePen	G	R	262		
TRUE (P)		SCRIBDOC	CScribbleDoc::OnNewDocument	G	R	68		
			CScribbleDoc::OnEditCopy	G	R	373		
			CScribbleDoc::OnOpenDocument	G	R	123		
			CStroke::DrawStroke	G	R	229		

解析内容 — Contributing Analysis Phases

データオブジェクト解析はLDRA Testbed のオプション内容により、以下の情報を含むことが出来ます。

- クロスリファレンス
- スタティックデータフロー解析
- インフォメーションフロー解析

このレポートは、各ソースファイルごと、全システム単位に対しても行えます。

変数ドキュメント — Variable Documentation

変数タイプレポート — Variable Type Reporting

データオブジェクト解析レポート上で表記される変数タイプ。

- コンスタント : C
- ローカル : L
- グローバル : G
- パラメータ : P
- ローカル-グローバル : LG (外部変数 extern)
- クラスメンバー : M

変数属性レポート – Variable Attribute Reporting

データオブジェクト解析レポートで表記される変数属性です。

- 宣言 Declared
- 定義 Defined
- 参照 Referenced
- インプットとして使用
- アウトプットとして使用
- 間接使用 Indirectly Used – 間接使用は、構造体のメンバ(構成要素)に対する、もしくは解析されていない関数へのパラメータの使用に適応されます。

構造化要素に対するドキュメント – Structure Element Documentation

構造化要素が使用されている場合、データオブジェクト解析に表記される内訳。

- 構造体メンバ Structure member
- ユニオンメンバ Union member
- enumメンバ – 一覧 Enumeration member
(Enumeration は、あるオブジェクトに属するある要素を、すべて列挙して数え上げる時に用いられます)
- クラスメンバ Class member
- ネームスペースメンバ Name-space member
- テンプレートクラスメンバ Template class member

その他のドキュメント Other Documentation

データオブジェクト解析では以下も解析されドキュメント化されます。

- 関数の引数としてエイリアスされる変数
- 関数の戻り値

コードドキュメンテーションレポート Code Documentation Reports (C/C++ only)

概要 Overview

自動コードドキュメンテーションは、開発およびメンテナンス両方のコストを低減します。LDRA Testbed は、以下のドキュメント機能を持っています。

関数のパラメータとグローバルレポート – Procedure Parameters and Globals Report

コード解析で、関数のインプット・アウトプットとして使用されるパラメータ、グローバル変数、メンバー変数をレポートします。ドキュメント化される関数インターフェイスの要素は、以下です。

- インプットパラメータ Input Parameters
- インプットメンバー変数 Input Member Variables
- インプットグローバル変数 Input Globals
- アウトプットパラメータ Output Parameters
- アウトプットメンバー変数 Output Member Variables
- アウトプットグローバル変数 Output Globals
- インスタンス生成 Template Instantiations
- リターン型 Return Type

```
*****  
* Procedure CItem::SetAttributes (dispense.cpp) *  
*****  
  
Number of Template Instantiations : 0  
Number of Input Parameters : 5  
Number of Input Member Variables : 0  
Number of Input Globals : 0  
Number of Output Parameters : 0  
Number of Output Member Variables : 5  
Number of Output Globals : 0  
  
Procedure Returns : void  
  
Input Parameters  
-----  
NAME TYPE  
-----  
szName char *  
nCode int  
nStock int  
nMaximum int  
dPrice double  
  
Output Member Variables  
-----  
NAME TYPE  
-----  
m_szName[15] char  
m_nCode int  
m_nCode ..  
.....
```

自動ヘッダーコメント生成 – Automatic Header Comment Generator

各ファイルを解析し、ファイル及び関数間ベースのコメントを自動生成します。コメントは以下のように構築されます。

- ルーチンもしくはファイル名
- 関数へのパラメータ
- リターン型
- 使用されるグローバル変数
- 関数コール情報

```
****  
**** Routine: files_in_set_table  
****  
**** Parameters:  
****  
**** Action Name Type Use  
**** -----  
**** I out_file FILE *  
**** I file_list struct string_list *  
**** I file_int_list struct string_int_list *  
**** I set_type int  
**** I links int  
**** I out_type int  
****  
**** Return value:  
**** -----  
****  
**** Type Meaning  
**** -----  
**** int  
****  
**** Global Variables:  
**** -----  
****  
**** Action Name Type Imported  
**** -----  
**** B test_table_flag int Yes  
**** B glob_int_cols int Yes  
**** B convert_angbrack_flag int Yes  
**** B mem_fail_flag int Yes  
**** B reformat_flag int Yes
```

クラス階層レポート Class Hierarchy Report (C++ only)

ファイル内の各クラスの以下内容を記録します。

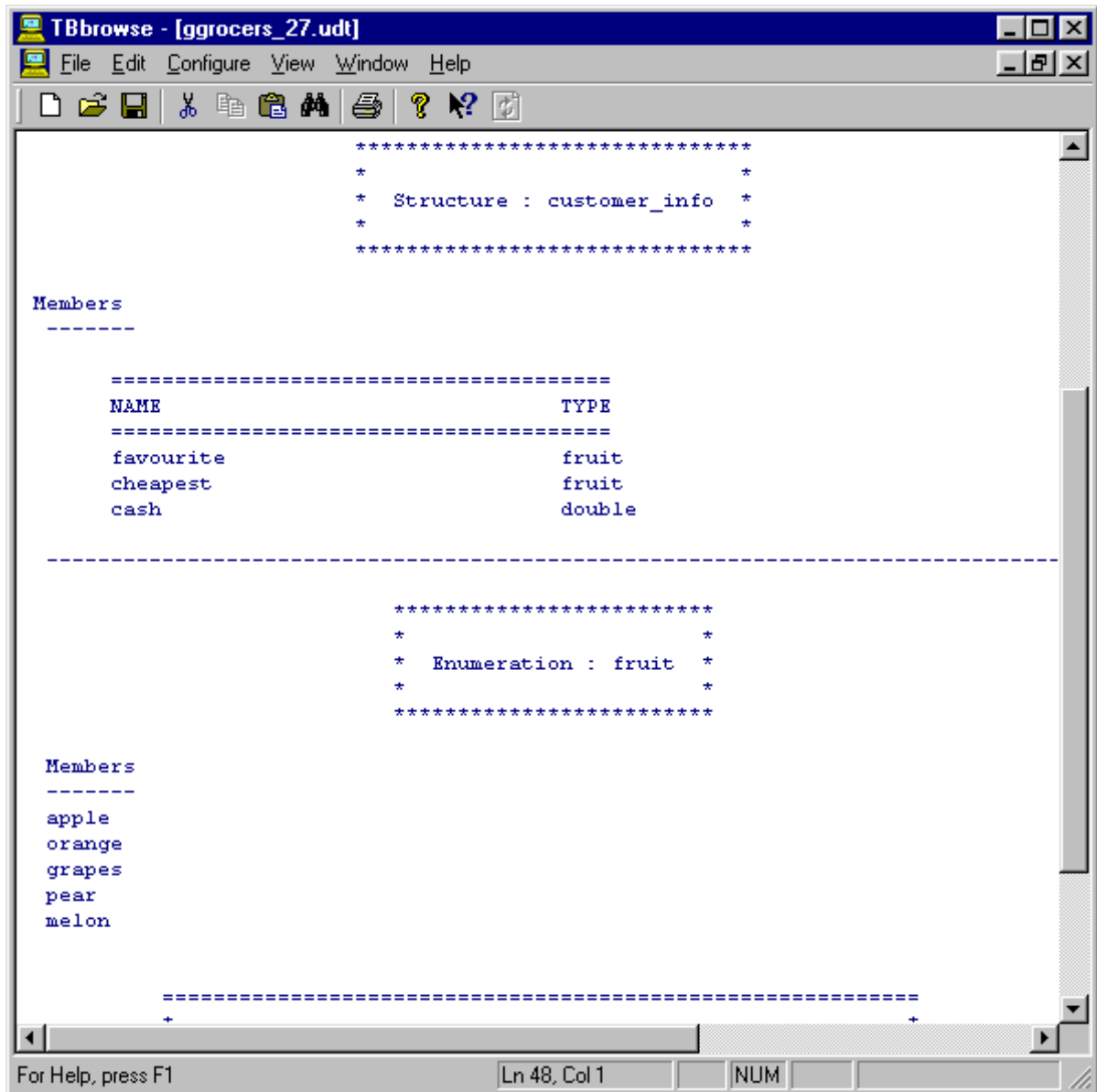
- メンバー変数
- メンバー関数
- 基本クラスリスト
- 継承先クラスリスト

```
*****  
* Class : CDrinksDispenser *  
*****  
  
Member Variables  
-----  
NAME TYPE ACCESS SPECIFIERS  
-----  
m_nMaxCans int private static  
m_nCoke CItem private  
m_nLemonade CItem private  
m_nFizzyOrange CItem private  
  
Member Functions  
-----  
NAME TYPE ACCESS SPECIFIERS  
-----  
CDrinksDispenser void public virtual  
WriteWelcome void public virtual  
GetMaxCans int public static  
  
Base Class List  
-----  
CDispenser  
  
Derived Class List  
-----  
No other classes are derived from this class.
```

ユーザ定義タイプレポート User Defined Types Report

ソースコード内の各ユーザ定義タイプ(以下)を記録します。

- enum Enumerations
- 構造体 Structures
- ユニオン Unions
- タイプ定義 Type Definitions
- クラス Classes



The screenshot shows a window titled "TBbrowse - [ggrocers_27.udt]" with a menu bar (File, Edit, Configure, View, Window, Help) and a toolbar. The main content area displays a report with the following text:

```
*****
*
* Structure : customer_info *
*
*****

Members
-----

=====
NAME                                TYPE
=====
favourite                           fruit
cheapest                             fruit
cash                                 double
-----

*****
*
* Enumeration : fruit *
*
*****

Members
-----
apple
orange
grapes
pear
melon

=====
+
+
=====
```

At the bottom of the window, there is a status bar with the text "For Help, press F1", a cursor position indicator "Ln 48, Col 1", and a "NUM" button.

概要 Overview

インストゥルメンテーションにより、動的テスト実行のソースコードのコードカバレッジやコントロールフローのモニタを実現しています。LDRA Testbed は、ソースファイルをコピーし自動でインストゥルメント(タグ付け)します。このタグ付けされたコードプローブ(差込まれるコード)は、以下の3つの動作をする単純なファンクションコールです。

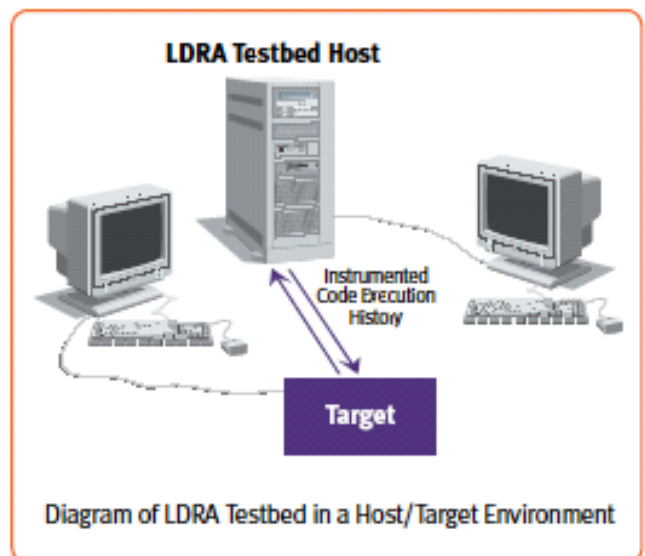
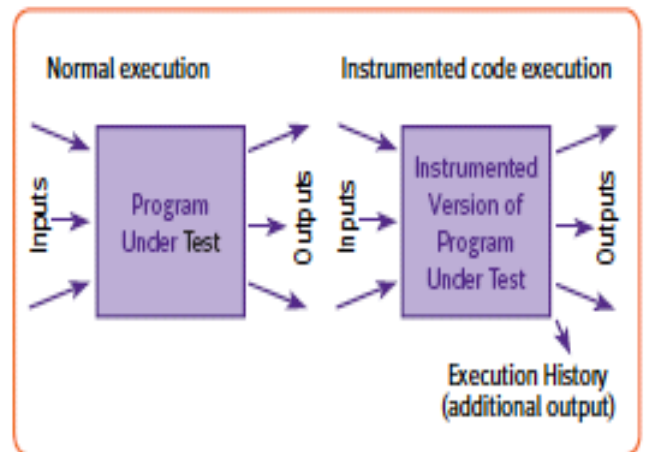
- 実行履歴ファイルの生成とオープン
- プログラム実行履歴情報(タグ情報)をこのファイルへ書き出す
- ファイルクローズ

このファイルは、LDRA Testbed の Dynamic Coverage Analysis でカバレッジ解析されます。また、組み込み開発では、配列等を用いてターゲット上のメモリにタグ情報を格納し、その結果をホストマシン上の Testbedに読み込ませて、カバレッジ解析を行えるようになっています。

Host/Target Testing

実ターゲットにおける動的テストを実現するにあたって、始めにLDRA Testbedのソースコードの静的解析が、ホストコンピュータ上で実行されます。その結果、タグ付けされたコードに対して、ターゲット固有のコンパイルなどを行って実際のターゲットMPU上で実行されます。動的カバレッジは、ターゲット上の実行履歴をホストマシン上へI/Oなどを通して戻され、解析されます。代表的なシナリオは、以下になります。

- メインフレーム Mainframe
- 組み込みシステム Embedded System
- シミュレータ Simulator



セマンティック解析 —Exact Semantic Analysis (part of TBsafe™)

LDRA Testbed は、実行時にLDRAの注釈(アサーション)をチェックすることが可能です。これは、変数値をboolean条件で比較します。アサーションは、ソースコードの様々な表記基準に基づいて特別なコメントとして記述されます。

LDRA Testbedは、ソースファイル内の特定の文字列を読み取ります。正確な文字列と一致ルールは、パラメータファイルに記述されます。各アサーションは、これらの特別な文字列で開始・終了しなければなりません。開始・終了を含む全アサーション行は、ソース内でコメントでなければなりません(コンパイル時コメントとして処理されます)。

アサーションの開始から終了までがアサーションとして扱われます。

表現は、有効なboolean表示であるか構文解析されます。

Ada、C、Pascal等の言語仕様に対応する明示的な記法にカスタマイズ可能です。

インストゥルメンテーションは、真理値(truth value)を判断する為に式の前後に加えられます。不正確なアサーションは、特定のアサーションを指す番号と共にfailure処理をコールします。

通常アサーションは、要求仕様書から指定されます。

コードに対してプリ・ポストコンディションを適用するように指定して使用されます。LDRA Testbedは、タグ付けソース(アサーション設定ONで)と、失敗に対する処理ルーチン(カスタマイズ可能な)を生成します。右図はExact Semantic Analysis によってアサーションを拡張・変換した例です。

Exact Semantic Analysisは、フォーマルメソッドと実用的なテストのコンビネーションで実現され、品質基準への準拠のために実行時にアサーションを検査する手法です。

```
with INTE_IO;
with TEXT_IO;

procedure test is
  k : integer;
begin
  k:=0;
  for i in 0 .. 40 loop
--assert
-- (i<40) and (k<300)
--
--end assert

  k:=k+i;
```

After Dynamic Conformance Analysis the Assertion (above) becomes expanded as demonstrated below:

```
with TEXT_IO;
with INTE_IO;
procedure assert_fail(n:integer) is
  fail : exception;
begin
  TEXT_IO.new_line;
  TEXT_IO.put("Assertion failure: ");
  INTE_IO.put(n,l); TEXT_IO.new_line;
  raise fail;
end;

with assert_fail;
with INTE_IO;
with TEXT_IO;
procedure test is
  k : integer;
begin
  k:=0;
  for i in 0 .. 40 loop
-- TESTBED assertion 1
  if not ( ( i < 40 ) and ( k < 300 ) ) then
    assert_fail(1);
  end if;
-- end TESTBED assertion
  k:=k+i;
```

上記アサーションは、forループカウンタ変数 $i < 40$ 、変数 $k < 300$ であることをチェックするものです。このアサーションは、タグ付け過程で展開(Expanded Assertion)され、この条件を満たさない場合、failure処理をコールします。

セマンティック解析 まとめ —Exact Semantic Analysis Summary

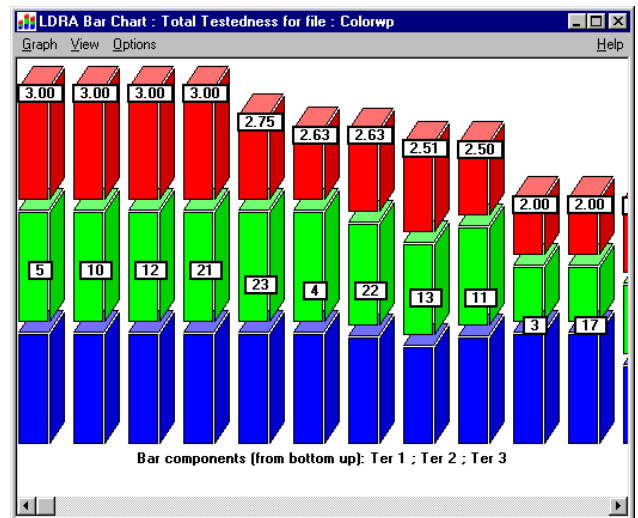
- ソースコードの正確な意味解析を実現
- ソースコードの診断
- ソースコード部分へプリ・ポストコンディション設定
- 入力指定の範囲を満足するかのチェック
- ループ、配列が境界内にあるかのチェック

動的解析 — Dynamic Coverage Analysis

LDRAツールの中で最も強力な機能の一つです。ソフトウェアの開発・保守時に使用することで、プログラムの堅牢性・信頼性に大きく貢献します。動的カバレッジ解析はテストデータの有効性の尺度であり、そのレベルは以下に分類されます。

- Statement Coverage (TER1)
- Branch/Decision Coverage (TER2)
- LCSAJ Coverage (TER3)
- Procedure/Function Call Coverage (P/FCall)
- Branch Condition Coverage (BCC)
- Branch Condition Combination Coverage (BCCC)
- Modified Condition Decision Coverage (MC/DC)

TERは、Test Effectiveness Ratio(テスト効率)のことで、コードカバレッジの達成率でテストデータの効果を表します。以下の式のように、実行コード領域に対すると実行結果の比率です。



TER1	=	$\frac{\text{number of statements exercised at least once}}{\text{total number of executable statements}}$
TER2	=	$\frac{\text{number of branches exercised at least once}}{\text{total number of branches}}$
TER3	=	$\frac{\text{number of LCSAJs exercised at least once}}{\text{total number of LCSAJs}}$
P/FCall	=	$\frac{\text{number of Procedure/Function calls exercised at least once}}{\text{total number of Procedure/Function calls}}$
BCC	=	$\frac{\text{number of Boolean operand values exercised at least once}}{\text{total number of Boolean operand values}}$
BCCC	=	$\frac{\text{number of Boolean operand value combs exercised at least once}}{\text{total number of Boolean operand value combinations}}$
MC/DC	=	$\frac{\text{no of Bool. operand vals independently affect dec. outcome}}{\text{total number of Boolean operands}}$

分母(実行見込み)は静的解析時に、分子はダイナミック解析時のタグ付けされたコードの実行結果から得られます。通常、カバレッジ率は一連のテストデータによるプログラム実行により高められます。

LDRA Testbed コードカバレッジ解析機能 – LDRA Testbed Code Coverage Capabilities

LDRA Testbed のカバレッジ尺度は、ソースコードの構造に合わせた各レベルで解析されます。

ステートメントカバレッジ 100% – Statement Coverage (TER1)

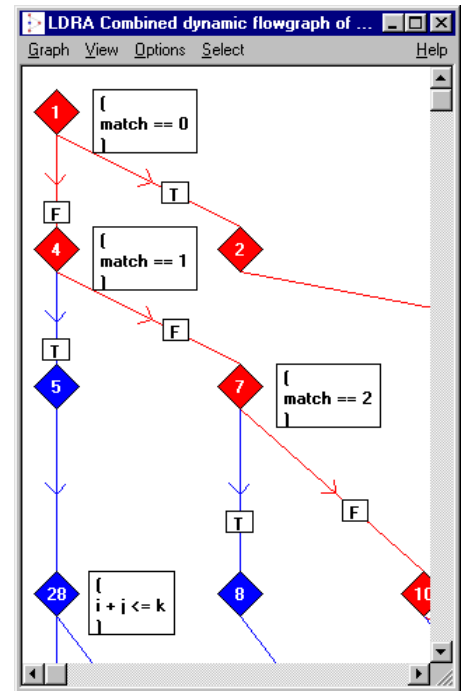
- コード内の全てのステートメントが実行される
- 全ての関数コールが呼び出される

ブランチ/デシジョンカバレッジ 100% – Branch/Decision Coverage (TER2)

- コード内の全てのステートメントが実行される
- 全ての関数コールが呼び出される
- 全ての分岐とリンクするステートメントが実行される

LCSAJカバレッジ 100% – LCSAJ Coverage (TER3)

- コード内の全てのステートメントが実行される
- 全ての関数コールが呼び出される
- 全ての分岐とリンクするステートメントが実行される
- 全ての関数リターンが呼び出される
- ゼロ回実行、シングル、ダブル、トリプル、それ以上の全てのループ実行がマルチループ内で実行される
- 全てのネスト、及びシーケンシャルループが実行される
- 全ての関数終了が全ての起こりうる終了パスで呼び出される
- 全てのブランチコンディションの組合せが実行される



TBbrowse - [imagedit.ms3.htm]

File Edit View Window Help

Combined Coverage Results

Procedure	Statement	Branch	LCSAJ
DrawSunkenRect	+100	+100	+100
DrawMarginBorder	+100	0	0
ViewSetPixel	0	0	0
ViewChar	0	0	0
ViewWndProc	+39	+29	+26
ViewReset	0	0	0
ViewUpdate	+100	0	0
ViewShow	0	0	0
ViewCreate	+41	+33	+30
ToolboxSelectTool	+71	+15	+3
ToolboxDrawBitmap	+100	+100	+100
ToolBtnWndProc	+75	+67	+75
ToolboxWndProc	+21	+13	+13
ToolboxUpdate	+59	+40	+44
ToolboxShow	+79	+50	+25
ToolboxCreate	+79	+71	+73
SaveIconCursorFile	0	0	0
IsValidDIB	0	0	0

関数/関数コールカバレッジ 100% –Procedure/Function Call Coverage (P/FCALL)

- 全関数/関数コールが実行される
- 全関数/関数コールリターンが実行される

ブランチコンディションカバレッジ 100% –Branch Condition Coverage (BCC)

- デジジョンコンディション(判定条件)内の各ブーリアンオペランドのTRUEとFALSEが実行される

ブランチコンディションコンビネーションカバレッジ 100% –Branch Condition Combination Coverage (BCCC)

- 各デジジョンコンディション(判定条件)内の各組のブーリアンオペランド値のユニークな組合せが実行される

モディファイドコンディション/デジジョンカバレッジ 100% –Modified Condition/Decision Coverage (MC/DC)

- 各デジジョンコンディション(判定条件)内の各ブーリアンオペランド値が実行される

コードカバレッジ測定 –Measuring Code Coverage

全てのコードが実行されたことを裏付けることがカバレッジ解析の用途です。テスト入力で一度も実行されないコード領域には、エラーが存在していたとしても検出され得ません。

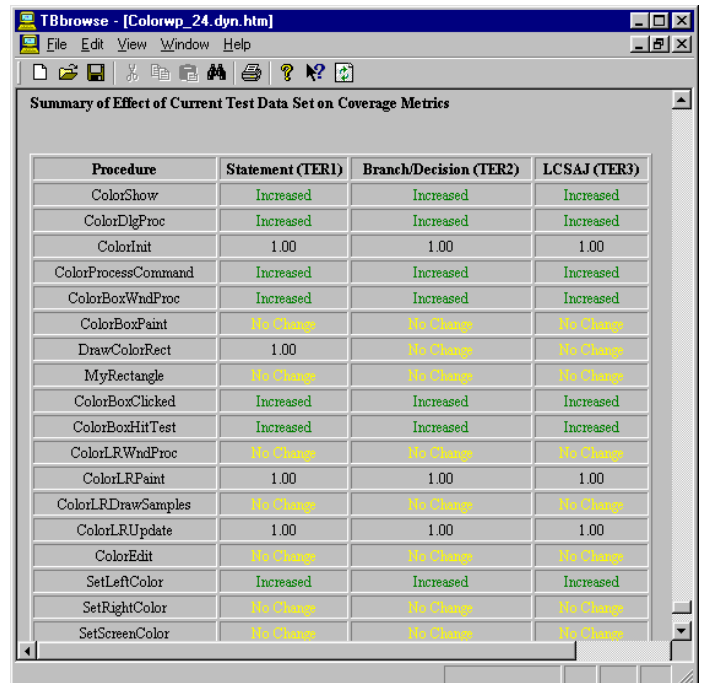
ステートメントカバレッジ(TER1)とブランチカバレッジ(TER2)は、相当な努力なくしてもある程度の達成度は得られます。

(実行不可能なブランチやコードも検出されるでしょう) LCSAJカバレッジ(TER3)を最大にするには相当なテスト計画を要します。

LCSAJで一貫性が到達できれば、多くの検知されていないエラーは十分に削減されるでしょう。LCSAJカバレッジを最大にするのは極めて詳細なテストが必要です。とりわけループ構造内のエラー検出に効果的でしょう。全ステートメントのカバーにはループがカバーされる必要は無く、直線的なパスの実行だけが必要とされます。

全ブランチのテストはループが一度は実行された事になりますが、全LCSAJをテストするには各ループが少なくとも2回カバーされることが必要です。

信頼性の高いコードが必要であれば、LCSAJカバレッジレベルを最大にすることをお勧めします。



The screenshot shows a window titled "TBrowse - [Colorwp_24.dyn.htm]" with a menu bar (File, Edit, View, Window, Help) and a toolbar. Below the toolbar is a section titled "Summary of Effect of Current Test Data Set on Coverage Metrics". This section contains a table with four columns: Procedure, Statement (TER1), Branch/Decision (TER2), and LCSAJ (TER3). The table lists various procedures and their corresponding coverage status.

Procedure	Statement (TER1)	Branch/Decision (TER2)	LCSAJ (TER3)
ColorShow	Increased	Increased	Increased
ColorDlgProc	Increased	Increased	Increased
ColorInit	1.00	1.00	1.00
ColorProcessCommand	Increased	Increased	Increased
ColorBoxWndProc	Increased	Increased	Increased
ColorBoxPaint	No Change	No Change	No Change
DrawColorRect	1.00	No Change	No Change
MyRectangle	No Change	No Change	No Change
ColorBoxClicked	Increased	Increased	Increased
ColorBoxHitTest	Increased	Increased	Increased
ColorLRWndProc	No Change	No Change	No Change
ColorLRPaint	1.00	1.00	1.00
ColorLRDrawSamples	No Change	No Change	No Change
ColorLRUpdate	1.00	1.00	1.00
ColorEdit	No Change	No Change	No Change
SetLeftColor	Increased	Increased	Increased
SetRightColor	No Change	No Change	No Change
SetScreenColor	No Change	No Change	No Change

カバレッジレポート — Coverage Reporting

以下のフォーマットでレポートされます。

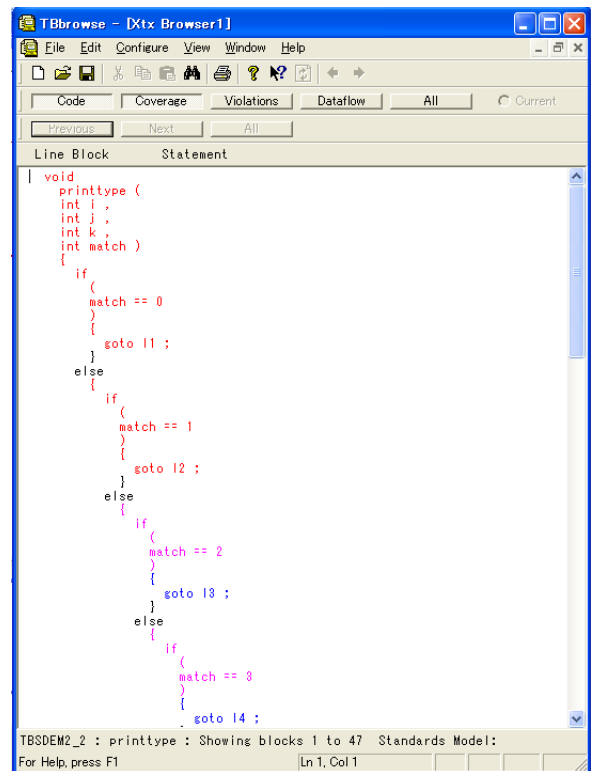
- HTMLorASCII 形式(注釈付きソースコードリスト)
- ダイナミックコールグラフ
- ダイナミックフローグラフ
- バーチャート(棒グラフ)
- フローグラフのコードブラウザ内の注釈付きソースコード

カバレッジ測定のPass/Fail判断基準はTestbed内で設定できレポート、グラフィカル表示から確認できます。

MCDC図

Full Truth Table For Decision						
=====						
EXPECTED EXECUTED BY RUNS...						
INDEX	C1	C2	C3	OUTCOME	PREVIOUS	CURRENT COMBINED
=====						
1	f(T, T, T)	=	T	NO	YES	YES
2	f(T, T, F)	=	T	NO	YES	YES
3	f(T, F, T)	=	T	NO	NO	NO
4	f(T, F, F)	=	F	NO	NO	NO
5	f(F, T, T)	=	F	NO	YES	YES
6	f(F, T, F)	=	F	NO	YES	YES
7	f(F, F, T)	=	F	NO	YES	YES
8	f(F, F, F)	=	F	NO	YES	YES

コードにおけるカバレッジ結果の色分け表示



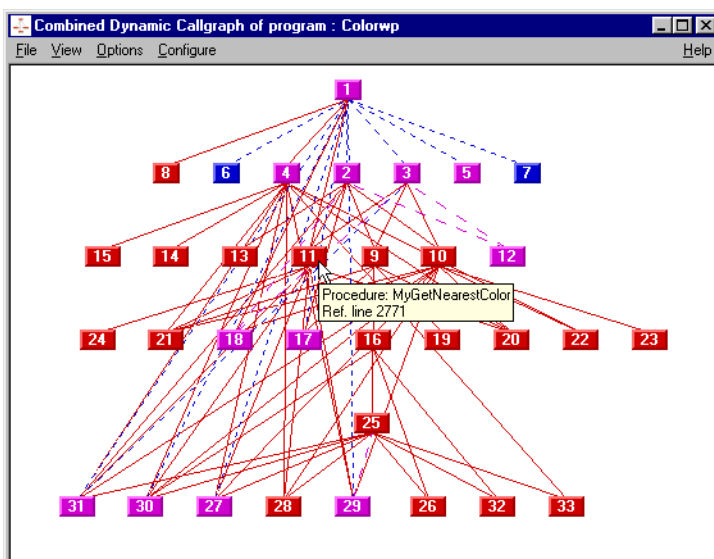
動的解析 まとめ — Dynamic Coverage Analysis Summary

カバレッジ尺度を最大にすることで、多くのコントロールフローパスがテストされたことを確かになります。

結果として、極めて多種多様の不具合が発見されるでしょう。

これらの不具合に対処する事で素晴らしい信頼性を得ることが出来るようになり、それは変更に対してもゆるぎない物になるでしょう。

ダイナミック解析は各ソースファイル、全システムに対応しています。



関数コールグラフによる カバレッジ結果の色分け表示

ダイナミックデータフローカバレッジ –Dynamic Data Flow Coverage

ダイナミックデータフローカバレッジは、変数のクロスリファレンス(ソースファイルやシステム内の何処で使用されているかや、そのタイプなど)リストを表示します。そしてこのリスト上の各変数に対し、最新のカバレッジ情報を統合して表示します。

ダイナミックデータフローカバレッジは、単一ファイル、システム内、グループ内で利用することができます。フィルターインターフェイスを用いて、特定基準としてある変数のみに的を絞る事も可能です。

```

*****
*****                               *****
*****  LDRA Testbed (R) Dynamic Data Flow Coverage Report *****
*****                               *****
*****                               Set : ddfset *****
*****                               *****
*****                               *****
*****                               *****
*****                               *****
*****                               *****

Dynamic Data Flow Table
-----
=====
VARIABLE      ALIAS      FILE      PROCEDURE  TYPE  ATTRIB.
=====
k
              ddf_test1  main      L        88
              L        R        92 ( 94)
              L        D        90
              ext_z    ddf_test2  ext_procl P        19
              P        R        ( 30)
              z        ddf_test1  procl P    52
              P        R        71 *****
              77
-----
j
              ddf_test1  main      L        87
              L        R        92  94
              L        D        90
              ext_y    ddf_test2  ext_procl P        18
              P        R        26
              y        ddf_test1  procl    P        51
              P        R        ( 60)
-----
i
              ddf_test1  main      L        86
              L        R        92  94  96
              L        D        90
              p3       ddf_test1  proc_p_call P    43
              P        O        45
              P        R        45
              ext_x    ddf_test2  ext_procl P    17
              P        R        23
              x        ddf_test1  procl    P    50
              P        R        56  67
=====

```

データセット解析 —Data Set Analysis

プログラムを変更したことをどのように再検証できるでしょう。明白なのは、既存の全テストデータに合わせて追加のテストで変更が正しく行われているかを検証することです。異なったバージョンのプログラムからのアウトプットを比較し、(望ましくは)より良くなった事を確認します。しかしながら、これには相当な時間やリソースが必要です。このようなリグレッション解析に対する簡単な答えはありませんが、LDRA Testbed により促進させることができるでしょう。

カバレッジ解析は、各テスト実行から累積されているので、データセット解析を使用してどのラインがどのテストで実行されたかを追跡できます。ソースコードが変更された場合、再実行が必要なコードを実行するためのテストデータセットが特定され、それを実行するのみです。

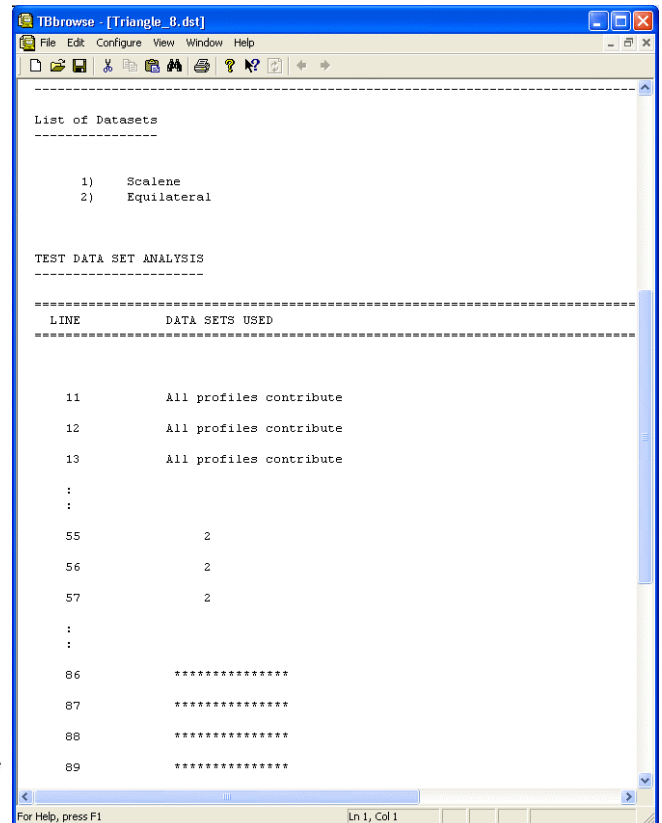
データセット解析は、データセットを示すテキスト(ダイナミック解析時にユーザから入力されたインプット)を利用します。このテキストはダイナミック解析表示の冒頭に表示されます。

特定のコード行が修正、拡張、メンテナンス等で変更された場合、どのテストを再実行すれば良いかを知る事は有益です。データセット解析は各実行行と、それを実行するテストデータセットの詳細を表示します。実行されなかった行には、アスタリスクが表示されます。ユーザは、どの特定のテストデータセットがコード変更により影響されたかを知った上で、リグレッションテストを構成する事が出来ます(グローバル変数は変更されないという条件で)。右上は関数対関数ベースの表示です。

図の中で、11行目は一番上で表示されているデータセット全てで実行されています。しかし55行目は、2番目のデータセットの実行のみです。86行目は実行されておらず、アスタリスクで表示されています。もし、56行目のコードが変更されるなら、それを実行する為に必要なテストは2番目のテストです。このようにして相当な時間とお金をセーブする事が出来ます。このテクニック(実行データのトレース)をデータスライシングと呼んでいます。

データセット解析 まとめ —Data Set Analysis Summary

データセット解析は2つの使用法が有ります。一つ目はソースコード上のどの行でどのデータセットが実行されたか。リグレッションテストに有効です。二つ目は、どの行が特定のテストデータセットで実行されたか。これは特定コード行を如何に実行するかのをひらめきを得るために有効なレポートとなるでしょう。



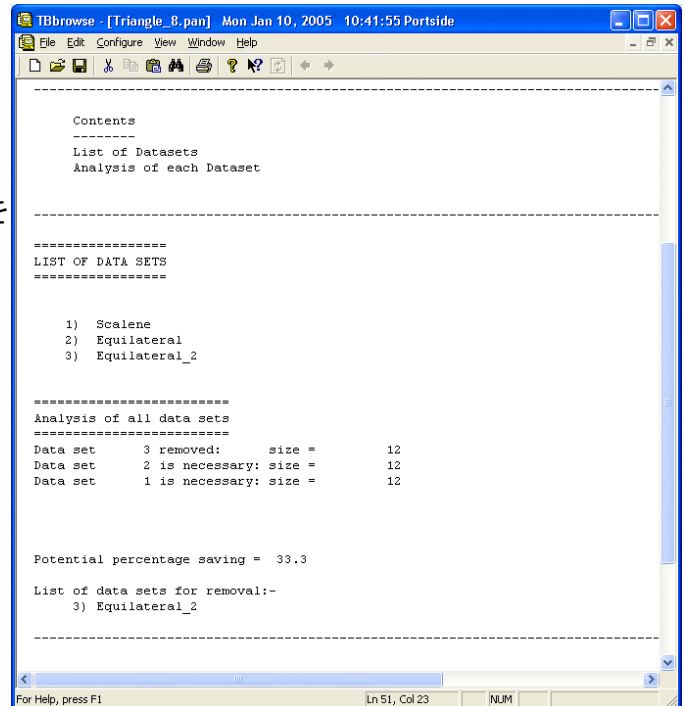
プロフィール解析 —Profile Analysis

ダイナミックカバレッジ解析率を高める為に、あらゆるテストデータセットを用いることでしょう。プロセス初期に使用されるテストデータセット(ステートメントやブランチ、LCSAJなどへ)は、頻繁にそれ以降のものと同様に重複します。重複しないテストデータセットでのみテストを実施していくことで、カバレッジの値はそのまま、各ステートメントやブランチ、LCSAJなどへの実行回数は削減できることでしょう。

テストデータセット X が重複した時に、これによるダイナミックカバレッジ結果をプロフィールから削除しても、カバレッジ結果は変わりません。この考えから、テストデータXはカバレッジ測定の観点では貢献しないといえます。

プロフィール解析の目的は、このような冗長なテストデータを検出しレポートする事です。これは各テストデータセットごとの個々のカバレッジプロフィールを解析して得られます。

いずれかのテストデータセットでカバーされた領域分を全カバレッジ結果の対照から外します。その結果、全カバレッジ結果が変わらなければ、そのテストデータは、全体的なカバレッジには貢献しないといえます。この解析を全テストデータセットに対して繰り返します。



The screenshot shows a window titled 'TBrowse - [Triangle_8.pan] Mon Jan 10, 2005 10:41:55 Portside'. The main content area displays the following text:

```
Contents
-----
List of Datasets
Analysis of each Dataset
-----

LIST OF DATA SETS
-----

1) Scalene
2) Equilateral
3) Equilateral_2

-----
Analysis of all data sets
-----
Data set      3 removed:      size =      12
Data set      2 is necessary: size =      12
Data set      1 is necessary: size =      12

Potential percentage saving = 33.3

List of data sets for removal:-
3) Equilateral_2
-----
```

恐らくカバレッジに貢献しないテストデータは多く見つかりますが、それらを全て同時に取り除く事はお勧めできません。幾つかは削除できるでしょう。2つ以上の削除できるテストデータがあった場合、LDRA Testbedによるコード実行結果の大きなトレース履歴を取得されることを推奨します。レポートされるテストデータを削除する事で、リグレッションテストのコストを削減する事が可能です。例えば、図中ではData Set 3 (Equilateral_2)が冗長としてハイライトされ削除を勧めています。

プロフィール解析 まとめ —Profile Analysis Summary

プロフィール解析の目的は、最小のテストセットでカバレッジを最大にすることです。これによりリグレッションテストの効率を高め、コストを削減します。

TBsafe™ - High Integrity Code Testing (DO-178B)

概要 — Overview

高信頼性が求められるソフトウェアのテストは、認証機関へ正当性を証明する為に相当なソースコード解析、カバレッジ解析が求められています。これらは、LDRA Testbed のTBsafe オプションによる追加のテストにて達成されえます。TBsafe は、LCSAJ カバレッジ機能と相まって使用される最も包括的な CAST ツールであり、最も厳格な基準にも対応しています。

TBsafe要約 — TBsafe Summary

TBsafeは、厳しい基準に対応するテスト解析を提供し、外部機関の認証を取得するために使用されます。

インフォメーションフロー解析 — Information Flow Analysis

これは、強力なドキュメント化ツールであり、依存関係がどうあるべきかをユーザが知っていることで、素晴らしい欠陥検出ツールにもなるでしょう。更には、メンテナンスでこれら依存関係が変更されることは不当な変更としてハイライトされなければなりません。詳しくは13ページのインフォメーションフロー解析を参考下さい。

セマンティック解析 — Exact Semantic Analysis

ダイナミックカバレッジ解析と合わせて使用する事で、アサーションは極めて広範なパスに対してチェックされます。またこれは診断生成にも使用されます。詳しくはセマンティック解析20ページを参照下さい。

MC/DC Coverage

DO178B認証で欠くことのできない特別なカバレッジ基準で、コンディションがテストされる事でエラーの存在を確認し、コードの信頼性を大きく高める為のものです。詳しくは23ページを参照下さい。

安全性への規約 — Safe Subsets

Safe Subsetsは、高信頼性が求められるアプリケーションに用いられるプログラミング言語の標準機能が危険性を秘めている為に考案されました。(例えばダイナミックメモリアロケーションによるメモリリークなど)
LDRA Testbedは、禁止された言語機能の使用をチェックします。詳しくは2ページの“Programming Standards Checking”を参照下さい。

TBrun™ - Test Harness Generator

概要 — Overview

TBrun は、テストドライバーとハーネス(ラッパーコード)を自動生成します。追加のコーディング、スクリプトは不要です。そのためコード単位のテスト実行を容易にかつ効果的に行えるようになります。また、これによるテストの入出力値と結果は保存され、リグレッションテストに対する解析の自動化に利用されます。ソースコード上の変更箇所を自動検出し、保存されているテストに変更が必要な部分はドキュメント化されますので、テストデータのメンテナンスが効率よく行えるようになります。

TBrun はホスト間、ホストターゲット間など、あらゆる環境でのテスト実行とカバレッジ解析をサポートします。要求、デザイン、テスト仕様などを参考にしながら、GUI、コマンド等を駆使してユニットテストを素早く生成できます。事例として、TBrun により、テストが76%まで削減できたことが報告されています。

TBrun:

- Saves time
- Test scripting not required
- Additional coding not required
- Frees up highly qualified staff
- Increases test efficiency
- Is less error prone, highly repeatable
- Improves motivation to test

TBrun:

- 時間の節約
- テストスクリプトの作成が必要で無くなる
- 追加コーディングの必要が無くなる
- 高い能力をもつスタッフを解放する
- テスト効率の向上
- 間違いを起こしにくい、高い再現性
- テストへのモチベーションを向上

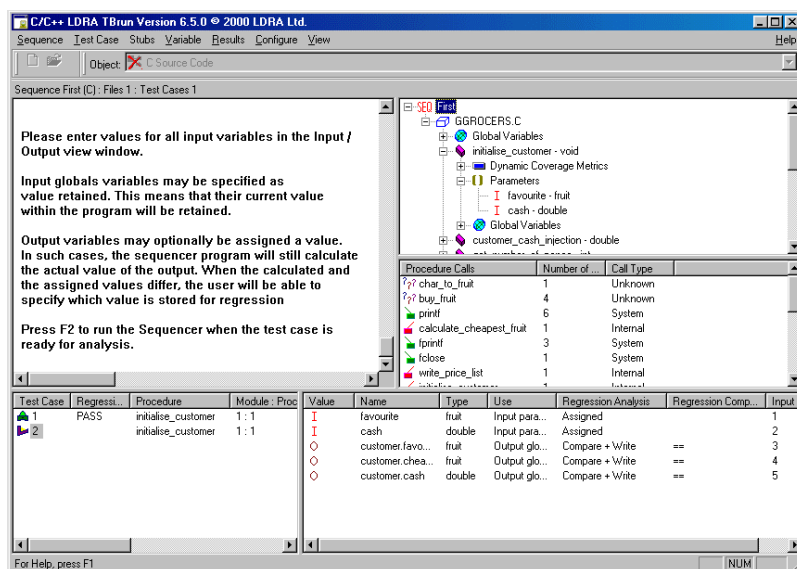
TBrun を使ったテスト生成 — Creating Tests with TBrun

TBrun は、LDRA Testbedによる解析から必要な情報を得ていますが、スタティックデータフロー解析が主要な手掛かりになっています。これは、ユニットインターフェースのパラメータ、グローバル変数(入出力)、戻り値、変数タイプと用法、関数コールなど、完全なソースコードのコントロールフロー解析データです。これらは自動的に収集されますので、ソースに対してのこれらに関する設定は必要ありません。

これらの解析結果から、テストデータを入力する為の、各ユニットごとの入出力インターフェイスが提供されます。これに対するテストデータは、テスト仕様書もしくは要求仕様書から得ることが出来るでしょう。

データの型などは、TBrun による解析結果から得られますので、レガシーコードや良くドキュメントされていないシステムのテストには欠くことのできない情報となるでしょう。

テストデータセットは、シーケンシャルに保持されますのでグローバルデータの伝播を考慮に入れることが出来ます。これにより、ユニットテストは従来手法に比べて、より実動作に近い(グローバルデータフローがインタラクティブにモニタされる)ものになります。



TBrunは、ソースコード解析とこれらのデータからテストデータをユニットで実行する為に必要なラッパーコードも自動生成します。

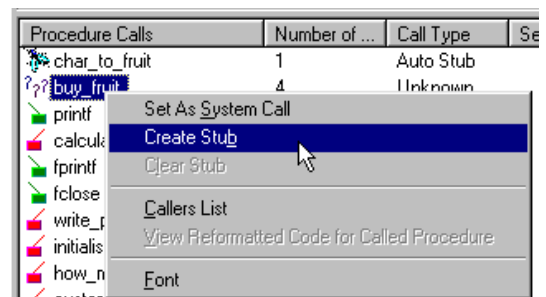
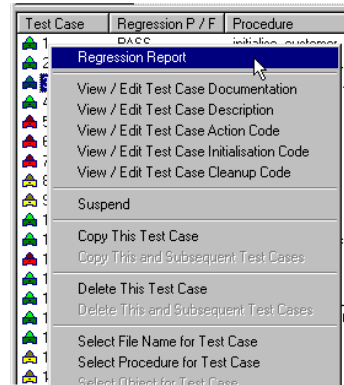
リグレッションテスト —Regression Testing

テストの実行に対して、トレーサビリティ情報を付加することで、自動でリグレッションテストの管理ができるようになります。テストとその結果はPVCSなどの、コードやテストのバージョンコントロールシステムへ簡単に保管することができます。そのため、ユニットへのリグレッションテストをビルドとテストサイクルの一部とする事ができるようになり、テストプロセスに対して柔軟に適応させることができます。

スタブ —Stubbing

ユニットテストでは、あるユニットから別のユニットへのコールはスタブ化される必要があります。記述されていないユニットや、テストで実行されないコードを含んだユニット、完全に分離した状態でテストされる全てのユニット、などです。

TBrun は、テスト内でのコールに対するスタブを自動生成します。生成されるスタブは、グローバル変数やポインタ等の解決、関数戻り値の設定など、テスト工程で柔軟に使用されます。



TBrun:

- Automatically generates test drivers and harnesses
- Runs tests on code units
- Automatic test vector generation
- Detects changes in source code
- Documents changes required in tests
- Performs regression tests
- Maintains test data and results
- Runs tests host/target
- Gathers code coverage metrics
- Advanced GUI
- Full command line interface

TBrun:

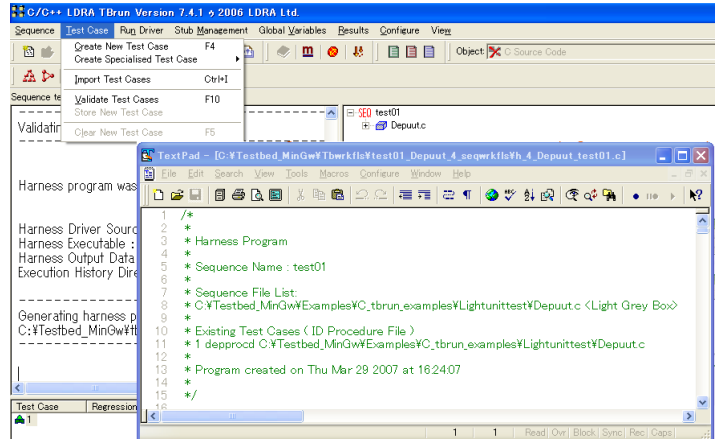
- テストドライバとハーネス(ラッパーコード)の自動生成
- コード単体でテストを実行—コーディングの必要なし
- ソースコードの変更を検出
- テストに必要な変更を文書化
- リグレッションテストの実行
- スタブの自動生成
- テストデータと結果の保持
- ホスト/ターゲット環境での実行
- コードカバレッジの尺度を収集
- 使いやすい GUI
- コマンドラインインタフェースにも完全対応

TBrun Features

TBrunは、ドライバプログラムの完全な自動生成を実現します。このドライバは、以下にあるようなあらゆる言語機能に対応しています。

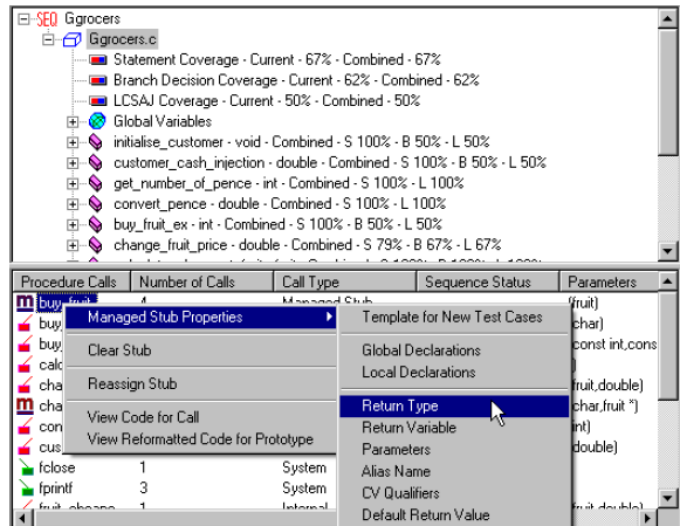
ドライバプログラムの自動生成 – Automatically Generated Driver Program

TBrun は、テスト対象ユニットの完全なインターフェースを得るために、洗練された制御フローとデータフローの解析技術を利用しています。これから得られる情報によりTBrun は自動的にテストドライバを生成することができますので、手作業によるスクリプト作成が不要になります。自動的に生成されるドライバへの制限はありません。ドライバは純粋な C/C++ や Ada83/95 のコードとして生成され、ホスト上あるいはターゲット環境で実行することができます。



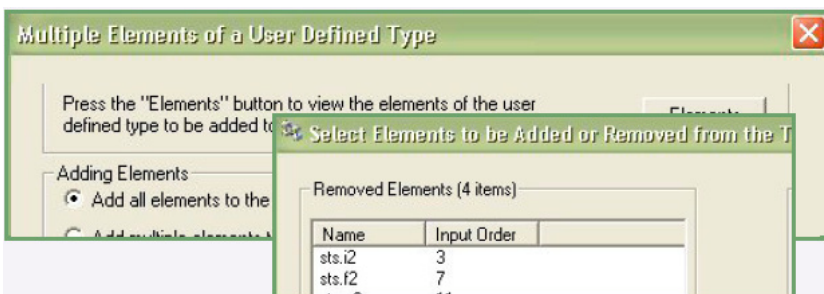
スタブ生成 – Stub Creation

スタブは、ワンクリックで生成されます。このスタブは、戻り値、値チェックなどを包括しているため、これらに対するコーディングの必要はありません。スタブは、関数、メソッド、コンストラクター、システムコール、パッケージなどに使用されます。



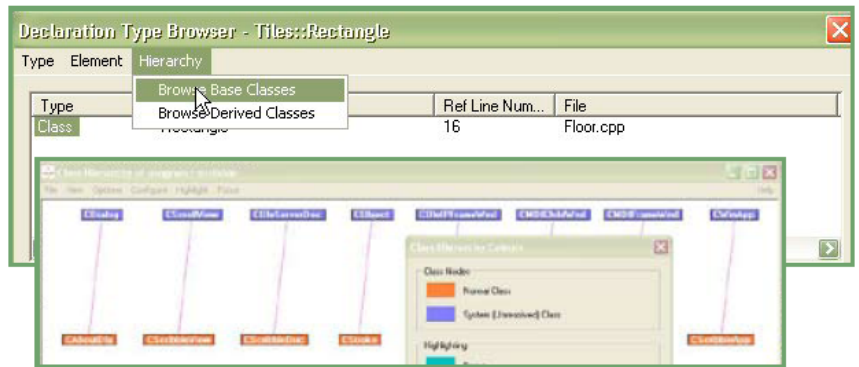
構造体／配列／共用体 – Structures / Arrays / Unions

TBrunには、テストに必要な構造体メンバを表示できる機能があります。テストデータとして値を指定して入力することができます。



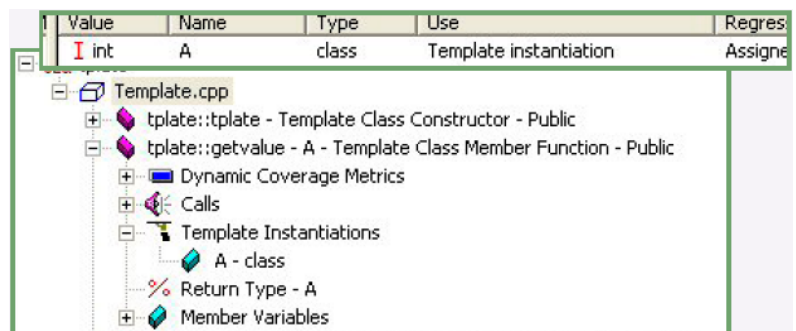
クラスの扱い — Class Handling

クラス階層は自動検出され、クラスに対するテストやインスタンスの生成は柔軟かつ効果的に行えるようになります。テストは、基本クラスに対して書かれ、派生クラスに自動的に適用されます。



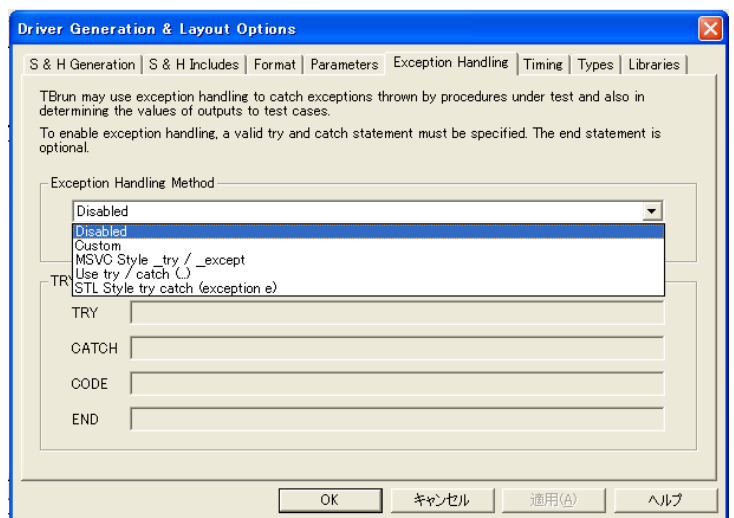
—テンプレートタイプの自動解決 — Automatic Resolution of Templated Types

TBrun は、テンプレートクラスの完全なテストとスタブ化を可能にします。この過程において、ユーザーは、テンプレートクラスのオブジェクトを生成するときに、テンプレート引数の型を最初に定義します。その後メソッドをテストするときに、テンプレート型は要求された型に自動的に変換されます。テンプレート引数から宣言時の型を得るアトリビュート、パラメータ及び戻り値は、テストケースの中で初期化され使用されます。メンバーのテンプレートも同様にテストされます。



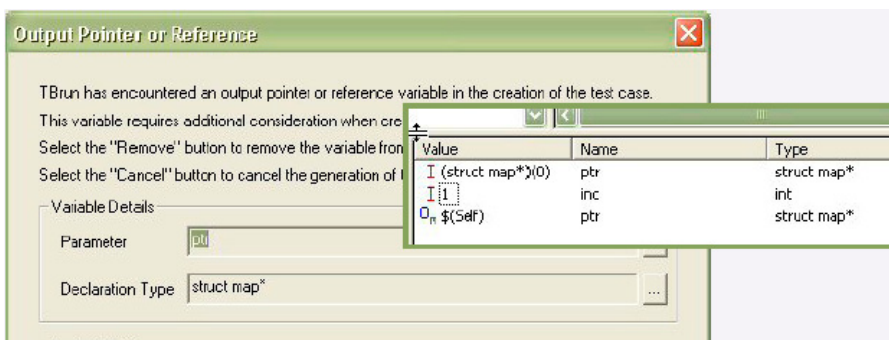
例外処理 — Exception Handling

テスト対象で発生した例外は自動的にキャッチされ処理されます。



ポインタの扱い — Pointer Handling

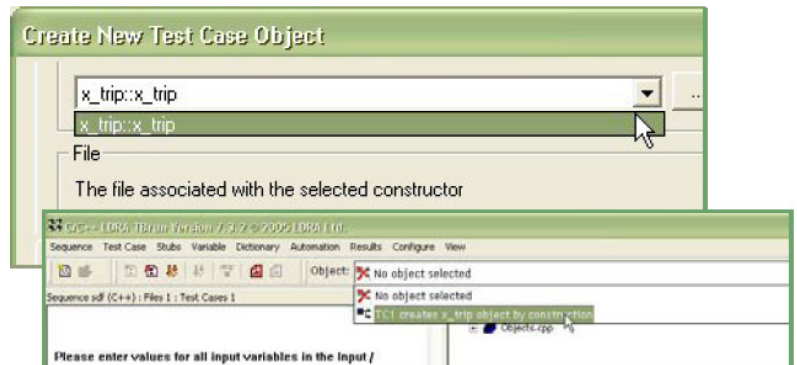
TBrun はポインタの使用を検出します。自動的に生成されたドライバープログラムではポインタ値も取り込まれ、ウィザードから入力も可能です。型の拡張とリンクリストの機能もあります。



自動生成とオブジェクトの再利用 — Automatic Creation & Object “Re-Use”

メソッドをテストするために必要になるC++のオブジェクトやクラスは、自動的に生成されます。

- ・オブジェクトを生成するためのコンストラクタのコール
- ・オブジェクトへの参照を返すメソッド
- ・オブジェクトを返すメソッド
- ・オブジェクトのアドレスを返すメソッド
- ・オブジェクトのアタッチメント—オブジェクトがグローバル変数にアタッチされている場合



その他の自動処理される言語の機能

- ・Abstractクラスのテスト
- ・テストにおける合成オブジェクトの自動生成
- ・Privateデータへのアクセス
- ・クラス階層内でのテストの再利用
- ・全階層に渡っての、メソッドとアトリビュートへのアクセス

大規模システムでのユニットテスト — Unit Testing Large Systems

TBrun は、ユニットテストが現実的でないと思われる大規模システムにも対応します。このようなシステムでは、未解決のグローバル変数に対して必要なテスト用の追加コーディングは相当な時間を要しコストのかかる作業でした。TBrun は、テスト対象ユニットのコンパイル・リンク時にどのグローバル変数が必要かを検出し、自動生成されるラッパーコードに宣言文を自動生成します。これによりテストにかかる労力を飛躍的に削減します。

TBrun まとめ — TBrun Summary

これは最も完全に自動化されたテストハーネス生成ツールです。ユニットを実行する為のラッパーコードは要りません。全て自動で生成されます。そのため、スクリプトやラッパーのアップデートなどで、リグレッションテストのたびに悩まされることが無くなります。

更なる利点は、テストデータセットが自動的にメンテナンスされる事です。ソースコードが変更されると、テストデータに変更が必要な場所が表示され、アップグレードが容易にできます。結果、マニュアルでのアップデートや追跡作業と比較して、多大な時間とコストの節約になります。

概要 - Overview

TBevolve は、テストプロセス内でコードの変更による影響を正確にモニターします。TBevolve はシステムのコピーを取り、コード変更による影響を受ける部分をハイライトします。アプリケーションコードへの変更ゆえ、ドミノ倒しの影響が及ぼされる部分の追跡とテスト作業を最小化します。

TBevolve は、従来厄介とされてきた、コード変更により影響を受ける部分の解析を自動化します。LDRA Testbedのコードカバレッジ機能とあわせることで、インパクトを受けた領域の再テストが十分なレベルであるかの評価も可能です。TBevolve は、ハザード解析(危険解析)追跡情報から、再テストが必要とされるハザード領域へリンクする事も可能です。

```

TDbrowse - [ezins.pdl]
-----
LDRA Testbed(R) Program Difference Analysis
-----
File : EZINS.COB
-----

Baseline date   : 19-APR-2000 12:00:26.32

Text Difference Overview
-----

Removed code areas:-
  6 reformatted lines in Paragraph FILE-OPEN
 14 reformatted lines in Paragraph D-KONST
  6 reformatted lines in Paragraph EZINS
  5 reformatted lines in Paragraph I2
  5 reformatted lines in Paragraph MAIN
  5 reformatted lines in Paragraph ENDS

Modified code areas:-
 548 ref lines changed to 1338 ref lines in Program L322EZIN
  7 ref lines changed to  9 ref lines in Paragraph EXP-ENDE
 19 ref lines changed to 24 ref lines in Paragraph LN2
 16 ref lines changed to 18 ref lines in Paragraph I1
  5 ref lines changed to  6 ref lines in Paragraph EZINS-ENDE
 14 ref lines changed to 16 ref lines in Paragraph ITER

New code areas:-
  9 reformatted lines in Section FILE-OPEN
 11 reformatted lines in Section FILE-CLOSE
 15 reformatted lines in Section PRINT-FKT
 14 reformatted lines in Section ABS
 99 reformatted lines in Section EISEC
 37 reformatted lines in Section EZINS
 30 reformatted lines in Section MAIN

Executable Code Difference Overview
-----

1 difference(s) containing 2 lines in Section L322EZIN
1 difference(s) containing 1 lines in Section FILE-OPEN
1 difference(s) containing 1 lines in Section FILE-CLOSE
1 difference(s) containing 11 lines in Section PRINT-FKT
1 difference(s) containing 1 lines in Section PRINT-POT
    
```

```

TDbrowse - [ezins.dcv]
-----
LDRA Testbed(R) Program Difference Coverage Report
-----
File : EZINS.COB
-----

-----
Ref  Reformatted Text          Previous Current
Line                                     Runs   Run   Combined
-----
Paragraph PRINT-LN-ENDE
488  WRITE R-HLP.                0      44    44
489                                     0      44    44
490% PRINT-LN-ENDE.             0      44    44
491% EXIT.                      0      44    44
492 *                            -       -     -
493%                             0      44    44

Section PRINT-LNP1
494% PRINT-LNP1 SECTION.        0      27    27
495% MOVE TDA TO TPI-TDA.        0      27    27
496% MOVE Y1 TO TPI-LX.          0      27    27
497% MOVE X1 TO TPI-LY.          0      27    27
498% MOVE SPACES TO R-HLP.       0      27    27
499% MOVE TPI TO R-HLP.          0      27    27
500% WRITE R-HLP.                0      27    27
501%                             0      27    27

Section READ-EIN
579% READ-EIN SECTION.          0      10    10
580% READ INP                    0      10    10
581% AT END                       0      10    10
582% GO TO READ-EIN-ENDE.        0       0     0 *LCS
583 ADD TDA , 1 GIVING TDA.      0      10    10
584 MOVE TDA TO E-TDA.           0      10    10

-----
Number of New Executable Lines      610    610    610
Number Executed                      0     389    389
Number not Executed                   610    221    221

Test Effectiveness Ratio
(New Line Coverage - rounded down)    0.00    0.63    0.63
-----
    
```

ツールのデモンストレーションや、
無償評価版を用意しています。

評価のお時間が取れない場合は、
サンプルコードをお預かりして弊社で
解析させていただきます。

ご興味いただける場合は、お手数ですが
右記までご依頼頂けると幸いです。

<http://www.ldra.co.uk/>

富士設備工業株式会社 電子機器事業部
担当: 浅野

TEL: 072-252-2128

E-Mail: info@fuji-setsu.co.jp

<http://www.fuji-setsu.co.jp>