Technical Description V9

日本語版

Produced by the LDRA Group of Companies.

Static Analysis

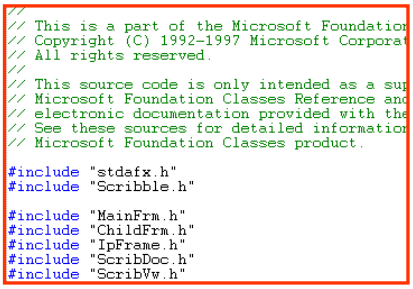
メイン静的解析 – Main Static Analysis

LDRA Testbed はすべての解析の土台として、単一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

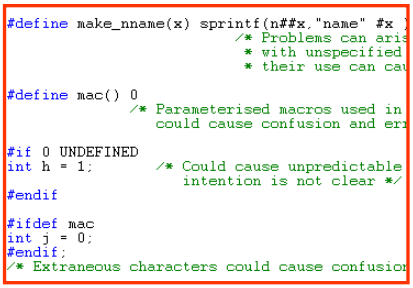
LDRA Testbed は独自の構文解析技術により、高度で柔軟性に優れた解析を提供しています。ツールによる制約が少ないのでテストに集中できます。LDRA Testbed で解析できるものは以下になります。

・ソースコード
・ソースコードとローカルヘッダーファイル
・コンパイラのヘッダーファイル
・外部ライブラリのソースファイル
・コンパイラのプリプロセッサーを通したファイル



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:単一関数、単一ソースファイル、関連しないソースファイル群
・統合/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



ヘッダーファイルのインクルード – File Inclusion

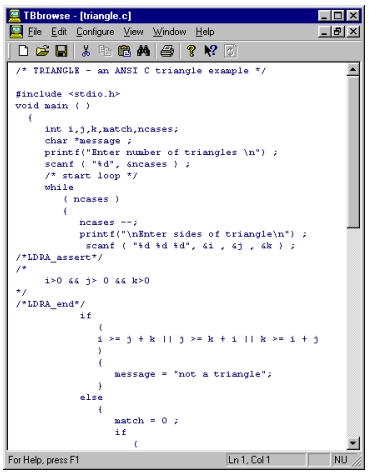
LDRA Testbed は、インクルードするヘッダーをデータファイルで指定できます。このため解析範囲の設定が非常に柔軟になっています。

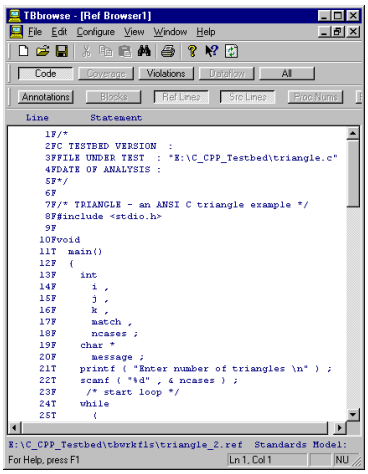
マクロ展開 – Macro Expansion

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すべ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。



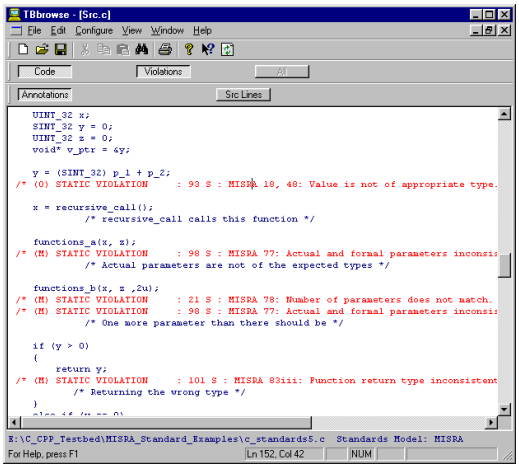


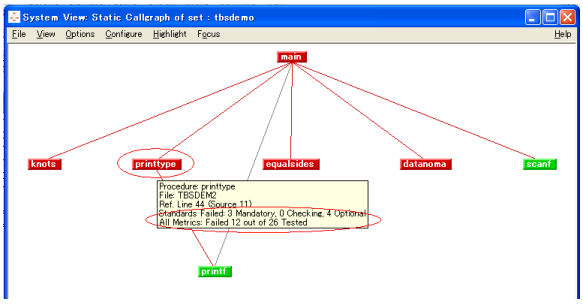
コーディングルールチェック – Programming Standards Checking

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すべての違反を
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。

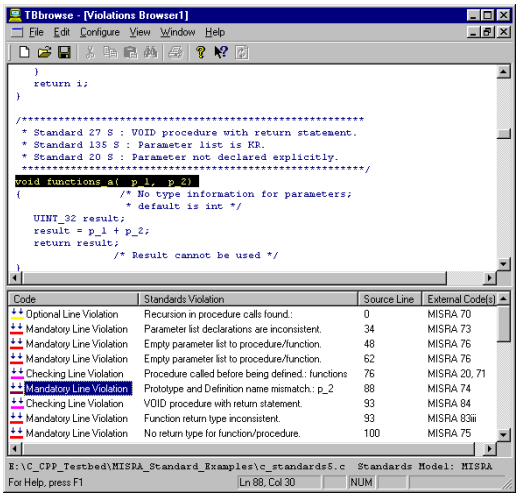




MISRA C:1998/2004/2012, MISRA C++:2008, MISRA AC Checking

MISRA 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の
3種すべての評価ができる唯一のツールです。

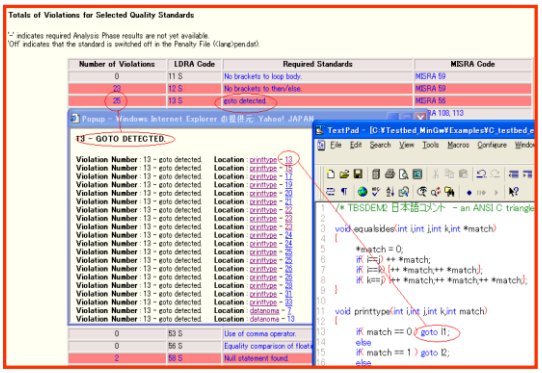
・コーディングルール
・複雑度の尺度
・コードカバレッジ



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。



DERA C (C/C++ only)

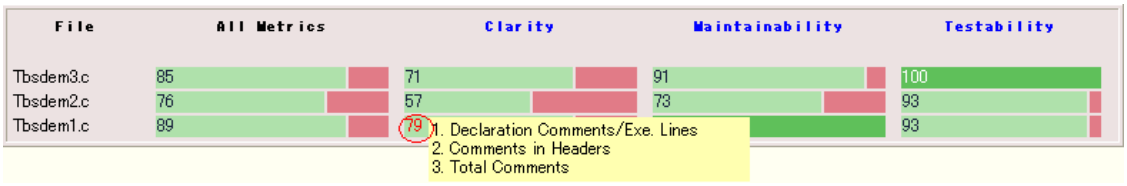
DERA C は、セーフティクリティカルシステムに対する MISRA C 標準の拡張版です。UK（英国） MOD（Military of Defence）で
LDRA の協力を元に制定されました。DERA C では、高度なソースコードデータフロー、インフォメーションフロー解析が求められ、LDRA が唯一対応出来ています。※DERA : Defence Evaluation and Research Agency

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、ソフトウエア製品の品質（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。

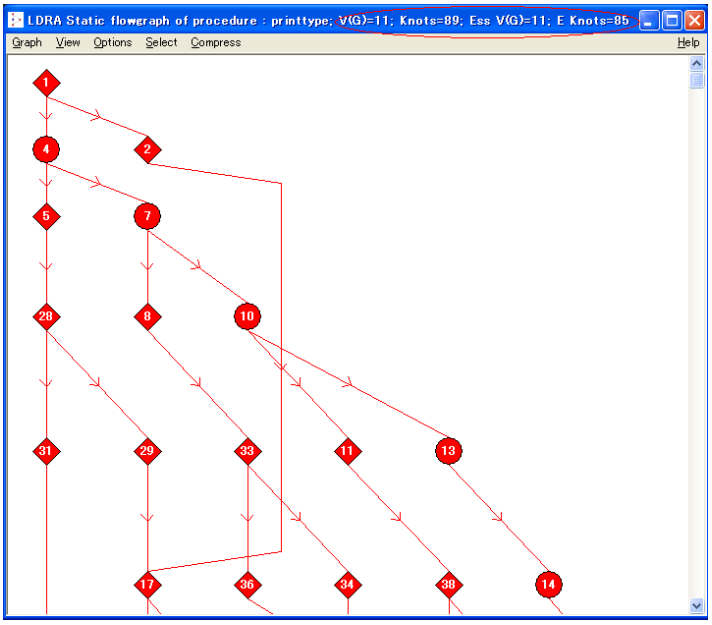


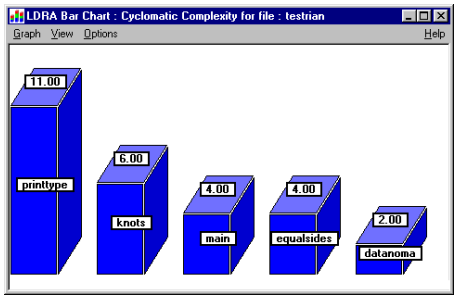
ノット解析 – Control Flow Knots

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) = グラフ中のリンク数－グラフ中のノード数＋2×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

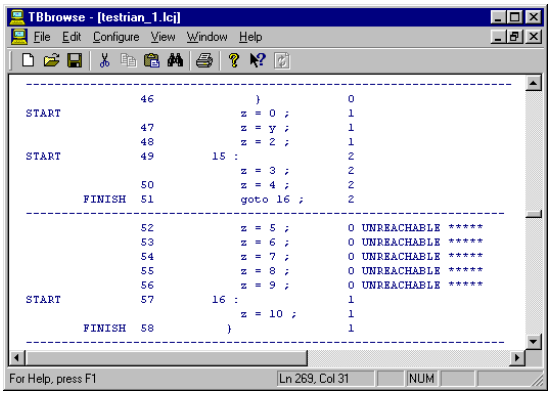
すべての実行行は、プログラムの開始から制御フローパスを通って到達可能であるべきでしょう。到達され得ないコードは、そのようなパスを持たないステートメントで構成されます。 LDRA Testbed は、これらの行を“Unreachable”としてマークし、抽出します。これらはプログラム実行に何の影響もないので、障害なく削除する事が出来ます。

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 – LCSAJ Density

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、変更によるすべ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。



コメント – Comments

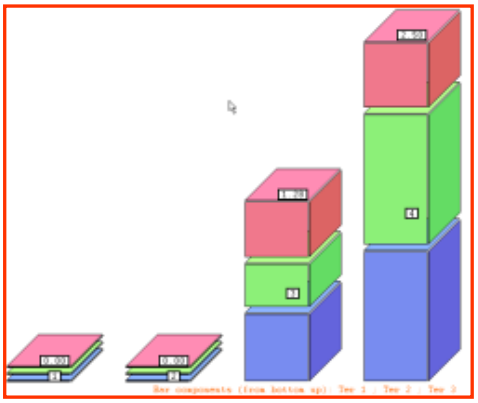
コメントによる可読性、メンテナンス性の評価。

・関数宣言前のコメント行数(関数のヘッダー)

・コメント内の全ブランク行数

・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 – Structuredness-Structured Programming Verification

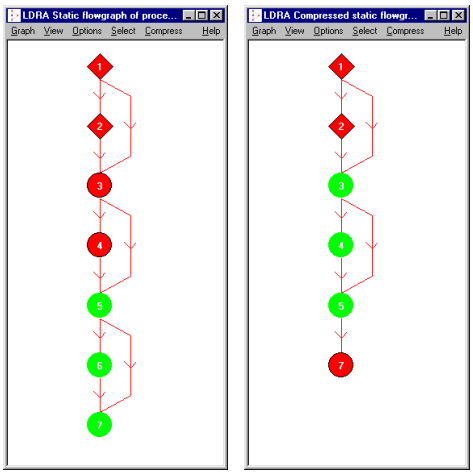
LDRA Testbed の構造化プログラミング検証(SPV)により、プログラムが適正に構造化されているかを判定します。LDRA Testbed は、テンプレートを用いて、対象プログラム内の制御文(例:if then else、do while、for など)を評価します。

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。

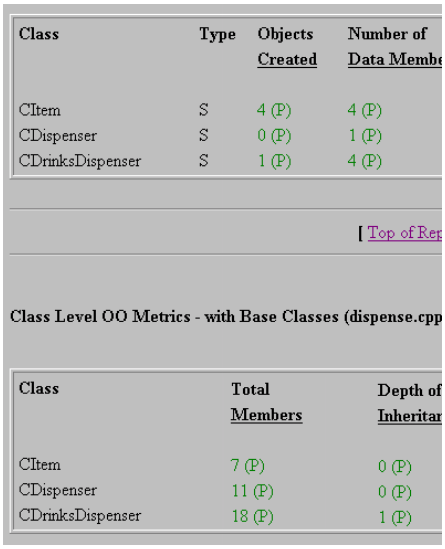
”Essential Cyclomatic Complexity”は、残差グラフ(残差=観測値－予測値)の式を適用して算出されます。Essential Knots も、構造化されていないことの尺度として得られます。適切に構造化されたプログラムは

“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 Oriented 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。



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

[Top of Rep

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

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

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

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

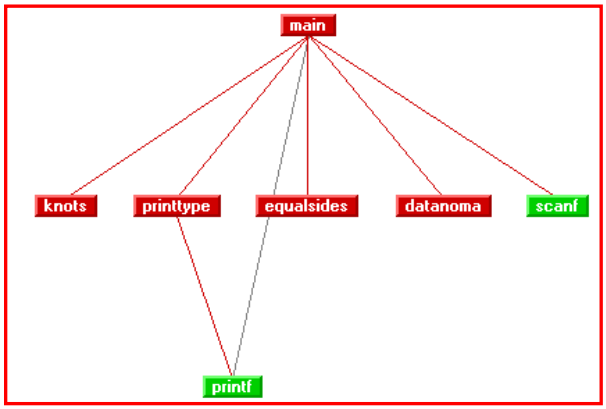
・メソッド内のクラス外オブジェクトの数　－Number of out of class objects in methods

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。

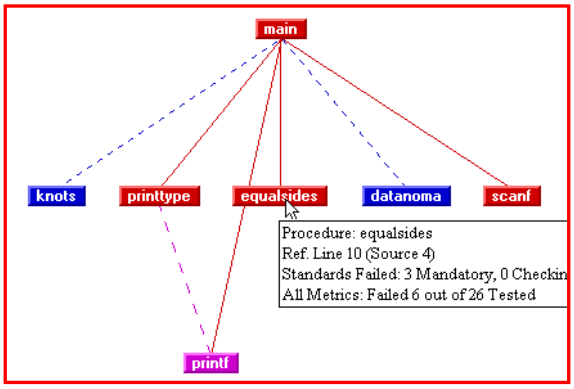
Fan Out は、他の関数へのコール数です。

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

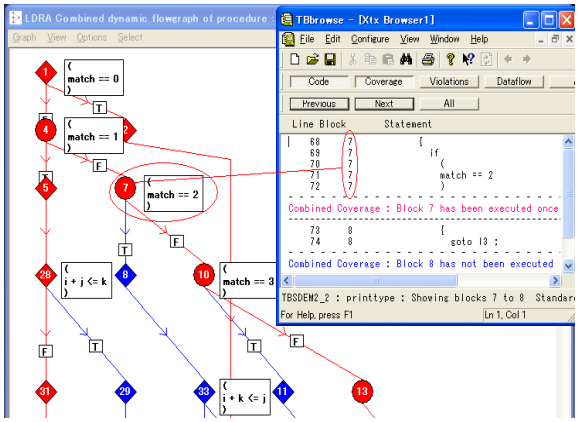
主要な静的解析と複雑度解析は統合され、グラフィカルに表示されます。



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 – Dynamic Callgraph



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、すべて実行で赤、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 – Dynamic Flowgraphs

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。

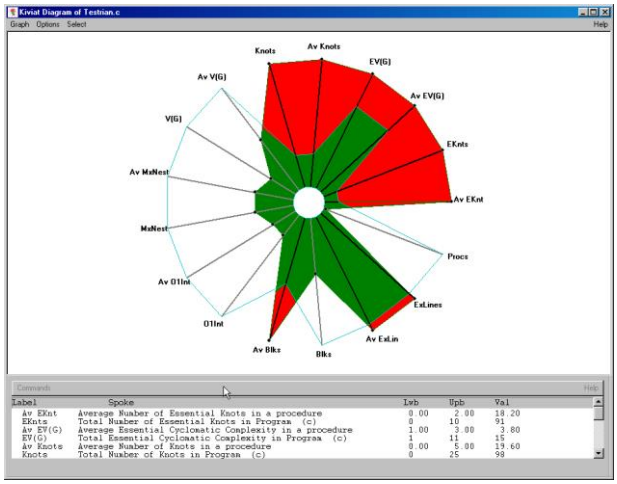
アクティブフローグラフ機能 – Active Flowgraph

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

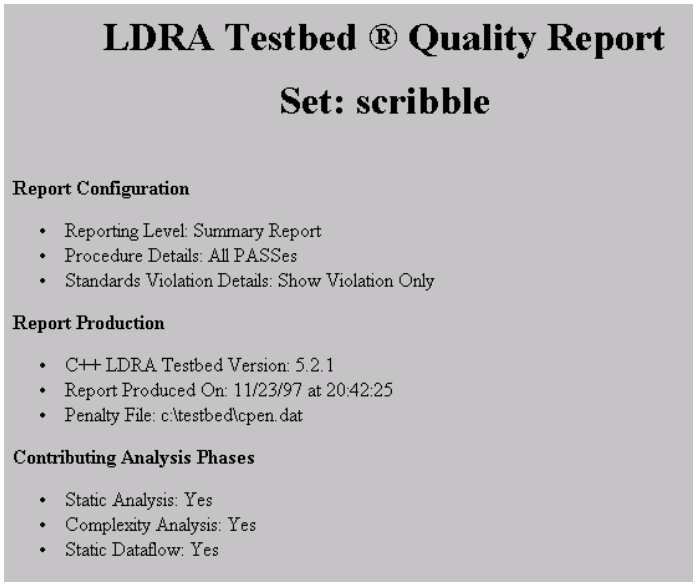


キビアット図は様々な尺度を基にグラフ化し、ソースコードが基準に順守しているかを表示します。もし、各尺度においてキビアット図が緑色なら合格です。赤色部分が有れば、追加のテスト、あるいはコーディング規約に適合されるような修正が必要であることを示します。

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

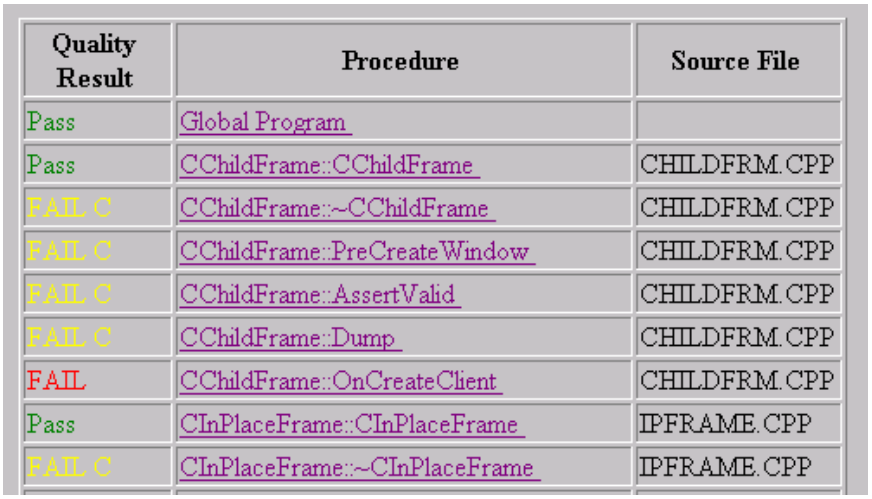
Contributing Analysis Phases

- 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。



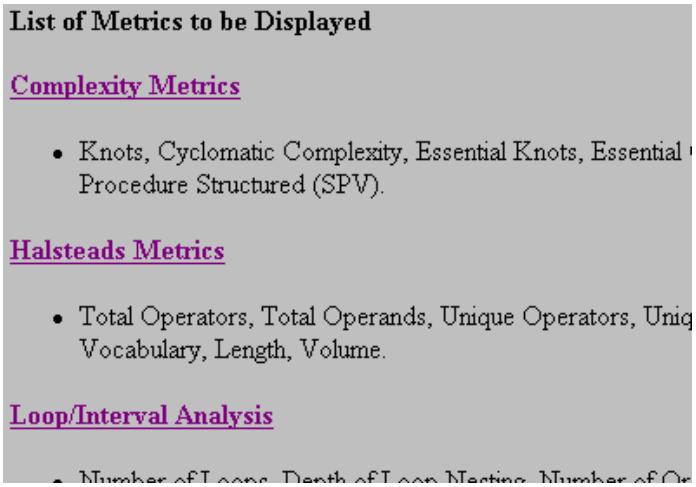
Quality Result	Procedure	Source File
Pass	Global Program	
Pass	CChildFrame::CChildFrame	CHILDFRM.CPP
FAIL C	CChildFrame::~CChildFrame	CHILDFRM.CPP
FAIL C	CChildFrame::PreCreateWindow	CHILDFRM.CPP
FAIL C	CChildFrame::AssertValid	CHILDFRM.CPP
FAIL C	CChildFrame::Dump	CHILDFRM.CPP
FAIL	CChildFrame::OnCreateClient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


List of Metrics to be Displayed

Complexity Metrics

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

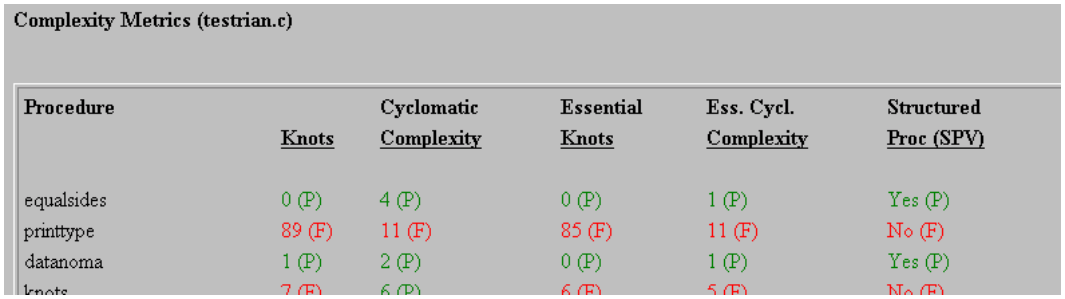
Halsteads Metrics

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

Loop/Interval Analysis

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

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)


Complexity Metrics (testrian.c)

Procedure	Knots	Cyclomatic Complexity	Essential Knots	Ess. Cycl. Complexity	Structured Proc (SPV)
equalsides	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

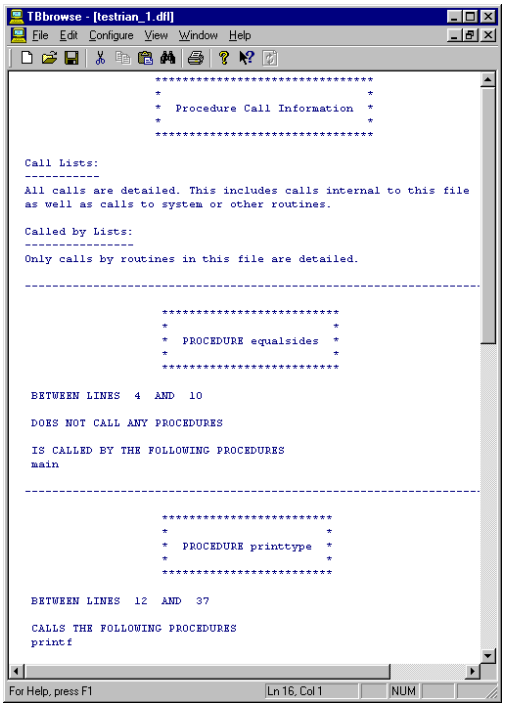
データフロー解析は、ソースファイルやシステムに対する、4 種の情報を生成します。

・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。再帰呼び出しもすべて解析され、どこで再帰が起こっているか詳細を示します。

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 一値の割り当てなど

・V が参照(Referenced)される 一値の使用など

・V が未定義(Undefined)になる一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)

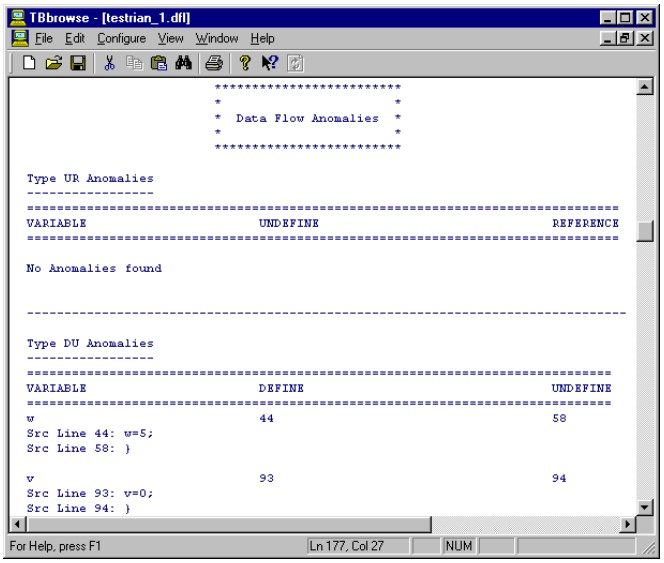
以下の例を考察してみます:

```
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、プログラム上のすべ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。各タイプ別にグループ化され、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。

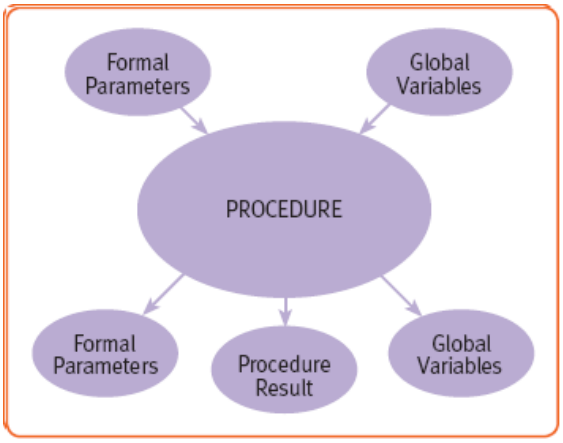


関数インターフェイス解析 – Procedure Interface Analysis

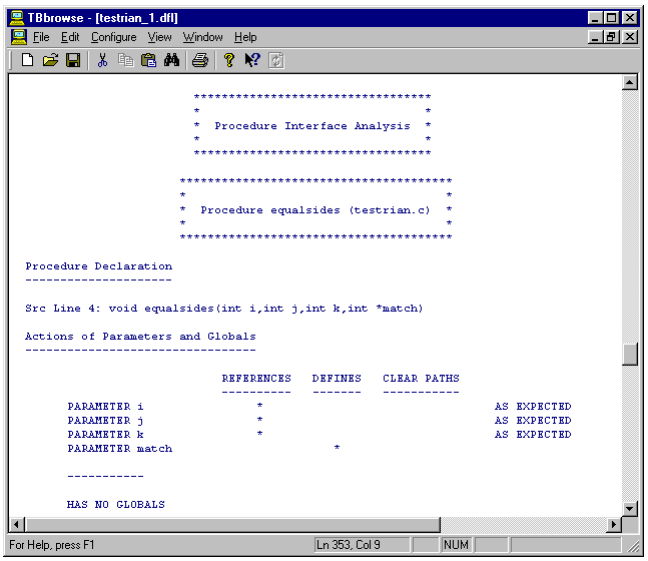
ソフトウェアの品質を左右する顕著な部分は、関数間の相互作用をコントロールし制限する関数インターフェイスです。LDRA Testbed は、全関数の引数、グローバル変数、関数ごとのローカル変数などすべてを解析します。

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。

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

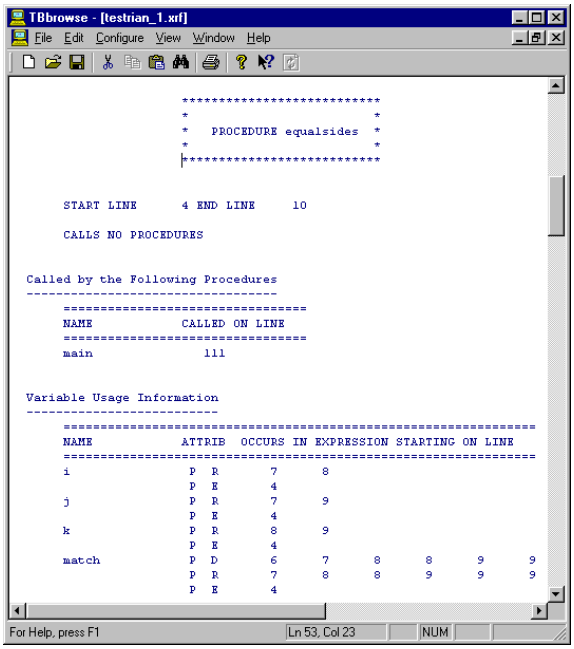
グローバルデータフロー解析は、各ソースファイルごと、全システムにまたがって行われます。

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、不具合を修正するための参考になります。
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すべ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すべ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。

記号　意味

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 – Information Flow Analysis (part of TBsafe™)

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。インプット変数のみです。 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 ー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 への割当が B に依存する場合です。

弱い依存 ー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 1 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 ー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 ー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 ー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)

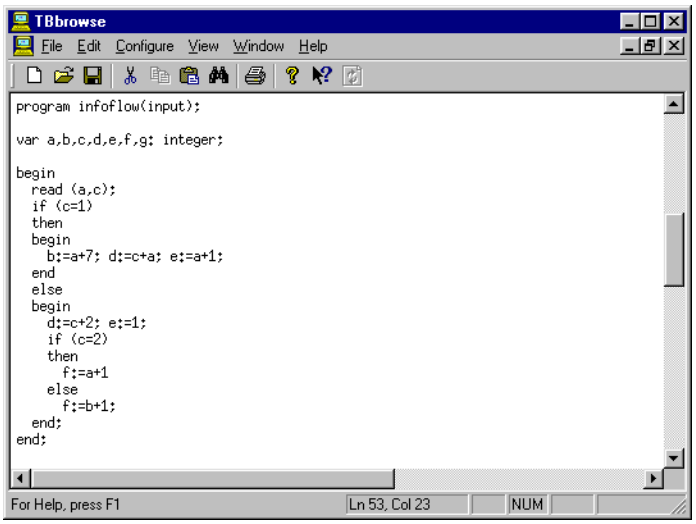
--LDRA_INFOFLOW DGSZ ( )

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

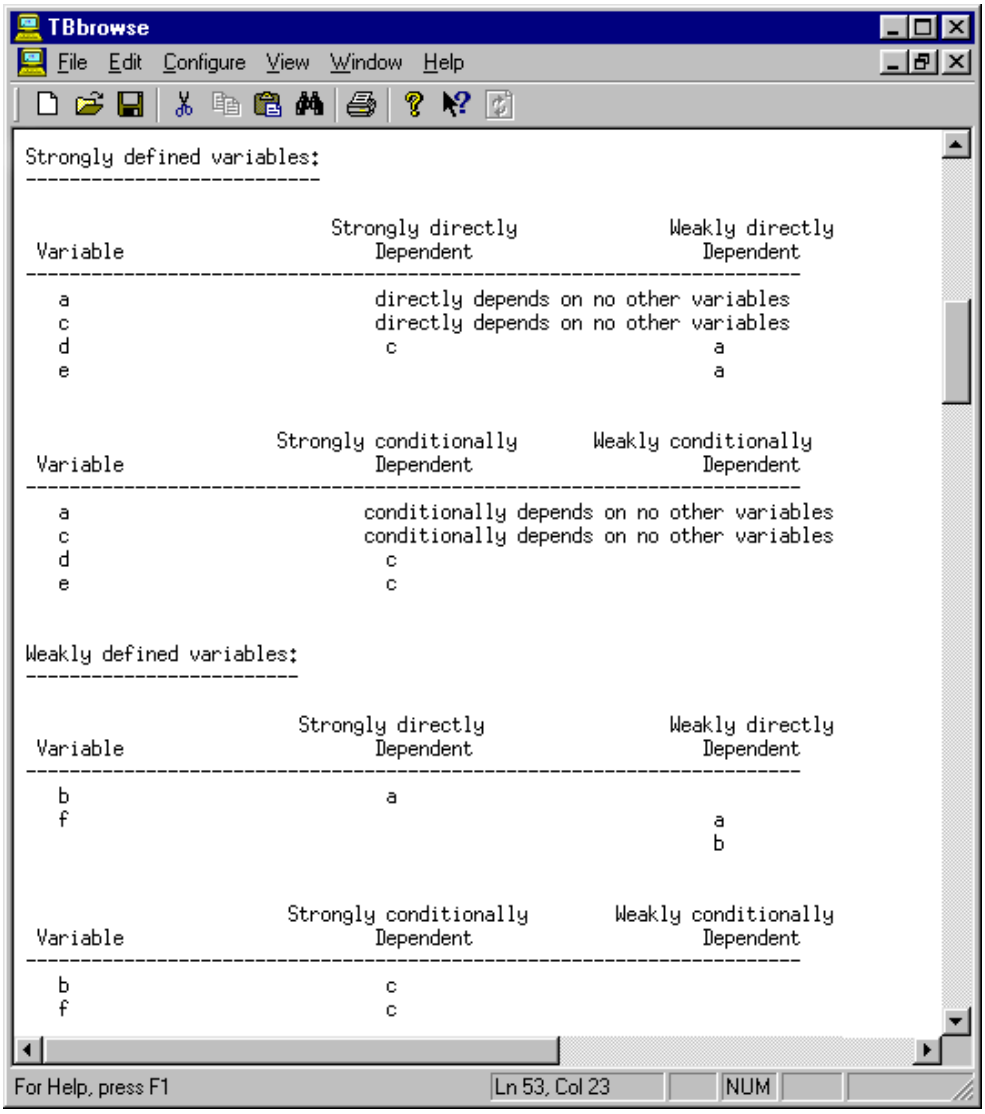
--LDRA_INFOFLOW MANY depends on (ONE,TWO,

--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



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）



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。

・構造体メンバー　Structure member

・ユニオンメンバー　Union member

・enum メンバー一覧　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）

・クラスメンバー　Class member

・ネームスペースメンバー　Name-space member

・テンプレートクラスメンバー　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)

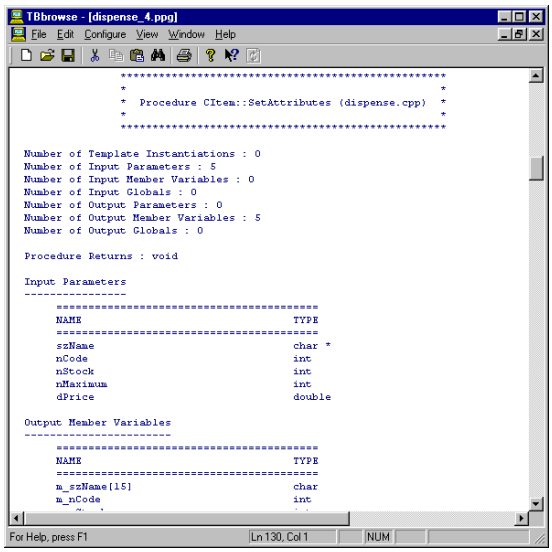
概要 – Overview

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。

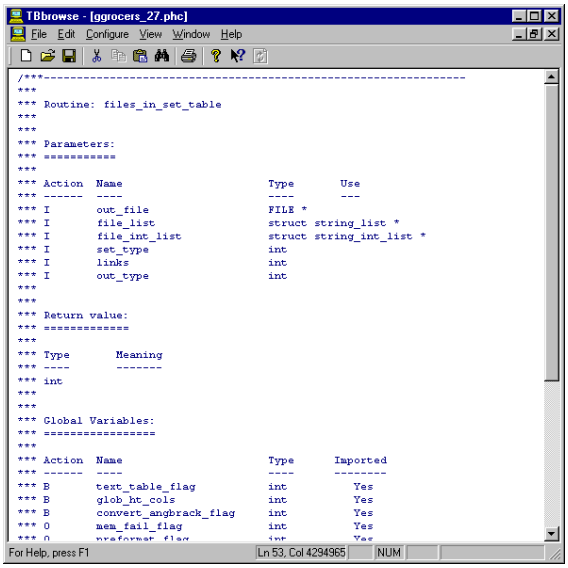
・インプットパラメータ　Input Parameters
・インプットメンバー変数　Input Member Variables
・インプットグローバル変数　Input Globals
・アウトプットパラメータ　Output Parameters
・アウトプットメンバー変数　Output Member Variables
・アウトプットグローバル変数　Output Globals
・インスタンス生成　Template Instantiations
・リターン型　Return Type



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。

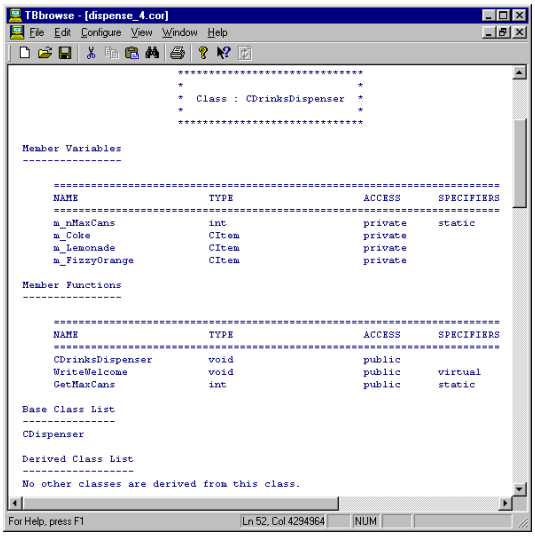
・ルーチンもしくはファイル名
・関数へのパラメータ
・リターン型
・使用されるグローバル変数
・関数コール情報



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。

・メンバ変数
・メンバ関数
・基本クラスリスト
・継承先クラスリスト



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。

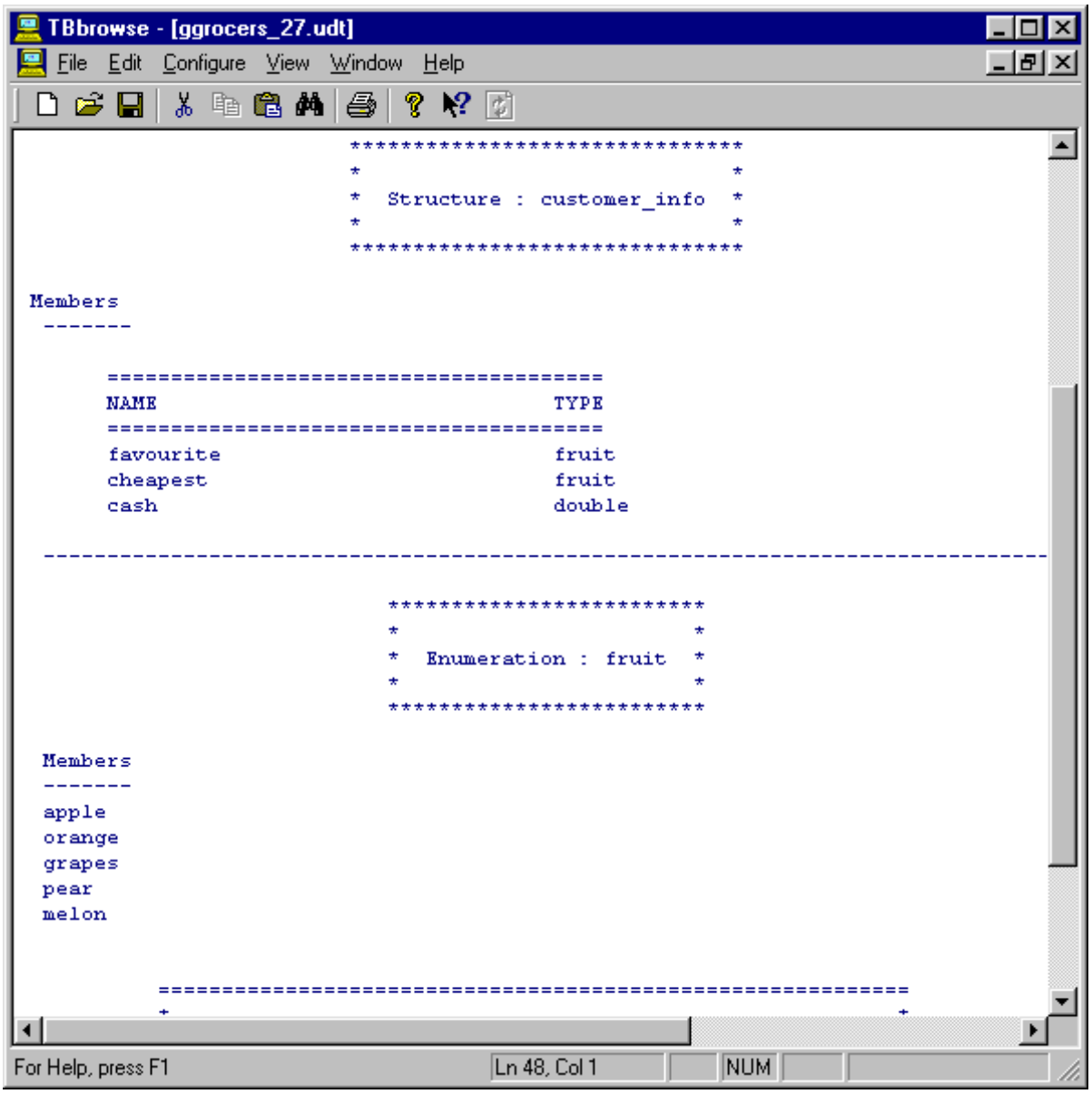
・enum　Enumerations

・構造体　Structures

・ユニオン　Unions

・タイプ定義　Type Definitions

・クラス　Classes



Instrumentation

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もなっています。

ホスト/ターゲットテスト – Host / Target Testing

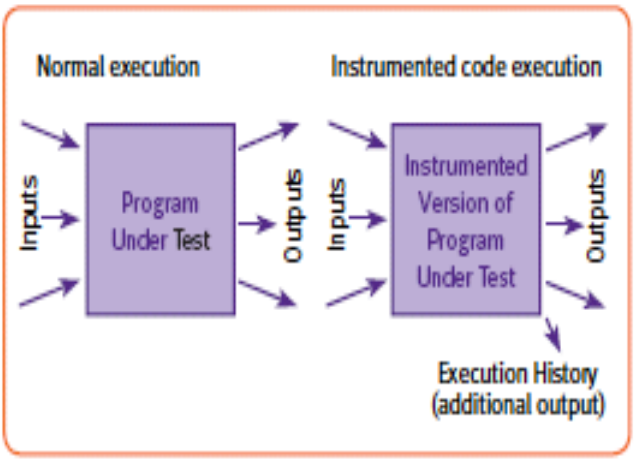
実ターゲットにおける動的テストを実現するにあたって、始めに LDRA Testbed のソースコードの静的解析が、ホストコンピュータ上で実行されます。その結果、タグ付けされたコードに対して、ターゲット固有のコンパイルなどを行って実際のターゲット MPU 上で実行されます。

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

・組込みシステム Embedded System

・シミュレータ Simulator



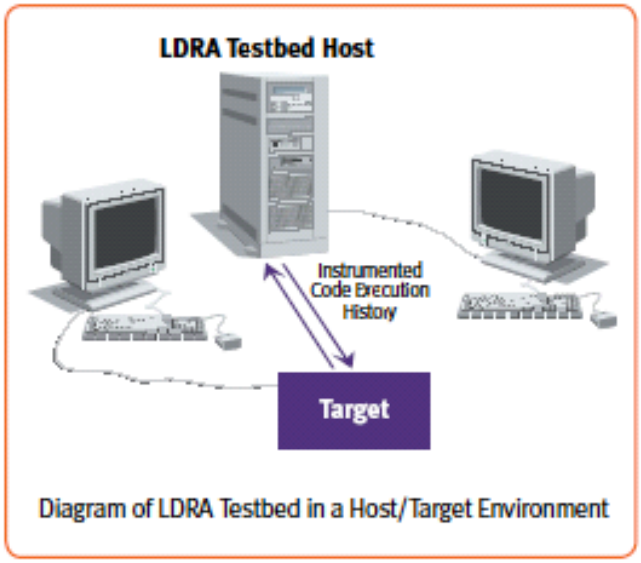


Diagram of LDRA Testbed in a Host/Target Environment

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)を判断するため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;
```

Assertion

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,1); 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;
```

Expanded Assertion

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

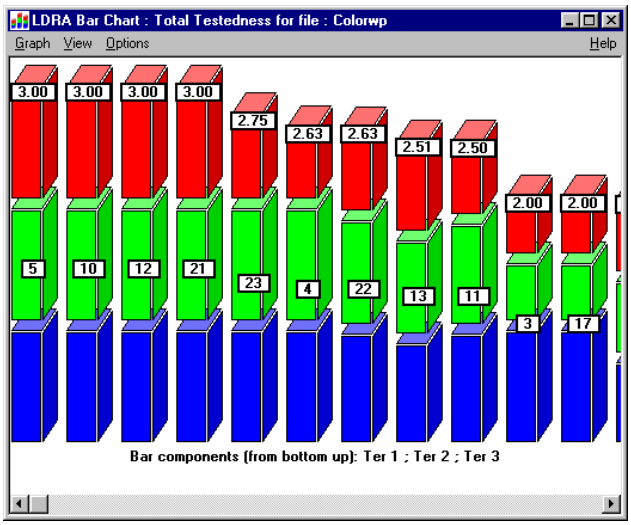
セマンティック解析 まとめ – Exact Semantic Analysis Summary

・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
・入力が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

Dynamic Analysis

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。ソフトウエアの開発・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。

$$\text{TER1} = \frac{\text{number of statements exercised at least once}}{\text{total number of executable statements}}$$

$$\text{TER2} = \frac{\text{number of branches exercised at least once}}{\text{total number of branches}}$$

$$\text{TER3} = \frac{\text{number of LCSAJs exercised at least once}}{\text{total number of LCSAJs}}$$

$$\text{P/FCall} = \frac{\text{number of Procedure/Function calls exercised at least once}}{\text{total number of Procedure/Function calls}}$$

$$\text{BCC} = \frac{\text{number of Boolean operand values exercised at least once}}{\text{total number of Boolean operand values}}$$

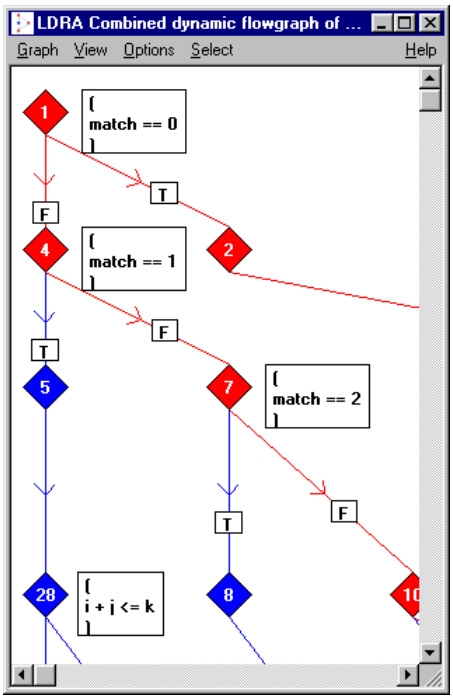
$$\text{BCCC} = \frac{\text{number of Boolean operand value combs exercised at least once}}{\text{total number of Boolean operand value combinations}}$$

$$\text{MC/DC} = \frac{\text{no of Bool. operand vals independently affect dec. outcome}}{\text{total number of Boolean operands}}$$

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 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)

・コード内のすべてのステートメントが実行される

・すべ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)

・コード内のすべてのステートメントが実行される

・すべての関数コールが呼び出される

・すべての分岐とリンクするステートメントが実行される

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

・コード内のすべてのステートメントが実行される

・すべての関数コールが呼び出される

・すべての分岐とリンクするステートメントが実行される

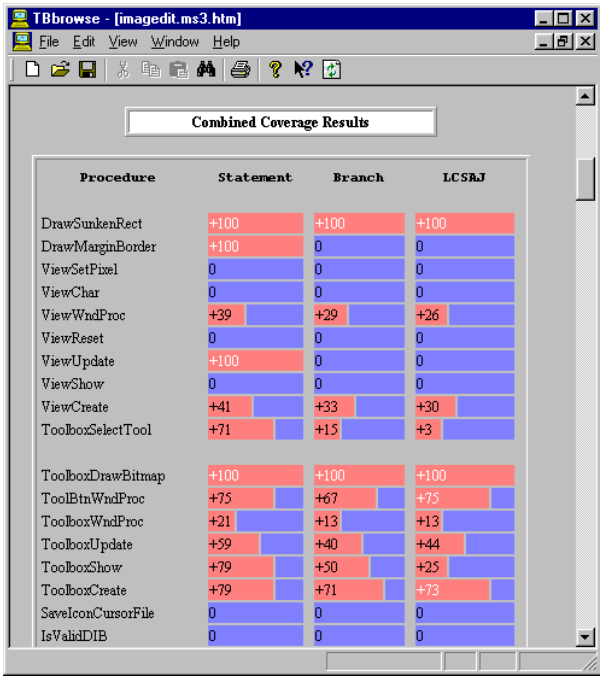
・すべての関数リターンが呼び出される

・ゼロ回実行、シングル、ダブル、トリプル、それ以上のすべてのループ実行がマルチループ内で実行される

・すべてのネスト、及びシーケンシャルループが実行される

・すべての関数終了がすべての起こりうる終了パスで呼び出される

・すべてのブランチコンディションの組合せが実行される



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

・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)

・デシ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)

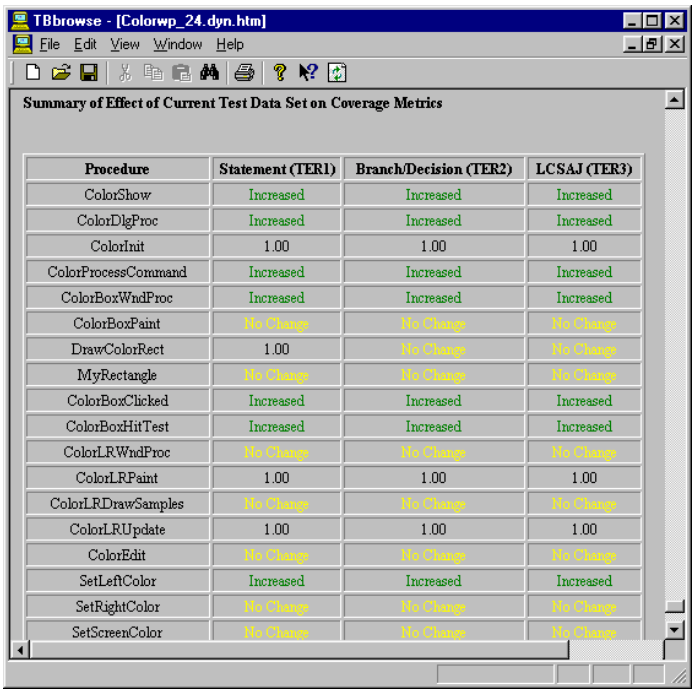
・各デシジョンコンディション(判定条件)内の各組のブーリアンオペランド値のユニークな組合せが実行される

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

·各デシ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

すべ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。



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

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ならば、LCSAJ カバレッジレベルを最大にすることをお勧めします。

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）

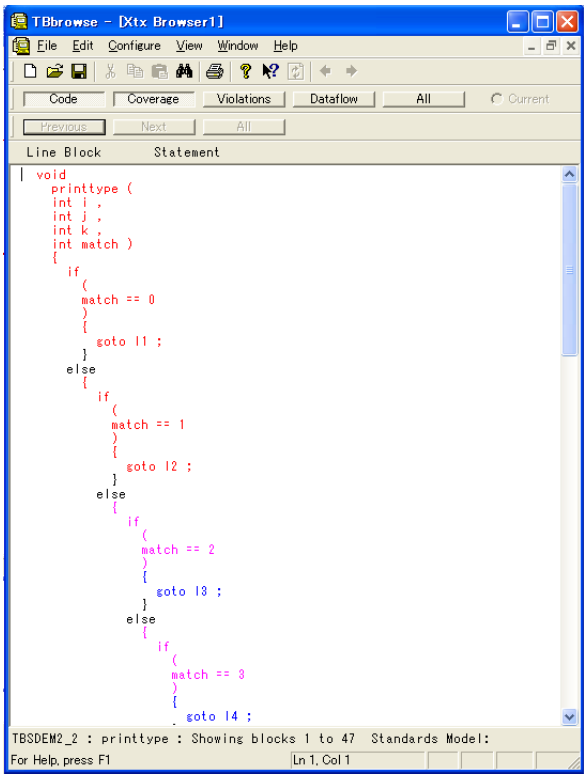
・フローグラフのコードブラウザ内の注釈付きソースコード

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



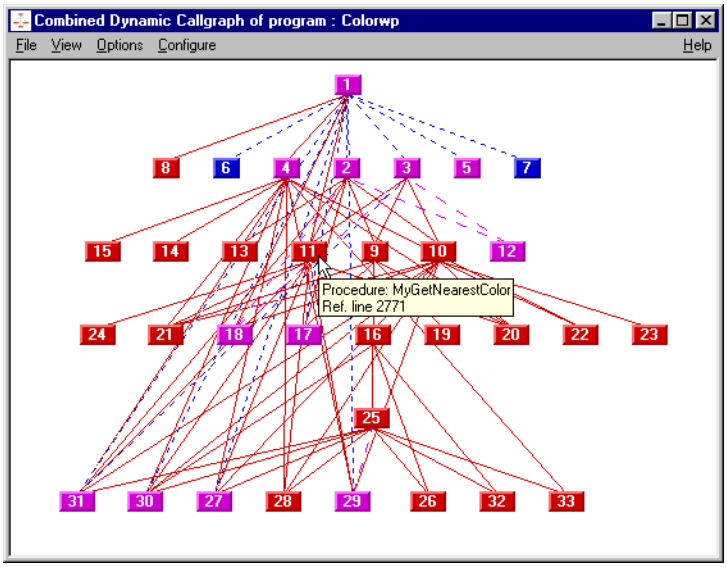
動的カバレッジ解析　まとめ – Dynamic Coverage Analysis summary

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します。

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すば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。

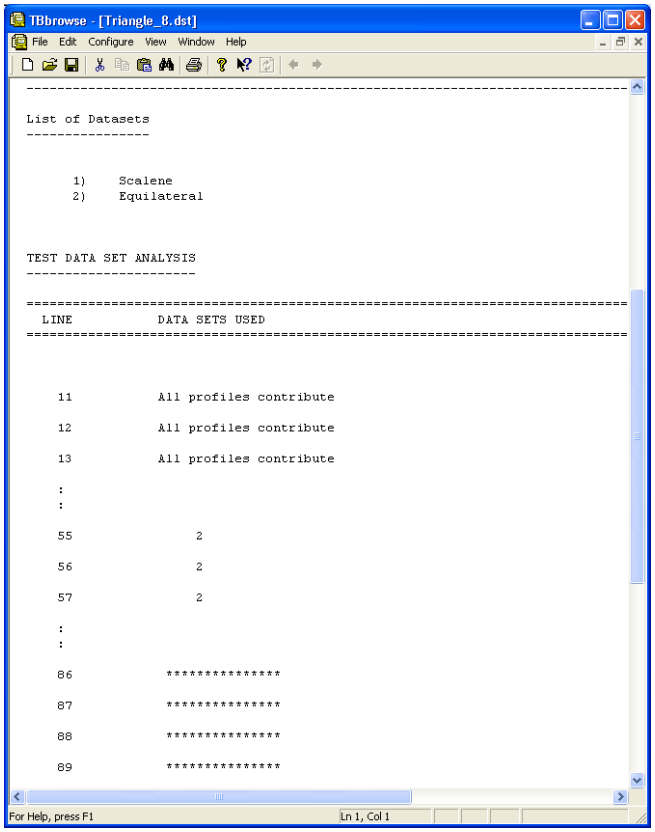
ダイナミックデータフローカバレッジは、単一ファイル、システム内、グループ内で利用することができます。フィルターインターフェイスを用いて、特定基準としてある変数のみに的を絞る事も可能です。

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

Dynamic Data Flow Table
-----------------------
=============================================================================================
VARIABLE        ALIAS        FILE           PROCEDURE    TYPE   ATTRIB.
                                                                      USED ON LINES...
=============================================================================================
k                            ddf_test1      main         L              88
                                                         L      R       92 (  94)
                                                         L      D       90
                ext_z        ddf_test2      ext_proc1    P              19
                                                         P      R    (  30)
                z            ddf_test1 proc1 P                          52
                                                         P      R       71 *****
                                                                        77
--------
j                            ddf_test1      main         L              87
                                                         L      R       92     94
                                                         L      D       90
                ext_y        ddf_test2      ext_proc1    P              18
                                                         P      R       26
                y            ddf_test1      proc1        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_proc1    P              17
                                                         P      R       23
                x            ddf_test1      proc1        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、既存のすべて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すべ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、それを実行するために必要なテストは 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。

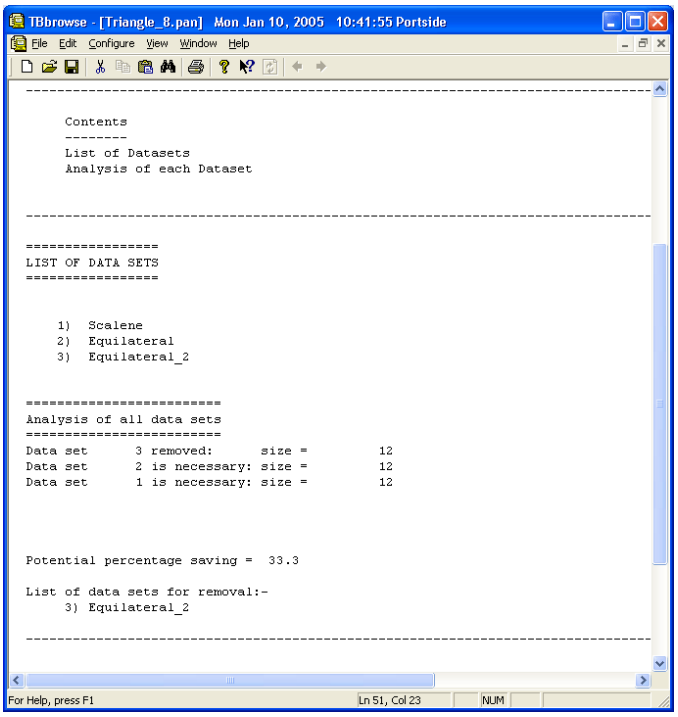
プロファイル解析 – Profile Analysis

ダイナミックカバレッジ解析率を高めるために、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で、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ァイルから削除しても、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。

プロファイル解析の目的は、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ァイ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。この解析をすべてストデータセットに対して繰返します。



プロファイル解析　まとめ – Profile Analysis Summary

プロファイル解析の目的は、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

高い信頼性が求められるソフトウエアのテストは、認証機関へ正当性を証明するため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可能です。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、すば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 16 ページのインフォメーションフロー解析を参照下さい。

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 23 ページのセマンティック解析を参照下さい。

MC/DC カバレッジ – 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ためのものです。詳しくは 26 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ために考案されました。(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)

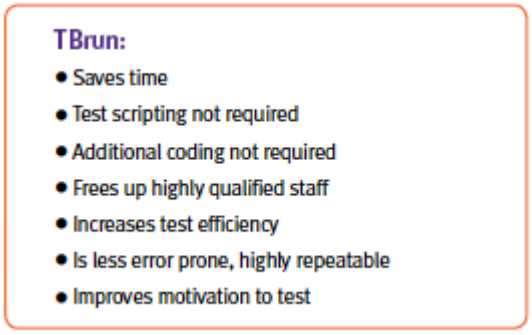
LDRA Testbed は、禁止された言語機能の使用をチェックします。 詳しくは 5 ページの“Programming Standards Checking”を参照下さい。

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:

・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

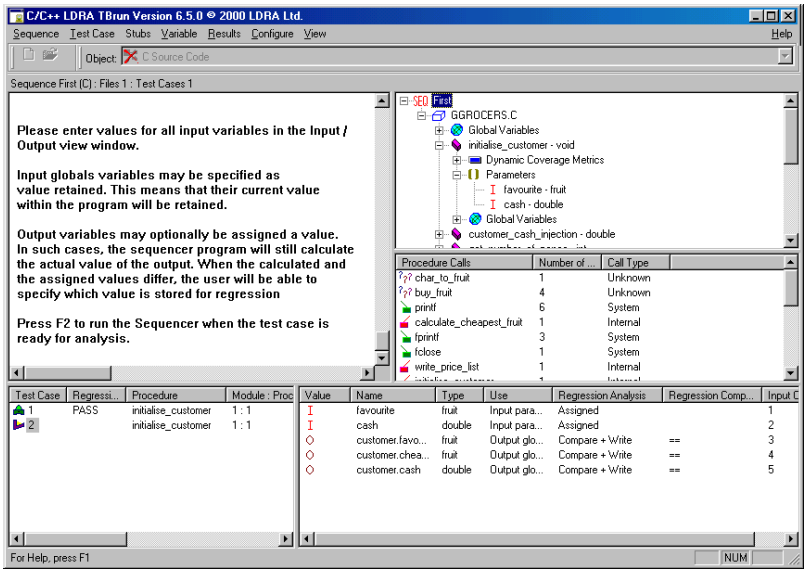
・間違いを起こしにくい、高い再現性

・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

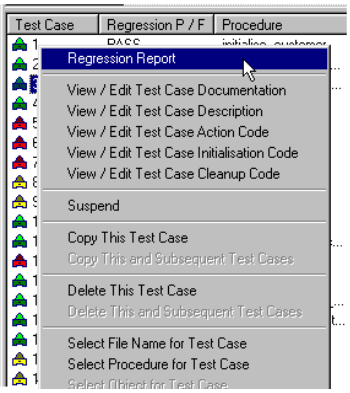
TBrun は、LDRA Testbed による解析から必要な情報を得ていますが、スタティックデータフロー解析が主要な手掛かりになっています。 これは、ユニットインターフェイスのパラメータ、グローバル変数(入出力)、戻り値、変数タイプと用法、関数コールなど、完全なソースコードのコントロールフロー解析データです。これらは自動的に収集されますので、ソースに対してのこれらに関する設定は必要ありません。

これらの解析結果から、テストデータを入力するための、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ため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

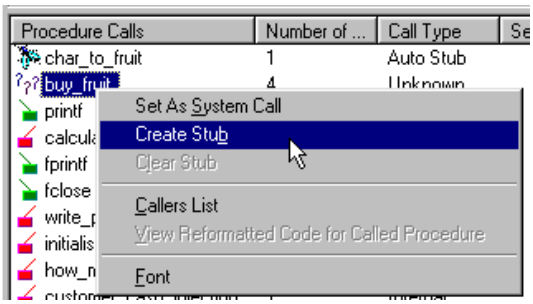
テストの実行に対して、トレーサビリティ情報を付加することで、自動でリグレッションテストの管理ができるようになります。テストとその結果はPVCSなどの、コードやテストのバージョンコントロールシステムへ簡単に保管することができます。そのため、ユニットへのリグレッションテストをビルドとテストサイクルの一部とする事ができるようになり、テストプロセスに対して柔軟に適応させることができます。



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すべてのユニット、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

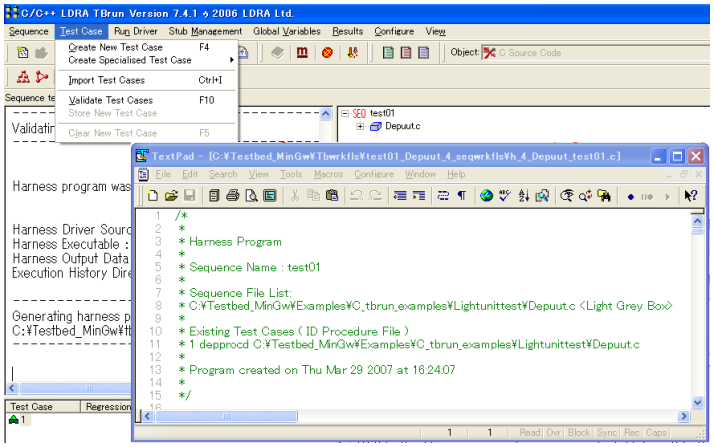
・テストドライバとハーネス（ラッパーコード）の自動生成
・コード単体でテストを実行－コーディングの必要なし
・ソースコードの変更を検出
・テストに必要な変更を文書化
・リグレッションテストの実行
・スタブの自動生成
・テストデータと結果の保持
・ホスト/ターゲット環境での実行
・コードカバレッジの尺度を収集
・使いやすい GUI
・コマンドラインインタフェースにも完全対応

TBrun Features

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

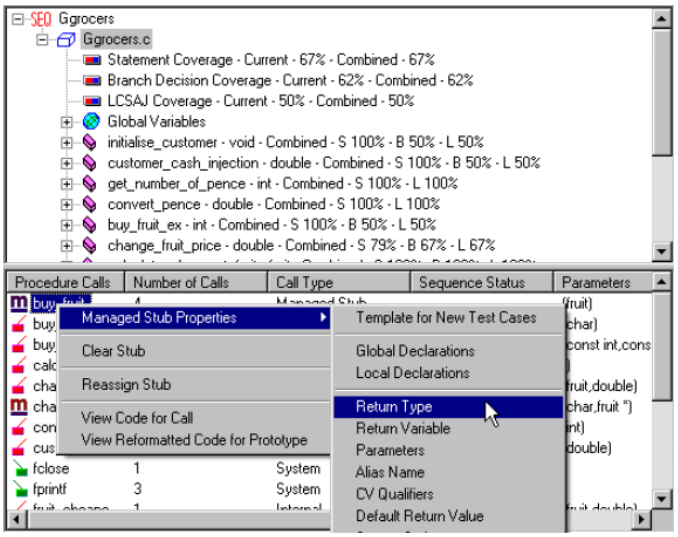
ドライバプログラムの自動生成 – Automatically Generated Driver Program

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るので、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ーへの制限はありません。ドライバーは純粋な 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。



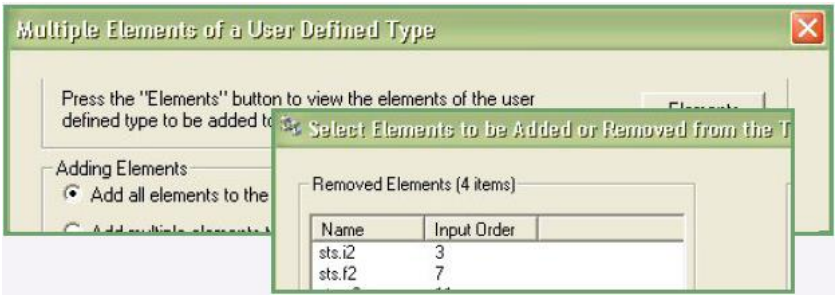
スタブ生成 – Stub Creation

スタブはワンクリックで生成されます。生成されたスタブは戻り値、値チェックなどを含んでいるので、これらをコーディングする必要はありません。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。



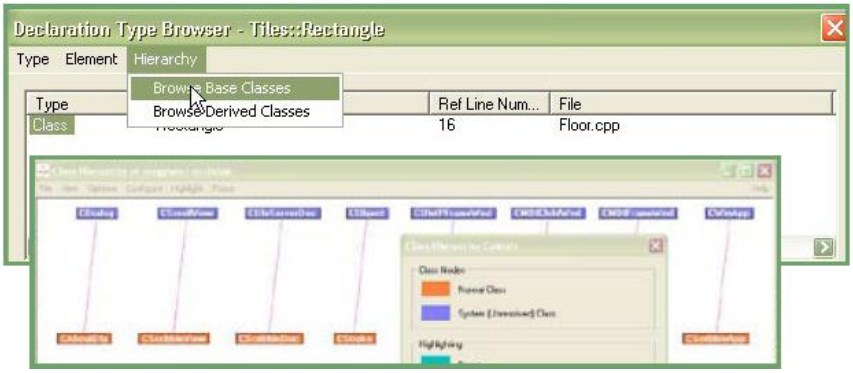
構造体/配列/共用体 – Structures/Arrays/Unions

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。



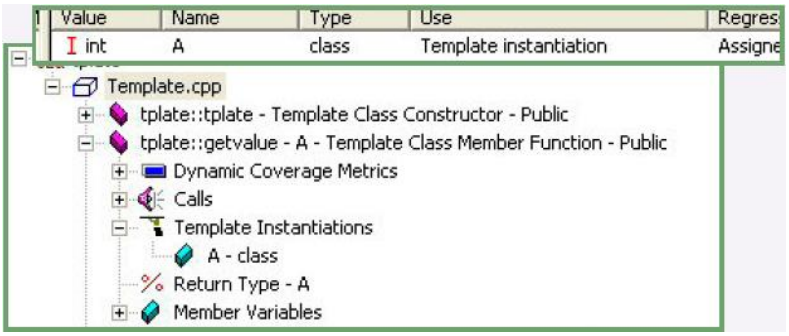
クラスの扱い – Class Handling

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。



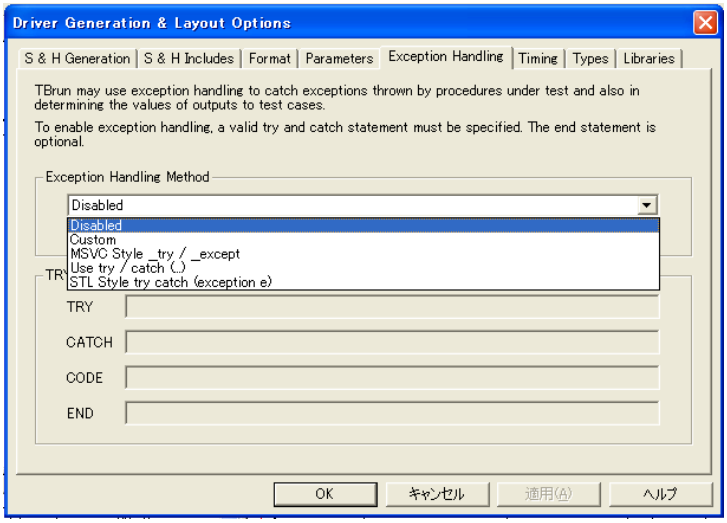
テンプレートタイプの自動解決 – Automatic Resolution of Templated Type

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。



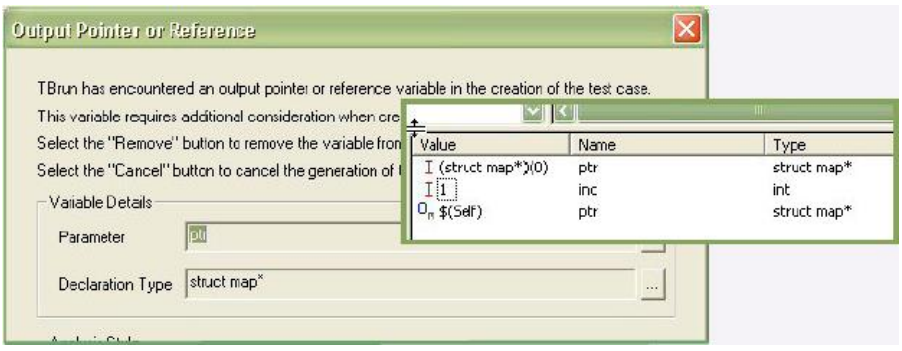
例外処理 – Exception Handling

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。型の拡張とリンクト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ー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

・Abstruct 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
・全階層に渡る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てい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ためのラッパーコードは要りません。すべ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。

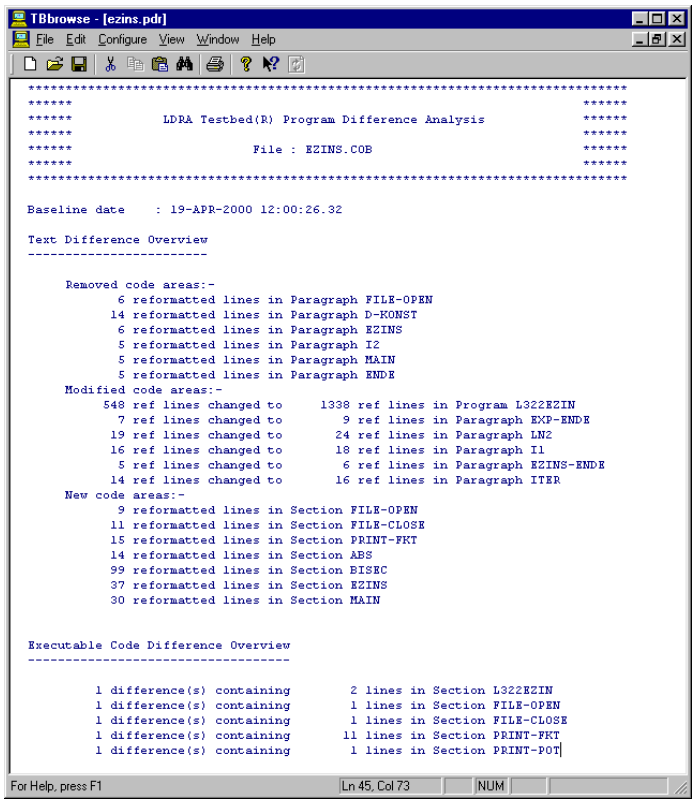
更なる利点は、テストデータセットが自動的にメンテナンスされる事です。ソースコードが変更されると、テストデータに変更が必要な場所が表示され、アップグレードが容易にできます。その結果、マニュアルでのアップデートや追跡作業と比較して多大な時間とコストの節約になります。

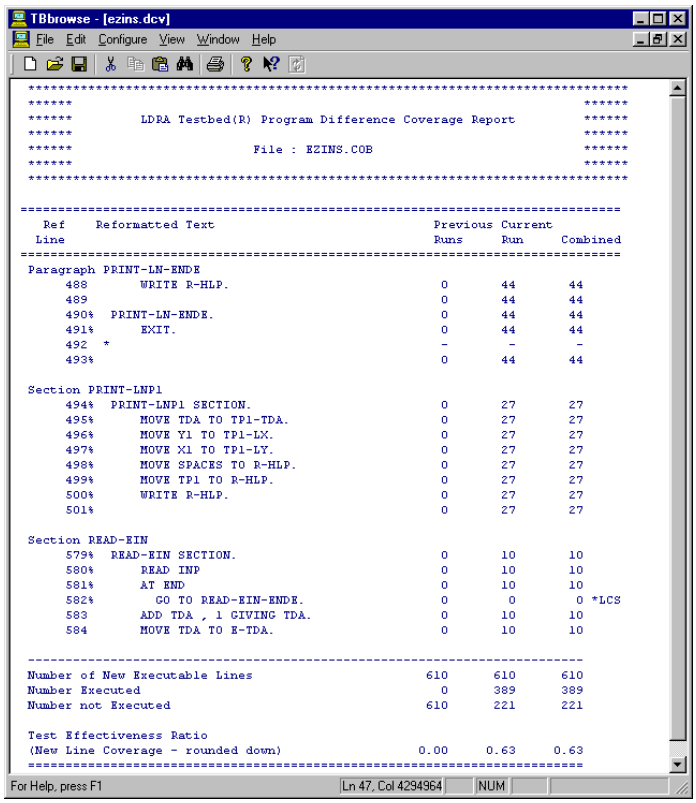
TBevolve™ Software Testing

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により、ドミノ倒し的に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。





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。

富士設備工業株式会社　電子機器事業部
TEL：072-252-2128
E-Mail：　info@fuji-setsu.co.jp
http://www.fuji-setsu.co.jp

FUJI SETSUBI