Imagix 4D
ユーザガイド

Imagix

Imagix 4D
ユーザガイド

Imagix Corporation
6025 White Oak Lane
San Luis Obispo, CA 93401
http://www.imagix.com

はじめに

Imagix 4D は、複雑であるか、サードパーティ製であるか、あるいは旧式である C、C++、Java のソフトウェアの解析、ドキュメント作成、および改善に役立ちます。このソースコード解析ツールは、コードのブラウジングと解析を自動化し、大規模なプログラムや未知のプログラムを素早く理解することができるようになっています。このツールを利用すれば、高レベルのアーキテクチャから個々の機能の詳細なプログラムロジックまで、あらゆるレベルでのリバース・エンジニアリングやソフトウェアの調査が可能です。また、制御構造、データ用法、クラス継承など、ソフトウェアのさまざまな側面を幅広くビジュアル化して調査することもできます。 プログラムをより早く、より正確に理解することができるようになる結果、生産性も高まり、ソフトウェアの欠陥も減らすことができます。

Imagix 4D の品質解析機能は、ソフトウェアの開発及び維持管理における潜在的問題の特定に役立ちます。変数の使用、タスクの連動、および割込みからの保護に関する一連の解析検証を行うことで、ユーザはリアルタイム組み込みの、マルチタスキング、マルチスレッド化されたシステムで起こりうる衝突を見極めることができます。ソースチェックを利用すれば、明瞭性および可搬性についての設計上及びコーディング上の例外を発見することができます。そしてソフトウェアメトリックスは、開発プロセスを管理し、テスト容易性および保守性に関する組織内の評価基準を満たさないソフトウェアの部分を特定するうえで役立ちます。

また、Imagix 4D は構成の解析、コードのレビュー、システム設計に関するドキュメントの作成に伴う面倒な作業を大幅にカットし、RTF や HTML などの多彩な形式でドキュメントを自動生成することにより、正確で最新の情報を提供します。

この「ユーザガイド」は Imagix 4D のテキストまたは使用説明書としてお使いいただくものです。この「はじめに」の項では、まずこのツールになじんだ上で、ソースコードを Imagix 4D にロードする手順を、段階を追って説明します。

コードのローディング

Imagix 4D を用いてソフトウェアを調査する場合には、はじめにソースコードをインポートする必要があります。以下の手順にしたがって、Imagix 4D にコードをロードしてください。

Imagix 4D では、プロジェクトは調査するソフトウェアに関する情報のリポジトリをさします。ソフトウェアをロードするときには、まず新規に空のプロジェクトを生成した上で、そこにソースコードから情報を収集します。

このプロセスでは、Imagix 4D によるソースコードの解析が中心になります。Imagix 4D のアナライザは、コンパイラのようにコードを解析します。コンパイラを使用する場合と同じで、このアナライザには、ソースファイルのリストとともに、インクルードファイルを探す場所、及び解析に使用するマクロ定義を指示するオプションのリストを渡します。コンパイラは、実行ファイルにリンクされるオブジェクトファイルを生成しますが、Imagix 4D のアナライザは、Imagix 4D のデータベースにロードされ、統合されるデータファイルを生成します。

ソースコードのインポートは次の手順で行います。

1. ライセンスをインストールする
2. 解析方法を選択する
3. 使用環境に合わせて Imagix 4D を構成する
4. 新しいプロジェクトを生成する
5. コードをロードする
6. アナライザの設定を微調整する

ライセンスをインストールする

1a. 適切なライセンスをインストールする

Imagix 4D にコードをロードするには、デモ用ではなく、評価用またはプロダクトライセンスが必要です。メインメニューで New Project の機能（メニュー[File] > [New Project]）が無効になっている場合には、適切なプロダクトライセンスがインストールされていません。Imagix 社または販売代理店に連絡し、ライセンスを入手してください。インストールの方法は、/imagix/readme に説明しています。

解析方法を選択する

2a. 解析方法を決定する

Java コードを Imagix 4D にロードする方法は単純です。ダイアログに解析対象のソースファイルを指定し、.class ファイルおよび.jar ファイルをインポートするためのクラスパスを指定します。

C および C++のコードの場合、ソースコードに対して Imagix 4D を起動する方法はいくつかあります。第 1 の方法では、ダイアログで以下の項目を指定します。

-- ソースファイルが置かれるディレクトリ
-- ソースファイルの名前（拡張子*.c を使用）
-- 使用する-I、-D、-U プリプロセッサオプション

第 2 の方法でもデータソース定義を含むダイアログを使用しますが、メイクログから情報を抽出することによって、これらのダイアログに自動的にデータが書き込まれます。ソースコードをコンパイルする際に、make が発行したコマンドをログファイルに記録している場合は、この方法を用いてそのメイクログを処理し、コンパイラ起動コマンドを切り離すことで、ソースファイルおよびコンパイラに渡された引数の-I、-D、-

Uを特定することができます。これ以降、同様のオプションを共有するソースファイルのグループごとに、各データソースダイアログにデータが埋め込まれます。

第 3 の方法では、ターゲットをメイクファイルに追加し、メイクのシステムを控え目ではなく活発に利用します。メイクファイルにはソースファイルとプリプロセッサオプションに関する情報が含まれているので、この第 3 の方法では既存の定義を利用します。ソフトウェア開発進行中には、メイクファイルにターゲットを追加する方法にも利点はありますが、この方法から取り掛かるのは一般的にはるかに難しいため、初期のツール評価にはおすすめしません。

最後の方法は、Microsoft Visual Studio (MSVC)のプロジェクトとソリューションをサポートするものです。Visual Studio によって生成されたメイクファイル、.dsp、.vcproj または.vcxproj ファイルから直接ファイル及びプリプロセッサオプションの情報を抽出します。ソリューション(.NET より古いバージョンではワークスペース)では、.dsw または.sln ファイルが読み込まれます。通常、Microsoft Visual Studio を使用してコードをビルドする場合には、この方法をおすすめします。

実際に解析方法を選択するのは 5a のステップです。メイクファイルを通して解析を行う場合を除き、ステップ 3b は不要です。

使用環境に合わせて Imagix 4D を構成する

(C または C++のソースコードを処理する場合、手順 3a は複雑になる可能性があり、どの方法をとる場合にもオプションとなります。ただし、あとの手続きが大幅に単純化されるので、この手続きを行うことを強くおすすめします。Java コードの場合、手順 3a は非常に簡単です。ターゲットをメイクファイルに追加する方法を選んだ場合のみ、手順 3b が必要になります。)

3a. コンパイラ設定ファイルを選択/設定する(C/C++)

警告 – C および C++のソフトウェアの場合、プリプロセッサオプションを正しく指定するのは、概念的にも、技術的にも、コードをロードする上で最も注意を要するところです。

原理

ソースコードをコンパイルするときには、メイクファイル、ビルドスクリプト、または IDE がソースファイルのリストとともに一連の-I、-D、-U オプションをコンパイラに渡します。これらのオプションは、コンパイラがどこでヘッダファイルを探し、どのマクロ定義をコード解析に使用するかを判断するために使用します。

これらのマクロは、ソースコードのどの部分を処理し、どの部分がアナライザのビルトインプリプロセッサによって無視するかを制御するため、#ifdef ステートメントと関連付けて使用されます。たとえば、PP Flags(プリプロセッサオプション)フィールドに-Dfoo と入力した場合、コードのなかに#ifndef foo というステートメントが含まれていれば、次の#endif ステートメントのところまで、以下のコードはスキップされます。メイクファイルまたはビルドスクリプトによって渡すオプションの指定は、5 で行います。

コンパイラは、明示的に渡されるこれらのプリプロセッサオプションのほかに、暗黙的な独自の-I、-D、-U オプションも使用します。これらは、システムのヘッダファイルを見つけ、正確に解析をするために必要なものです。

Imagix 4D のアナライザはコンパイラに依存しません。コンパイラ固有のヘッダファイルの場所、及びその解析に使用するマクロ定義を指定しておけば、Imagix 4D のアナライザをコンパイラの動作に適合させることができます。

方法

アナライザをコンパイラに適合して動作するように設定する方法は、ふたつあります。ひとつは、(5 で)コンパイラに明示的に渡される正規の-I、-D、-U オプションを指定するときに、コンパイラが暗黙的に追加する-I、-D、-U オプションを追加する方法です。

もうひとつは(こちらのほうを強くおすすめしますが)、Imagix 4D のコンパイラ設定ファイルを利用する方法です。この方法では、ひとつのコンパイラ設定ファイルでシステムヘッダファイルの場所とマクロ定義を指定した上で、Data Sources ダイアログ(メニュー[Project] > [Data Sources])を利用し、使用するコンパイラ設定ファイルを指定します。

コンパイラ設定ファイルは、../imagix/user/cc_cfg のディレクトリに置かれています。ここには、複数の.inc ファイルがあります。ファイルのベース名はコンパイラ、及びその設定ファイルがサポートするターゲットプラットフォームを示しています。

コンパイラ設定ファイルは、新規に追加することもできますし、すでにあるものを修正することもできます。通常はシステムヘッダファイルが実際にインストールされている場所を反映させるため、あらかじめ存在するファイルで定義されているシステムインクルードディレクトリの場所を変更する必要があるでしょう。

コンパイラ設定ファイルを利用する方法には、単純に正規の-I、-D、-U オプションに暗黙的にコンパイラが-I、-D、-U オプションを追加する方法に比べ、いくつかの利点があります。まず、ご自分のコンパイル環境に適した設定ファイルがすでに存在していれば、少し手間が省けます。それに、コンパイル環境が変わらなければ、最初に一度情報を指定するだけでよく、毎回、新しいプロジェクトを生成する必要はありません。さらに、繰り返し設定を調整するのも容易です。

手順

/imagix/user/cc_cfg のディレクトリを開きます。ご自分のコンパイル環境に適合する設定ファイルをお探しください。適当なファイルが見つかったら、ご自分の環境に応じてインクルードディレクトリを修正します。見つからなければ、新しいコンパイラ設定ファイルを生成し、ご自分のコンパイラ及びターゲットプラットフォームを示す名前をつけておきます。そのさい、まったく新しいファイルを生成してもかまいませんが、通常はご自分の環境に近いコンパイラ/ターゲット用の設定ファイルが存在しますので、それをコピーして修正したほうがよいでしょう。あとで解析結果をレビューし、コンパイラのドキュメントを読んでいるうちに定義を微調整する必要が出てきたときの手順は、6 で説明します。コンパイラ設定ファイルの詳細については、../imagix/user/cc_cfg ディレクトリの readme.txt ファイルに記載されています。

3a. コンパイラ設定ファイルを選択/設定する(Java)

Java コードの場合、言語設定ファイルを使用することにより、標準のシステム.jar ファイルに対するクラスパスを 1 度だけ指定すればよく、新しいソースファイル群の解析の都度それを指定する必要はありません。指定するには、../imagix/user/java_cfg ディレクトリ内の適切なファイル内に、ご使用のシステムの.jar ファイルに対するクラスパスを追加または変更します。別々のソースファイル群に対して異なる.jar ファイルを使用する場合、一連の言語設定ファイルを作成して、コードを新しいプロジェクトにロードする際にファイルを選択することもできます。

3b. メイクコマンドを設定する

ターゲットをメイクファイルに追加する方法を選んだ場合、実際には make コマンド(Windows では nmake)でメイクファイルを起動します。別の呼び出しで make を起動するのを標準とする場合には、それに応じて Imagix 4D を設定する必要があります。Options ダイアログの Data Collection において、make の名前フィールドを変更する必要があります(メニュー[Tools] > [Options] > [Data Collection])。Windows では、環境変数 PATH でパスが指定されていない場合、ダイアログを用いて make(または nmake)実行ファイルへのパスも指定する必要があるかもしれません。

新しいプロジェクトを生成する

4a. 新しいプロジェクトを生成する

New Project ダイアログを開きます(メニュー[File] > [New Project])。Directory フィールドに、生成するプロジェクトを置きたいディレクトリを参照します。自分に書き込みパーミッションがある現存するディレクトリでなければなりません。多くの場合、ユーザはプロジェクトの場所がすぐにわかるように、ソースコードが存在するディレクトリを使用します。しかし、プロジェクトはどこへ置いてもかまいません。

Name フィールドには、プロジェクトにつけたい名前を入力します。Imagix 4D はその名前のついた新しいディレクトリを生成し、それが Imagix 4D プロジェクトのディレクトリだとわかるように.4D の拡張子を追加します。

オプションの Description フィールドにプロジェクトに関するコメントを入力することもできます。プロジェクトの数がふえてくると、このコメントがプロジェクトの内容を思い出すのに役立ちます。また、このコメントは、Project List ダイアログを通してあとで追加することもできます。

OK ボタンをクリックします。

コードをロードする——ダイアログを通して解析を行う場合(C/C++)

5a. 新しいデータソースの追加を指定する

Data Sources ダイアログを開きます(メニュー[Project] > [Data Sources])。既存のデータソースの設定を修正するのではなく、プロジェクトに新しいデータソースを追加することを指定するには、ダイアログの左側の Data Sources の下にある[+ new data source]を選択します。これは新しいプロジェクトであり、既存のデータを修正しないため、選択可能なオプションはこれだけです。

5b. ダイアログを通して解析を行う方法を選択する

以降のステップは、Data Sources ダイアログの右側での作業になります。上部にある、Select Data Source Type というラベルのメニューボタンから、[Source Files] > [Dialog Based (C/C++)]を選択します。

5c. 解析するソースファイルを指定する

Source Files タブの Directory フィールドに、ソースコードが置かれているディレクトリを入力します。

注:ソースコードが複数のディレクトリにまたがっている場合には、Directory フィールドの下にある Analyze source files in subdirectories オプションを利用することもできます。解析しようとしているコードが、ひとつのディレクトリとそのサブディレクトリ(さらにそのサブディレクトリ)に展開している場合には、前記の Directory フィールドに最上位ディレクトリを入力します。特定のサブディレクトリを除外するには、Exclude ダイアログで指定します。ソースコードがディレクトリ構造で展開している状態によっては、解析するコードを含む各ディレクトリについて、5 を繰り返します。

Source Files フィールドにソースファイルの名前を入力します。アスタリスク(*)を含むパターンで指定すると、対象が拡大されます。たとえば、*.c と指定すると、指定したディレクトリに含まれるすべての.c ファイルが対象となります。複数の名前及び/またはパターンを、スペースで区切って入力することもできます。h ファイルは、指定する必要はありません。アナライザは、指定した.c または.cpp ファイルのひとつに含まれるあらゆる.h ファイルを自動的に解析します。拡張子の異なるファイルを指定する場合には、*.c *.cpp のように、それらのファイル名の間をスペースで区切ってください。

5d. ソースコードの解析に使用する-D 及び-U オプションを指定する

さらに Source Files タブにおいて、PP Flags フィールドに適宜'-D*macroname* -U*macroname*'を入力します。-D 及び-U オプションは、必要な数だけ入力してかまいません。-D または-U とそのあとに続く名前の間をスペースで区切らないでください。名前と次の-D または-U との間はスペースで区切ってください。

-D、-U オプションは、それぞれマクロを定義済み、未定義にします。マクロ名に対するマクロ置換値を定義する場合には、-Dmacroname=value のように指定します。等号(=)の前後には、スペースを入れないでください。

5e. 標準的なコンパイラ環境を指定する

さらに Source Files タブにおいて、使用するコンパイラ及びビルドのターゲットプラットフォームを指定します。Compiler & Target コンボボックスに該当するコンパイラとターゲットの組み合わせがある場合には、それを選択します。まだない場合には、3a に戻り、適切なコンパイラ設定ファイルを生成することをおすすめします。あるいは、前記のコンボボックスで other を選択します。other を選択した場合には、PP Flags フィールドに暗黙的なオプションを指定する必要があります(5d 参照)。

5f. インクルードするヘッダファイルの場所を指定する

5c.で説明したように、ヘッダファイルを解析するために明示的に Imagix 4D を指定する必要はありません。代わりに、ソースコードの#include ステートメントを使用して、インクルードされたヘッダファイルを検索するインクルードディレクトリだけを指定します。これは Include Dirs タブで指定します。

Include Dirs タブには、ディレクトリを指定する方法がふたつ用意されています。いずれかの方法を使用するか、または両方同時に使用できます。ヘッダファイルが、ひとつのディレクトリとそのサブディレクトリ(さらにそのサブディレクトリ)に展開している場合には、タブの[Specify Include Directories By Root Directory]部分を使用します。最上位のインクルードディレクトリの名前を Directory フィールドに入力し、適宜 Search subdirectories for header files 及び Exclude ダイアログに入力します。

ヘッダファイルが、分散したインクルードディレクトリに展開している場合、またはインクルードディレクトリが検索される順序を制御したい場合は、Specify Include Directories Individually フィールドで、適宜-I*dirname1* -I*dirname2* を入力します。-I は必要な数だけ入力してかまいません。-I とディレクトリ名との間には、スペースを入れないでください。ディレクトリ名とそのあとに来る-I との間はスペースで区切ってください。Windows 環境下では、ディレクトリ名にスペースが含まれる場合には、"-Ic:/program files/msvc/include"のように、-Idirname の部分をダブルクォーテーションでくくります。

-I オプションは、アナライザにインクルードファイルを検索するディレクトリを伝えます。インクルードファイルは、指定された順番に検索されます。パス名は、-I../parallel/example のように、Source Files タブの Directory フィールドに指定されたディレクトリに対する相対パスで指定します。

5g. 解析プロセスをスタートさせる

これでアナライザを起動する準備ができました。ただし、コードを解析しようとすると、設定、特にインクルードディレクトリと-D フラグの指定方法にエラーがあったことがわかるのがふつうです。最初はコードのひとつのサブセットだけをロードするようにしたほうがよいでしょう。その場合には、Source Files タブの Source Files フィールドで、特定のファイルをひとつだけ初回の解析対象として指定します。

コードを解析する準備ができたら、ダイアログの Add Data Source ボタンをクリックします。

コードをロードする――メイクログからの抽出を通して解析を行う場合

5a. make によって発行されたコンパイラコマンドを含むログファイルを生成する

この方法を採るには、ソースコードをコンパイルする際にメイクシステムが発行するコマンドを含む、ログファイルが存在する必要があります。これらのうち最も重要なコマンドは、お使いのコンパイラに対して発行されたコマンドです。また、コンパイラに渡された相対パスから絶対パスを正確に割り出せるように、ディレクトリの変更を引き起こすコマンドも必要になります。

メイクログの作成にかかわる実際のプロセスはビルド環境によって大きく異なるため、ここではその原理についてだけ説明します。make が実行されるとき、ビルドに必要な各コマンドラインが標準出力に表示されたのちに実行されます。メイクログの一般的な生成方法は、コードをコンパイルする make を起動して、表示されたコマンドをファイルにキャプチャまたはリダイレクトすることです。make を-silent や-quiet のようなオプションを付けて実行している場合は、それらを無効にして、make コマンドが表示されてログに記録されるようにする必要があります。make が実行中のコマンドを表示させるようにするには、別のオプションを追加する必要がある場合があります。

この方法ではダイアログを通したデータソース定義が書き込まれるので、メイクログから抽出された設定の変更が可能になります。しかし、make のビルドが作成したいプロジェクトに非常によく適合する場合、ソースファイルがコンパイル済みであり、-I、-D、-U オプションがコンパイラに渡されていれば、この方法が最も簡単です。

5b. 新しいデータソースの追加を指定する

Data Sources ダイアログを開きます（メニュー［Project］> ［Data Sources］）。既存のデータソースの設定を修正するのではなく、プロジェクトに新しいデータソースを追加することを指定するには、ダイアログの左側の Data Sources の下にある［+ new data source］を選択します。これは新しいプロジェクトであり、既存のデータを修正しないため、選択可能なオプションはこれだけです。

5c. メイクログからの抽出を選択する

以降のステップは、Data Sources ダイアログの右側での作業になります。上部にある、Select Data Source Type というラベルのメニューボタンから、[Source Files] > [Make Based] > [Extract from Makelog]を選択します。

メイクログからの抽出によって、ダイアログの左側に一連のデータソースが追加で定義されているのが後に確認できます。この過程で、既存のデータソースはすべて失われます。データソースが不用意に失われないようにするため、他のデータソースが全く定義されていない場合にのみ、Extract from Makelog が選択可能にされます。

この方法によって定義されたデータソース以外にも、プロジェクトに追加したいデータソースがあるかもしれません。そのような場合には、追加データソースを含む別のプロジェクトを作成してから、ComboProject 機能を使用して 2 つのプロジェクトを合体させるのが最適な方法です。(詳しくは、ユーザガイドの「コンボプロジェクト」の章を参照してください。)

5d. メイクログの名前を指定する

Makelog タブの[Makelog file]フィールドに、ログファイルのフルパス/ファイル名を入力します。

5e. 追加のコンパイラオプションとソースアナライザオプションを指定する

通常、Options フィールドは空のままにします。メイクログに出現する-I、-D、-U のコンパイラオプションはすべて自動的に抽出されます。しかし、Imagix 4D でのソースコードの解析中に使用したいコンパイラオ

プションの一部がメイクログに書かれていない場合、ここでそれらを加えることができます。また、Imagix 4D ソースアナライザオプションを適用したい場合はそれも追加できます。(詳しくは、ユーザガイドの「アナライザの構文とオプション」の章を参照してください。)

5f. 標準的なコンパイル環境を指定する

使用するコンパイラ及びビルドのターゲットプラットフォームを指定します。Compiler & Target コンボボックスに該当するコンパイラとターゲットの組み合わせがある場合には、それを選択します。まだない場合には、3a に戻り、適切なコンパイラ設定ファイルを生成することをおすすめします。あるいは、前記のコンボボックスで other を選択します。other を選択した場合には、Options フィールドに暗黙的なオプションを指定する必要があります(5e 参照)。

5g. 処理ルールをレビューして変更する

Processing Rules タブには、メイクログの解析と情報の抽出をどのように行うかを制御する一連のルールが表示されます。ルールをメイクログと突き合わせて見直すことによって、メイクログの書式に適合するルール設定を指定することができます。個々の設定に関する説明は、カーソルをルールの上に移動させたときに現れるツールチップを参照してください。

これが、繰り返し処理の始めになります。現在の設定を使用したメイクログの処理結果を調べたのち(手順 5i)、この手順に戻ってルールを改良することになるでしょう。結果に満足するまで、この手順を繰り返すことが可能です。

5h. メイクログを処理する

これでメイクログを処理する準備ができました。ダイアログの下部にある Process Makelog をクリックします。

5i. 結果をレビューして処理ルールを改良する

Imagix 4D は、メイクログ処理結果のレビューに使用するフィードバックのソースをいくつか備えています。最初に表示されるインジケータは、処理中に識別されたソースファイルの数をリストするダイアログです。このダイアログから、プロジェクト設定を生成することが可能です。生成の実行を選択した場合、個々のデータソースは Data Sources ダイアログの左側にあるリスト内に定義されます。それらをクリックすると、どのソースファイルが識別されたのかに加えて、各ファイル集合に-I、-D、-U オプションのどれが関連付けられたのかをレビューすることができます。

特に有効なのは、メイクログの処理中に生成される抽出ログです。これらのログは、Imagix 4D のプロジェクトディレクトリ(手順 4a)にある、extract_xxx.log というタイトルのファイル群です。別々のログに、コンパイラの動作、コンパイラの詳細、ディレクトリ変更動作、および処理の警告が書き込まれます。これらのログには、メイクログの処理後に Data Sources ダイアログの右側に表示される Status タブからアクセスできます。

結果のレビューとルールの修正は反復的処理であり、解析と演繹的推理をユーザ側で行う必要があります。ここに、この処理をより容易にする方法をいくつか示します。

ソースファイルを識別する

最初は、メイクログで参照されているソースファイルを特定することに集中します。

"Include files only if currently found in file system"というルールを無効にして作業を始めます。こうすると、ディレクトリ解決の問題で以前の結果に影響が及ぶことが避けられます。

ソースファイルが見つからない場合は、Compiler ログを参照してください。これは、メイクログのどの行がコンパイラ起動コマンドを含むものとして特定されたかを示します。コンパイラ起動コマンドが見つからな

い場合は、まずメイクログを調べてください。ここで手動によりコンパイラ構成を特定できない場合は、メイクログを再生成する必要がある可能性があります(手順 5a)。コンパイラ起動行が存在するものの、Compiler ログにしたがってそれが特定されない場合、Compiler Invocation ルールをチェックしてください。指定したコンパイラコマンドが、メイクログに書かれているコマンドと一致していることを確認します。また、"Commands to ignore at the beginning of line"ルールを修正する必要がある場合もあります。

しかし、コンパイラ起動コマンドが検出され、ソースファイルがまだ見つからない場合は、起動コマンドの接尾語を"Source file suffixes"ルールと比較してください。

コンパイラオプションを識別する

ソースファイルが適切に識別されたら、コンパイラオプションフラグの抽出箇所をレビューします。

このとき、Compiler Details ログを調べると特に有効です。メイクログの実際の行を、そこから派生したオプションと比較することができます。オプションが欠落していたり、オプションの形式が望み通りでない場合には、Compiler Options ルールを変更します。

また、Warning ログも役に立つことがあります。コンパイラ行に"unrecognized arg"のリストが示されている場合、オプションの形式を変更する必要があるか、いくつかのオプションが"Compiler options taking argument"ルールに登録されていない可能性があります。

パスを調整する

最後の手順として、パスのレビューと解決を行います。

ソースファイルの名前と、それに関連付けられた-I、-D、-U オプションを特定する以外に、抽出処理ではディレクトリの位置が追跡されるため、ソースファイルとインクルードディレクトリの両方に対する相対パス名を解決できます。

CD ログからは、メイクログ内のどの行が、それ以降の相対ディレクトリの決定に使用されるベースを変更するものとして特定されるかを探ることができます。cd を起動する行がない場合には、"Change directory command"ルールを見直してください。cd 起動行に相対パスが含まれている場合、"Processing change directory commands with relative paths"ルールを選択して、適切なパスが決定されるようにします。

最後に、Imagix 4D ソースアナライザが存在しないソースファイルの解析を試みないようにするため、"Include files only if currently found in file system"ルールを有効にする必要があります。ルールを変更するときは、変更によって特定されるソースファイルの数が変化するかどうかに注意してください。変化する場合は、Warnings ログを参照して、ファイルシステム内に見つからないファイルがどれであるかを判定します。そして、この原因がファイルが実際になくなっているためであるか、ファイルパスが処理の際に正しく解決されていなかったためであるかを診断する必要があります。パスの解決に問題がある場合、ディレクトリ変更ルールを修正する必要があります。

パスに関して、もう 1 つ注意すべき点があります。状況によっては、Imagix 4D によって解析するために、ソースコードがメイクログの生成時とは異なる場所に移されている場合があります。この現象は、解析時とコンパイル時とで異なるプラットフォームを使用しているなど、様々な理由で発生します。メイクログに絶対パス名ではなく相対パス名が含まれている場合は、コンパイル環境からメイクログを解析コードへの相対パスと同じ場所にコピーして、処理対象のメイクログとして新しいコピーを指定しても問題ないかもしれません。あるいは、メイクログに絶対パスが含まれている場合は、"Adjust paths using vtgSys(Automount) substitution"ルールを使用できます。設定方法に関してさらに詳しくは、../imagix/user/sample.4Ddefaults を参照してください。

5j. 解析プロセスをスタートさせる

結果のレビューとルールの修正は反復的処理です。ルールの変更(手順 5g)およびメイクログの処理(手順 5h)を行うたびに、新しい結果を生成してレビューすることになります。初めは、明らかにルールを再び改良してメイクログを(再)処理する必要があることが、レビュー(手順 5i)によって十分に裏付けられます。

ある時点で、処理ルールが十分に改良されているようだと判断し、結果となるダイアログベースの(C/C++)データソース定義がどれくらい適切にソースコードを解析するかを確認したいと考えるでしょう。これは取り消し不能の手順ではありません。後に 5g あるいは 5a にまで戻って、手順 5 の繰り返し処理を続行することも可能です。

生成されたダイアログベースの(C/C++)データソース定義を適用してコードを解析できる状態になったら、ダイアログ下部の Load Data Sources をクリックします。

コードをロードする——ターゲットをメイクファイルに追加して解析を行う場合

5a. 新しいデータソースの追加を指定する

Data Sources ダイアログを開きます(メニュー[Project] > [Data Sources])。既存のデータソースの設定を修正するのではなく、プロジェクトに新しいデータソースを追加することを指定するには、ダイアログの左側の Data Sources の下にある[+new data source]を選択します。これは新しいプロジェクトであり、既存のデータを修正しないため、選択可能なオプションはこれだけです。

5b. ターゲットのメイクファイルへの追加を選択する

以降のステップは、Data Sources ダイアログの右側での作業になります。上部にある、Select Data Source Type というラベルのメニューボタンから、[Source Files] > [Make Based] > [Add Targets into Makefile]を選択します。

5c. メイクファイルの名前を指定する

Makefile タブの Makefile フィールドに、メイクファイルの完全パス/ファイル名を入力します。

注:Imagix 4D は指定されたメイクファイルを更新しますが、この段階では、実際のメイクファイルを更新したくないかもしれません。その場合には、imagix.mak という名前をつけて、正規のメイクファイルと同じディレクトリに新しいファイルを生成します。imagix.mak の最初の行を次のように編集してください:

```
include /path/name/of/real_makefile
```

#include ではなく、include を使用します。その上で、Makefile フィールドに imagix.mak の完全パス/ファイル名を入力します。

5d. Imagix 4D がメイクファイルに追加するターゲットのベース名を指定する

通常、Make Target フィールドは imagix のままにしておいてもかまいませんが、メイクファイルにすでに imagix という名前のターゲットが存在する場合には、このフィールドを未使用の名前に変更する必要があります。

5e. 標準的なコンパイル環境を指定する

使用するコンパイラ及びビルドのターゲットプラットフォームを指定します。Compiler&Target コンボボックスに該当するコンパイラとターゲットの組み合わせがある場合には、それを選択します。まだない場合には、3a に戻り、適切なコンパイラ設定ファイルを生成することをおすすめします。あるいは、前記のコン

ボボックスで other を選択します。other を選択した場合には、メイクファイルに暗黙的なオプションを指定する必要があります(IMAGIX_FLAGS)。

5f. メイクファイルを開いて編集する

Edit makefile にチェックがはいった状態で、ダイアログの Add Data Source ボタンをクリックします。Imagix 4D が指定されたメイクファイルにデータ収集ターゲットを追加し、そのメイクファイルが開いて編集できる状態になります。

5g.Imagix 4D 固有のターゲットを修正する

メイクファイルには、5 つの固有のメイクファイルマクロが定義されているセクションがあります。ベースターゲット名を imagix と指定している場合には、これらメイクファイルマクロの最初の 3 つは IMAGIX_SRCDIR、IMAGIX_MAKEFILE、IMAGIX_PROJDIR です。これらの定義は無視してかまいません。Imagix 4D によって自動的に生成されるもので、メイクファイルをほかの場所に複製するためにあります（7 参照）。

ターゲットを imagix と指定している場合、あとの 2 つは IMAGIX_SOURCES、IMAGIX_FLAGS です。これらは修正する必要があります。

IMAGIX_SOURCES

IMAGIX_SOURCES マクロを、読み込みたいソースファイルのリストに展開するように修正します。

正規のメイクファイルでソースのリストを検索します。これは、SRCS=foo1.c foo2.c foo3.c のようにソースファイルのリストで表示されることもあれば、OBJS=foo1.o foo2.o foo3.o のようにオブジェクトファイルのリストで表示されることもあります。

ソースファイルのリストで表示された場合には、ソースをリストするマクロと同等になるように IMAGIX_SOURCES を定義します。たとえば、ソースが上記のように定義されていれば、IMAGIX_SOURCES の行を IMAGIX_SOURCES=$(SRCS)と修正します。

オブジェクトファイルのリストで表示された場合には、IMAGIX_SOURCES を関連するソースファイルまで評価するように修正します。たとえば、オブジェクトファイルが OBJS という名前で、その関連するソースファイルが.cc という拡張子を使用しているとすると、IMAGIX_SOURCES の行を IMAGIX_SOURCES=$(OBJS:.o=.cc)と変更します。

IMAGIX_FLAGS

IMAGIX_FLAGS マクロをコンパイラに明示的に渡される-I、-D、-U オプションのすべてに展開するように修正します。

正規のメイクファイルで-I、-D、-U オプションが定義されているところを検索します。通常、-I オプションは-D オプションと異なったマクロで収集されるでしょう。たとえば、-I オプションはすべて INCLUDES というマクロに含め、-D オプションはすべて CCFLAGS というマクロに含めていたりするでしょう（-U オプションを使用することはまれです）。

IMAGIX_FLAGS の行は、処理するコンパイラ設定ファイルの指定にも利用され、ほかのオプションを Imagix 4D アナライザに渡す目的でも利用することができます。

上記の例のようなマクロが存在するとすれば、IMAGIX_FLAGS の行は IMAGIX_FLAGS=-inc/コンパイラ/設定/ファイルの/パス名/file.inc$(INCLUDES)$(CCFLAGS)のように修正します。コンパイラ設定ファイルの選択を Other とした場合には（5e 参照）、ここで-inc のオプションは表示されません。この行でもコンパイラの暗黙的なオプションを指定する必要があります（3a 参照）。

5h. 解析プロセスをスタートさせる

これでアナライザを起動する準備ができました。ただし、コードを解析しようとすると、-I、-D、-U オプションの指定の仕方にエラーがあったことがわかるのがふつうです。最初はコードのひとつのサブセットだけをロードするようにしたほうがよいでしょう。その場合には、IMAGIX_SOURCES の行を(#をつけて)コメントで無効にし、IMAGIX_SOURCES=foo1.c のような新しい行を生成します。

コードを解析する準備ができたら、メイクファイルへの変更を保存します(LOCAL メニュー[File]> [Save])。メイクファイルを閉じ、情報ウィンドウを閉じます。解析プロセスがスタートします。

コードをロードする——Microsoft Visual Studio を通して解析を行う場合

5a. 新しいデータソースの追加を指定する

Data Sources ダイアログを開きます(メニュー[Project] > [Data Sources])。既存のデータソースの設定を修正するのではなく、プロジェクトに新しいデータソースを追加することを指定するには、ダイアログの左側の Data Sources の下にある[+ new data source]を選択します。これは新しいプロジェクトであり、既存のデータを修正しないため、選択可能なオプションはこれだけです。

5b. MSVC プロジェクトまたは MSVC ソリューション用の解析方法を選択する

以降のステップは、Data Sources ダイアログの右側での作業になります。上部にある、Select Data Source Type というラベルのメニューボタンから、[Source Files] > [Microsoft Visual Studio]の選択肢のどれかを選択します。特定のプロジェクトに関連付けられたファイルをロードするには、MSVC Project を使用します。ソリューション全体(旧バージョンの Visual Studio ではワークスペース)をロードするには、MSVC Solution / Workspace を使用します。

5c. 使用する MSVC のバージョンを指定する

Type フィールドの右側で、使用している MSVC のバージョンを指定します。異なるバージョンの MSVC では異なるファイルタイプを使用してプロジェクトの情報が格納されるため、バージョンを選択すると、ダイアログの他の部分も変更されます。

5d. MSVC プロジェクトファイルまたはソリューションファイルの場所を指定する

Imagix 4D では、コードに関する情報によって、特定のファイルタイプが処理されます。ファイルタイプは、Type と MSVC Version の選択によって異なります。たとえば、MSVC Project と Visual Studio .NET を選択した場合、関連付けられるファイルタイプは.vcproj ファイルになります。表示される.vcproj File フィールドに、MSVC で生成されたプロジェクトの.vcproj ファイルの完全パス名を入力します。通常このファイルは、*project*.vcproj という名前で最上位の MSVC プロジェクトディレクトリにあります。

MSVC Project とメイクファイル(Windows 環境下でのみ利用可能)を選択した場合は、Makefile フィールドに、プロジェクト用に MSVC で生成されたメイクファイルの完全パス名を入力します。通常このファイルは、*project*.mak という名前で最上位の MSVC プロジェクトディレクトリにあります。*project* は MSVC プロジェクトの名前になります。

5e. プリコンパイル済みヘッダを含むディレクトリを指定する

プリコンパイル済みヘッダを使用している場合には、コンパイルを始めるオリジナルのヘッダファイルを含むインクルードディレクトリに関する情報が、MSVC プロジェクト設定で使用不可になっていることがあります。この場合には、Imagix 4D アナライザにヘッダファイルの場所がわかるように、これらのディレクトリを明示的に指定する必要があります。

Options フィールドに、-I（または-S）構文-Idirname を使って、ヘッダファイルを含む各ディレクトリのディレクトリ名を入力します。-I は必要な数だけ入力してかまいません。-I とあとに続くディレクトリ名との間には、スペースを入れないでください。ディレクトリ名と次の-I との間はスペースで区切ってください。

5f. 調べたい MSVC プロジェクトの設定を指定する

Configuration コンボボックスで、ビルドしたい Visual Studio プロジェクトの設定を選択します。この選択を省略（デフォルトのままに）すると、いつでも Imagix 4D プロジェクトのデータが再生されるときには、MSVC プロジェクトの現在の設定が解析されます。ソリューションでは現在の設定が常に使用されるため、このコンボボックスは利用できません。

5g. 標準的なコンパイル環境を指定する

使用するコンパイラ及びビルドのターゲットプラットフォームを指定します。Compiler & Target コンボボックスに該当するコンパイラとターゲットの組み合わせがある場合には、それを選択します（3a 参照）。通常は msvc_win を選択します。

5h. 解析プロセスをスタートさせる

コードを解析する準備ができたら、ダイアログの Add Data Source ボタンをクリックします。

コードをロードする——ダイアログを通して解析を行う場合(Java)

5a. 新しいデータソースの追加を指定する

Data Sources ダイアログを開きます（メニュー［Project］> ［Data Sources］）。既存のデータソースの設定を修正するのではなく、プロジェクトに新しいデータソースを追加することを指定するには、ダイアログの左側の Data Sources の下にある［+ new data source］を選択します。これは新しいプロジェクトであり、既存のデータを修正しないため、選択可能なオプションはこれだけです。

5b. ダイアログを通した解析方法を選択する

以降のステップは、Data Sources ダイアログの右側での作業になります。上部にある、Select Data Source Type というラベルのメニューボタンから、[Source Files] > [Dialog Based (Java)]を選択します。

5c. 解析対象のソースファイルを指定する

Source Files タブにおいて Directory フィールドを使用し、ソースコードがあるディレクトリの名前を入力します。

注：ソースコードが複数ディレクトリに広がっている場合、Directory フィールドのすぐ下にある Analyze source files in subdirectories を利用することができます。解析したいコードが 1 つのディレクトリとそのサブディレクトリ（およびそのサブディレクトリ）に広がっている場合、最上位ディレクトリの名前をこの Directory フィールドに入力します。省略したいサブディレクトリがある場合は、それらを Exclude ダイアログに指定します。ソースコードがどのようにディレクトリ構造のなかに分布しているかによって、解析したいコードを含むディレクトリごとに手順 5 を繰り返す必要がある可能性があります。

Source Files フィールドに、ソースファイルの名前を入力します。* の文字のあるパターンは拡張され、例えば、*.java とするとディレクトリ内のすべての.java ファイルが一覧表示されます。複数の名前やパターンは、空白で区切って入力することができます。.class ファイルを指定する必要はありません。アナライザは、指定した.java ファイルのどれかによってインポートされる.class ファイルを自動的に解析します。

5d. Java 言語環境を指定する

さらに Source Files タブにおいて、ソースコードの Java 言語環境を指定します。ご使用の言語環境が Language コンボボックスにリストされていれば、それを選択します。other を選択すると、Class Paths タブにある.jar Files の領域の Import Settings に、明示的に –cp オプションを指定する必要があります（手順 5e）。

5e. インポート対象のクラスファイルおよび jar ファイルの場所を指定する

5c に記述したように、通常は明示的に Imagix 4D に対してクラスファイルを解析するように指示する必要はありません。代わりに、ソースコードのインポート文を使用して、インクルードされたクラスファイルを探すべきクラスパスを単に指定します。これは Class Paths タブで指定されます。

タブにある[Import Settings for .class Files] の部分で、クラスファイルのインポート方法を設定します。インポート指令が解析対象の.java ファイルに存在する場合、ここに定義されたクラスパスにより、.class ファイルを検索するディレクトリが制御されます。複数のクラスパスを指定する必要がある場合は、Additional -cp flags フィールドを使用できます。-cp は必要な数だけ入力してかまいません。-cp とディレクトリ名との間に、スペースを入れます。ディレクトリ名と次の-cp との間にもスペースを入れてください。Windows 環境で稼動している場合、ディレクトリ名に空白が含まれるときは *dirname* を二重引用符で囲み、`-cp "c:/program files/ include"' のようにします。

タブにある[Import Settings for .jar Files] の部分で、.jar ファイルのインポートを制御できます。これは言語設定ファイル（手順 3a を参照）に記述されていない.jar ファイルのために使用されます。Directory フィールドおよび Jar Files フィールドの設定に合致する.jar ファイルがインポートされます。

5f. 解析プロセスをスタートさせる

これでアナライザを起動する準備ができました。しかしながら、コードを解析すると設定でのエラー、とくにクラスパスの指定方法についてのエラーが発生することが多くあります。まず、コードの一部のみをロードする方が望ましいかもしれません。その場合は、Source Files タブの Source Files フィールドに移動し、最初の処理で解析する特定のファイルを 1 つだけ指定します。

コードを解析する準備ができたら、ダイアログの下部にある Add Data Source ボタンをクリックします。

アナライザの設定を微調整する(C/C++)

6a. 解析結果をレビューする

解析が完了すると、Analysis Results ウィンドウが表示されます。このウィンドウの情報を使用して、解析の設定を微調整します。

6b. ファイル名指定上の問題を修正する

メイクファイルのターゲットを追加する方法を使用した場合、Analysis Results ウィンドウは make によって生成されたコマンドをエコーするところから始まります。ウィンドウに Analyzing foo.c 型のコメントが含まれていなければ、問題が発生しています。ソースファイルが指定されていない場合がよくあります。通常のエディタを使用して、メイクファイルの IMAGIX_SOURCES の定義を確認します。

6c. 欠落している-I、-D、または-U フラグを確認する

次に、どの解析方法を選択した場合にも、Analysis Results ウィンドウには、解析されたファイルが（Analyzing foo.c のように）次々と表示され、その下にファイルの解析中に発生した問題に関するメッセ

ージのリストが表示されます。これらのアナライザのメッセージは、コードの解析に使用する-I、-D、-Uオプションを修正することによって訂正することができます。

たとえば、cannot open fileというアナライザのメッセージが表示された場合には、そのインクルードファイルを含むディレクトリが指定されていなかった可能性があります。問題のインクルードファイルが実際にはどこに置かれていて、どのディレクトリを-Iスイッチで含めなければならないかを確かめてください。

アナライザがそのインクルードファイルを見つけようとしても、まだ unknown type foo near symbol bar のようなエラーメッセージが多数出る場合には、-D、-Uオプションの指定が不完全または不正確だった可能性があります。

この場合には、-D、-Uオプションの指定の仕方に問題があって必要な型が不明のために、ヘッダファイルのセクションでエラーが発生した可能性があります。まず、Analysis Resultsウィンドウの該当するメッセージからファイルを開き(マウス、左ダブルクリック)、解析上の問題がレポートされたコードを調べてください。必要な型で、不明なものがないかどうかを調べます(この作業には、Symbol Indexタブまたは Database Lookupタブを利用すると便利です)。不明なものがあれば、どこで型を定義すべきだったかを確かめます。この作業には、Grep Toolタブを利用することができます。目的のコードを解析するには、どのようなマクロ定義のセットが必要だったかを確かめます。

あるいは Analysis Resultsウィンドウで自動的にデータの多くを収集することもできます。Analysis Resultsウィンドウで特定のエラーメッセージの行を選択します(マウス、左クリック)。ここでエラー解析ビューに切り換えます(LOCALメニュー[Display] > [Show Error Analysis])。上のパラグラフで説明したようなデータの多くが自動的に生成されます。マクロ定義をどう変更すべきかは、なお判断する必要があります。

注:Imagix 4Dのアナライザは、字句解析プログラムでは得られないようなエラーメッセージを生成します。Imagix 4Dは、ソースファイルからより完全で正確なデータを抽出するために、コンパイラのように意味解析を行うからです。ただし、コンパイラとは異なり、Imagix 4Dのアナライザにはビルトインのエラー訂正機能があります。解析上の問題に遭遇したら、Imagix 4Dはソースコードに再同期し、再同期中にはどの行をスキップしなければならなかったかをレポートします。このため、解析結果をレビューして解析上の問題の重要性を判断するときには、何行が無視されたかも考慮に入れてください。

6d. -I、-D、-U オプションを修正する

コンパイラ設定ファイルを利用していて、欠落しているインクルードディレクトリがシステムヘッダファイルのディレクトリの場合、あるいは欠落している-Dがシステムヘッダファイルに適合する場合には、コンパイラ設定ファイルを適切に修正します。

それ以外の場合には、追加の-I、-Dオプションを他のプリプロセッサオプションのリストに追加します。ターゲットをメイクファイルに追加する方法を採る場合は、メイクファイル自体を処理して、そこに定義されている IMAGIX_FLAGSを変更する必要があります。

他の方法を採る場合は、Data Sourcesダイアログで操作を続けます。必要である場合は、メニュー[Project} > [Data Sources]からダイアログを再度開きます。ダイアログを通した解析では、Source Filesタブまたは Include Dirsタブで必要な変更を行ってください。

メイクログからの抽出を通した解析では、問題がメイクログ自体によるものかメイクログの処理によるものかを判断する必要があります。新しいメイクログを生成するか処理ルールを変更しても問題が解決しない場合、2つの回避策を検討してください。第1の回避策は、Makelogタブの Optionsフィールドにオプションを追加することです。この場合、オプションが次に生成される際に、ダイアログベースの(C/C++)データソースのそれぞれに伝播されます。第2の回避策は、既に生成されているダイアログベースの(C/C++)データソースの一部だけを変更することです。一般に、第2の回避策は推奨されません。特定のダイアログベースの(C/C++)データソースに対する変更は、次にメイクログを処理するときに失われま

す(手順 5h)。メイクログの処理は、メイクログの変更または再生成、処理ルール変更、または Options フィールドへのオプションの追加を行ったときに必要になります。

6e. ソースファイルのセット全体を指定する

意図的にファイルのひとつのサブセットだけを解析しながらプリプロセッサオプションを調整してきて、エラーメッセージがほとんど表示されなくなるまで-I、-D、-U オプションの調整ができたときには、ファイル定義を解析したいすべてのファイルを反映するように変更します。メイクファイルを通して解析を行う場合には、そのメイクファイルの IMAGIX_FLAGS を再定義します。ダイアログを通して解析を行っている場合には、元の Data Sources ダイアログを呼び出し(メニュー［Project］> ［Data Sources］)、Source Files タブの Files フィールドで変更を指定します。

6f. ソースファイルを再解析する

コンパイラ設定ファイル、メイクファイル、及び/または Data Sources ダイアログを適切に修正したら、コードを再解析します。コンパイラ設定ファイルまたはメイクファイルの変更は、メニュー［Project］> ［Regenerate Project Data］を利用して行います。

6g. 調整を繰り返す

6a に戻り、解析結果に満足がいくまで以上の調整プロセスを繰り返します。

アナライザの設定を微調整する(Java)

6a. 解析結果をレビューする

解析が完了すると、Analysis Results ウィンドウが表示されます。このウィンドウの情報を使用して、解析の設定を微調整します。

6b. 欠落している–cp オプションを確認する

次に、どの解析方法を選択した場合にも、Analysis Results ウィンドウには、解析された java ファイルが(Analyzing foo.java のように) 次々と表示され、その下に java ファイルの解析処理の一部としてインポートされたクラスファイルの一覧、そしてそのファイルの解析中に発生した問題に関するメッセージのリストが表示されます。これらの解析メッセージは、コードの解析に使用する-cp オプションを修正することによって訂正することができます。

たとえば、could not locate import for という解析メッセージが表示された場合には、クラスファイルのクラスパスが指定されていなかった可能性があります。クラスが実際にある場所を特定して、-cp オプションで含めるべきディレクトリを判定してください。

6c. クラスパスの設定を変更する

言語設定ファイルによる解析方法を使用しており、ご使用の標準 Java 環境のための.jar ファイルが見つからない場合は、言語設定ファイルを適切に修正します。

あるいは、メニュー［Project］> ［Data Sources］を使用して元の Data Sources ダイアログを呼び出し、Source Files タブまたは Class Paths タブに必要な変更を加えます。

6d. ソースファイルのセット全体を指定する

意図的にファイルのひとつのサブセットだけを解析しながらプリプロセッサオプションを調整してきて、エラーメッセージがほとんど表示されなくなるまでクラスパスの設定を調整できたときには、ファイル定義を

解析したいすべてのファイルを反映するように変更します。メニュー［Project］> ［Data Sources］で元の Data Sources ダイアログを呼び出し、Source Files タブの Files フィールドで変更を指定します。

6e. ソースファイルを再解析する

言語設定ファイルおよび Data Sources ダイアログを適切に修正したら、コードを再解析します。言語設定ファイルの変更は、メニュー［Project］> ［Regenerate Project Data］を利用して行います。

6f. 調整を繰り返す

6a に戻り、解析結果に満足がいくまで以上の調整プロセスを繰り返します。

大規模、ワイド展開プロジェクトのローディング

ここでは、大規模なプロジェクト、または 多数のディレクトリに展開しているプロジェクトを扱う場合に一般的に考慮すべき点を示します。

7a. 大規模なプロジェクトはより小規模なプロジェクトに分割する

プロジェクトの規模を制限する理由は、ふたつあります。プロジェクトの規模が大きくなればなるほど、Imagix 4D のパフォーマンスは低下します。さらに重要なのは、解析するプロジェクトの規模が大きくなればなるほど、関心のないデータを解析するためによけいな労力をとられることです。このため、できれば、大規模なプロジェクトはサブシステムごとに個別のプロジェクトを生成し、より小規模なプロジェクトに分割してください。

それでも、ときにはより広い視点でシステムを見てみたいことがあります。サブシステムの調査に必要な個々のプロジェクトを作成したら、ソフトウェアのより大きい部分を解析するために、個々のプロジェクトをひとつにまとめることができます。この処理は、Open Combination of Projects ダイアログ (メニュー[File] > [Open Project...] > [Combine>>])から実行できます。このダイアログでは、コンボプロジェクトとして同時にロードする個々のプロジェクトをまとめて指定できます。

あるプロジェクトの組み合わせを後に再び呼び出したくなることが想定される場合、それをコンボプロジェクトとして保存しておけば、通常のプロジェクトと同様に Open Project ダイアログを通してそれを開くことができます。さらにこの作業を繰り返すことにより、コンボプロジェクトのコンボプロジェクトを生成することもできます。(詳しくは、ユーザガイドの「コンボプロジェクト」の章を参照してください。)

7b.プロジェクトを複製する（メイクファイルを通して解析を行う場合）

ソースコードが多数のディレクトリにまたがって展開している場合には、個々のディレクトリごとに別々のプロジェクトを生成します（ファイルは通常、ディレクトリごとに関連付けられてまとまっているので、個々のディレクトリが特定のサブシステムを保持することになります）。

プロジェクトを複製するには、まず 1～6 の要領でひとつのディレクトリにそのプロジェクトをセットアップします。ただし、imagix.mak を通して解析する方法をとってください（5b 参照）。

次に、ソースコードの各ディレクトリに imagix.mak ファイルをコピーします。1 行目（include/path/name/of/real/makefile）を新しいディレクトリの正しい実メイクファイルを指すように修正します。IMAGIX_SRCDIR、IMAGIX_MAKEFILE、IMAGIX_PROJDIR の定義を新しいディレクトリの場所を指すように修正します。

メイクファイルが一貫したものであれば、必要な修正作業は以上ですべてです。個々のメイクファイルが異なったメイクマクロ名を使用している場合は、IMAGIX_SOURCES と IMAGIX_FLAGS を修正する必要があります。

カレントディレクトリで imagix.mak に対する修正を保存してから make -f imagix.mak imagix_proj を起動します。これでこのプロジェクトはセットアップされます。すぐにコードを解析したい場合には、make -f imagix.mak imagix を起動します。そうでない場合には、コードは最初にこのプロジェクトを開いたときに解析されます。

個別のディレクトリまたはプロジェクトをセットアップしたら、コンボプロジェクトによって個別のプロジェクトを結合し、プロジェクトの規模を大きくします（7a 参照）。

プロジェクトとデータ収集

Imagix 4D は、みなさんのソフトウェアのさまざまな側面を物語る数多くのソフトウェア成果物をその情報源として利用します。これらのデータソースの大半は、すでにみなさんの環境に存在します。たとえば、Imagix 4D は最も重要な成果物、つまりソースコード自体から直接、クラス、関数、変数、及び型の情報を抽出することができます。また、メイクファイルを解析し、ビルド依存関係に関する情報を生成することもできます。

ソフトウェアの他の側面を調べるには、明示的にデータソースを生成した上で、Imagix 4D を用いてそこからデータを収集する必要があります。通常、データを生成するために使うツールはコンパイラです。たとえば、Unix 環境で生成されたテストカバレッジの結果を解析するため、Imagix 4D は.d ファイルから情報を抽出します。これらは、コンパイラの-(x)a オプションを設定し、コードをコンパイルした上で、実行ファイルを実行することによって生成されます。

みなさんがどのデータソースをインポートするかを選択することにより、ソフトウェアについて収集される情報の種類と範囲を制御します。データソースからの情報は、Imagix 4D のデータベースに統合されます。その結果として収集された情報はプロジェクトと呼ばれ、ファイルシステムに格納されます。通常は数多くのプロジェクトを生成し、それらを個別に利用して、ソフトウェアのさまざまな部分を調べます。

データをインポートするプロセスは、次の 3 段階から成ります。

1. 追加したい情報を含むデータソースを選択します。
2. そのデータソースがまだ存在しない場合には、それを生成します。
3. そのデータソースをプロジェクトに追加します。

データをインポートしたら、それをソースコードに同期させておく方法はいろいろあります。

プロジェクト

Imagix 4D では、プロジェクトという言葉は、調査しようとしているソフトウェアの特定の部分について保持するあらゆる情報のリポジトリの意味で使用します。データをインポートしてビューするとき、Imagix 4D はデータソースから情報を抽出し、それをビュー用にロードします。この情報の大半は、カレントプロジェクトにパーマネントに格納されます。

プロジェクトに関するパーマネントな情報はすべて、ファイルシステムのディレクトリ構造に格納されます。その点では、プロジェクトは共用リソースです。通常、ファイルの読み書きのパーミッションは、プロジェクトのディレクトリ及びファイルに適用されます。誰がプロジェクトをアクセスすることができるか、また更新することができるかは、標準的なパーミッションメカニズムを通して制御することができます。Imagix 4D プロジェクトの最上位ディレクトリは拡張子.4D を使用するので、これで識別することができます。

Imagix 4D の使用が増加すると、どのデータがどのプロジェクトに存在するかを管理することが、このツールを最大限に活用する上で重要であることにお気づきになるでしょう。データが多ければ多いほど、プロジェクトのなかでより広範囲の情報を解析することができます。ただし、関心のある特定の情報を見つけるには、より多くの情報をフィルタにかける必要があります。サイズが大きいデータベースの場合は、いくらか応答時間が長くなります。

コンボプロジェクト

Open Combination of Projects ダイアログ（メニュー [File] > [Open Project...] > [Combine>>] ）を用いると、重複するデータを個々のプロジェクトに格納せずに、個別のプロジェクトをまとめてコンボプロジェクトに結合することができます。ソフトウェアのサブシステムごとに別々のプロジェクトを生成したのち、それらを素早く結合させて、ソフトウェアのより大きな部分をカバーするプロジェクトにすることができます。

ダイアログでは最初に、結合するプロジェクト群を指定します。そして、単にそれを開くよう指定します。Imagix 4D はプロジェクト群を単一の一時プロジェクトに結合し、それを開きます。その結果は、選択されたすべてのプロジェクトにまたがる単一のプロジェクトを作成して、開いたときと同様になります。

または、あるプロジェクトの組み合わせを将来再び解析したくなることが想定される場合、それを開くかわりに保存するよう選択できます。その結果、選択されたすべてのプロジェクトにまたがる、永続的な新しいプロジェクトが作成されます。新しいコンボプロジェクトでは、個々のプロジェクトに格納されるデータの重複が避けられますが、通常のプロジェクトとまったく同様に動作します。例えば、コンボプロジェクトは通常のプロジェクトと同様に、通常の Open Project ダイアログ(メニュー [File] > [Open Project])から選択できます。

コンボプロジェクトは、プロジェクトの管理をしやすくするきわめて有用なツールであることがおわかりになるでしょう。生成されるときにデータが複製されないので、コンボプロジェクトは生成に要する時間が短く、余計なディスク容量もとりません。また、個々のプロジェクトについて生成されるデータをポイントするので、個々のプロジェクトのデータが更新されれば、コンボプロジェクトも自動的に更新されたことになり、その逆の場合も同じことがいえます。コンボプロジェクトのコンボプロジェクトを生成することも可能です。

複数のデータソースを 1 つのプロジェクトの中で使用するケースよりも、プロジェクトをコンボプロジェクトにまとめるケースの方が多くみられます。Data Sources ダイアログ(メニュー [Project] > [Data Sources])では、ソフトウェアのそれぞれ異なる部分を、個々のプロジェクトに含めるように指定できます。その後、Imagix 4D は指定されたソフトウェアのすべてを解析する必要がありますが、そのプロジェクトは分割できない状態のままです。これに対し、コンボプロジェクトが定義される際には新しいデータは一切収集されません。既存の個々のプロジェクトからのデータが単に結合されます。

データソース

Imagix 4D は、ソフトウェアのさまざまな側面とレポートを表示する多様なディスプレイウィンドウを提供します。これらのディスプレイに表示される情報は、いくつものデータソースから来ます。Imagix 4D のプロジェクトデータベースにデータをインポートするときには、使用可能な情報の型と範囲を制御します。

データソースには、Imagix のディスプレイで使用される、次の 4 つの基本的なタイプがあります。

ファイルシステム ファイル情報は、ディレクトリ及びファイルの階層に関するデータを含みます。また、所有者、読み書きのパーミッション、更新日など、その属性も含みます。Imagix 4D はこの情報をファイルシステムから自動的に収集します。

ビルド規則 ビルド規則に定義されるビルドターゲット及びビルド依存関係に関する情報は、明示的に指定するメイクファイルから直接収集されます。

プログラムの要素 Imagix 4D のディスプレイの大半は、品質メトリックスなど、プログラムの要素の属性を説明する情報を含め、プログラムの要素及びそれらの相互関係に焦点をあてています。Imagix 4D は、ソースファイルそのものを解析することによってこのデータを生成します。

プロファイルデータ 実行ファイルの実行時動作を決定するデータソースは、コンパイラにオプションを設定した上で実行ファイルを実行することによって生成されます。これらのデータソースのロードは、明示的に指定する必要があります。

以上のデータソースのなかで最も重要なのはソースファイルそのものです。どのデータソースに調べたい情報が含まれているかがわかれば、まだ生成されていないデータソースを生成することもできます。

C/C++コードの解析

Imagix 4D は、みなさんのシステムの数多くのソフトウェア成果物から情報を収集することができますが、そのなかで最も重要なのはソースコードファイルそのものです。ソースコードの解析は、Imagix 4D のアナライザによって処理されます。Java のアナライザについては後のセクションで説明します。ここでは C/C++ のコードのアナライザについて説明します。

Imagix 4D の C/C++のアナライザは、コンパイラとよく似た動作をします。コンパイラのように、ソースファイルの完全な意味解析を行い、コード中のすべてのシンボルを解析します。ただし、コンパイラはリンカによって実行ファイルに組み込まれるオブジェクトファイルを生成するのに対し、Imagix 4D のアナライザは Imagix 4D のデータベースにロードされ、ほかのデータファイルと統合されるデータファイルを生成します。

アナライザはコンパイラから独立していて、C/C++コンパイラ用に開発されたソースコードを正確に解析することができます。アナライザはコンパイル設定ファイルを通して、どのようなものであれ、通常、みなさんのソフトウェアにお使いのコンパイラの動作をエミュレートすることができます。

Imagix 4D のアナライザは、Imagix 4D のデータベース/ユーザインターフェイスから独立した実行ファイルです。これは Imagix 4D のユーザインターフェイスから起動するか、または個別に（メイクファイルなどの）コマンドラインから起動できます。

アナライザの構文とオプション

Imagix 4D の C/C++のアナライザは、Imagix 4D のデータベース及びユーザインターフェイスから独立した実行ファイルです。/imagix/bin のディレクトリに置かれていて、次の構文を使用します。

imagix-csrc [*option*...] *file*...

ただし、option はひとつ以上の imagix-csrc オプション（以下に説明）、file はひとつ以上のソースファイルの完全パス名をさします。通常は、関係があるのは.c および.cpp のソースファイルだけです。解析対象となる *file* のリストで直接ヘッダファイルを指定できますが、通常これらは、解析対象のソースコード内の #include "filename" プリプロセッサディレクティブの結果として解析されます。Imagix 4D のアナライザは以下のオプションをサポートします。

-1prog　すべてのソースコードが同一の実行可能モジュールから供給されているものとして扱います。

ほとんどの Flow Check レポートの基盤となるデータフロー解析では、すべてのソースコードが1つのプログラムから供給されている必要があります。 -1prog オプションにより、アナライザはすべてのソースコードが同一の実行可能モジュールから供給されているかのように処理します。一致しているすべてのグローバル名は、同一のオブジェクトを参照しているものとして考えられます。

オプションは、Data Sources ダイアログの Options タブで解析オプションのひとつとして選択します。

-asm　キーワード asm の使用をサポートします

一部のコンパイラは、C/C++ソースファイル内でのインラインアセンブラのコードをサポートします。これはさまざまな言語構成要素を通して行います（次項「言語拡張」参照）。キーワード asm は通常、そのような言語構成要素の始まりを標識するために使用します。これらのインラインアセンブラ構成要素の一部はアナライザによって自動的に処理されますが、-asm オプションは、問題を避けるため、アナライザに一部の他の構成要素を含むコードの処理を有効にします。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-at キーワード@及び_at_の使用をサポートします

一部のコンパイラは、メモリへの変数の物理プレースメントを指定するために非標準的キーワード@及び_at_の使用をサポートします。この言語拡張は Imagix 4D コンパイラ設定ファイルの通常の#define keyword 指令ではサポートすることができません。-at オプションは、問題を避けるため、アナライザに@及び_at_キーワードを使用するコードの処理を有効にします。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-cdyn Dynamic C 言語の使用をサポートします。

Dynamic C 言語は C 言語を正規に修正したもので、Rabbit ハードウェア用 Digi International / Rabbit / Z-World クロスコンパイラでサポートされています。-cdyn オプションはアナライザに Dynamic C 構文と意味を認識させ、サポートさせます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-cfrontpt Cfront にパラメータ化されたテンプレート命令の使用をサポートします。

Cfront コンパイラは特殊指令 PT_names、PT_define、及び PT_end を一種の初期の C++テンプレートとしてサポートしました。-cfrontpt オプションはアナライザにこれらの指令を認識させ、サポートさせます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-cmmd*Filename* *Filename* からコマンドを追加します。

imagix-csrc ソースアナライザはコマンドライン実行ファイルです。-cmmd オプションを使用することで、コマンドラインオプションを間接的に指定することができます。*Filename* で指定されたオプションは、コマンドライン自体で指定したオプションに追加されます。*Filename* が複数行で構成されている場合、imagix-csrc は *Filename* 内の次の行から、オプションと合わせて複数回起動されたように動作します。

-cpascii マルチバイトのソースファイルを ASCII コードページとして解析します。

-cpeuc マルチバイトのソースファイルを EUC コードページとして解析します。

-cpsjis マルチバイトのソースファイルを SJIS コードページとして解析します。

-cputf8 マルチバイトのソースファイルを UTF-8 コードページとして解析します。

imagix-csrc ソースアナライザのデフォルトの動作では、汎用のマルチバイト・コードページがサポートされます。これはほとんどのマルチバイトのソースファイルで機能します。汎用的にサポートされない特異なファイルについては、これらの-cpxxx オプションを使って imagix-csrc が特定のコードページの規格をサポートするように指定できます。-cpxxx オプションは 1 つだけ使用してください。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加するか、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-cwd*Dirname* 解析用のカレントディレクトリを *Dirname* に設定します。

imagix-csrc ソースアナライザは-I、-S、-H、および-R オプションと同様に、ソースファイルに対する相対パスを受け入れます。これらのパスはカレントディレクトリとの相対パスであり、デフォルトではソースアナライザが起動される場所になります。-cwd オプションにより、相対パスの開始ディレクトリが変更されます。

オプションはメイクファイルとともに使用するか、ソースアナライザをコマンドラインから起動するその他の方法によってのみ使用すべきです。

-cpp .c ファイルを C++ファイルとして解析します。

通常、拡張子.c を持つファイルは C 言語規則を用いて解析されます。C++拡張子(.C、.cc、.cpp 等)を持つファイルは C++言語規則を用いて解析されます。-cpp オプションは.c ファイルにも C++言語規則を適用させます。

オプションは、Data Sources ダイアログの Options タブで解析オプションのひとつとして選択します。

-D*macroname*[=*value*] マクロ macroname を定義します

これはコンパイラのオプション-D と同じです。マクロ置換は、アナライザによるソースコードの前処理の一部です。マクロが定義されている間、出現した識別子 macroname はすべてトークン value に置き換えられます。value が指定されていない場合には、デフォルト値 1 が使用されます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールド、あるいはメイクファイルで指定します。Windows 版アナライザは/D も受け付けます。マクロはコンパイラ設定ファイルでも定義することができます。

-encmac マクロに関するクロスリファレンスデータを収集します。

imagix-csrc ソースアナライザは、マクロを完全に展開して、マクロに隠蔽されたシンボル使用情報を収集します。 例えば、マクロ展開によってなされた関数呼出しは Function Call Tree ビューに表示されます。 また、関数でのマクロの使用は Function Calls with Macros ビューに表示されます。 しかし、アナライザのデフォルトの動作ではマクロのクロスリファレンス情報は記録されません。-encmac オプションにより、この情報が生成されます。追加情報はハイライト表示され、File Editor のほか、 Function Calls with Macros ビューにも現れます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加するか、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-genflow*Dirname* ディレクトリ *Dirname* のファイルに制御フロー及びデータフローデータを書き込みます。

アナライザは、解析されたソースコードの制御フローとデータフローに関するディスプレイとレポート生成するデータを収集することができます。このフローデータはエンティティ/関係/属性データが格納されているところとは別に、ファイルの集合に保存されます。-genflow オプションはこれらのフローデータファイルの場所を制御します。Imagix 4D がこのデータを検索できるようにするには、genvdb データとディレクトリパスが同じ(-genvdb データと並列)である、cfd というディレクトリ内の場所を指定する必要があります。-genflow オプションが使用されていない場合は、フローデータは生成されず、フローチャート、計算ツリー、制御フローグラフ、及び特定のレポートが作成されません。

オプションは、データソースを追加するときに自動的にセットされます。

-genqcf*Dirname* ディレクトリ *Dirname* のファイルに品質管理データを書き込みます

アナライザは、ソースコードチェックを実行し、解析されたソースファイルのシンボルに関するメトリックスを収集することができます。この品質管理データは、エンティティ/関係/属性データが格納されているとこ

ろとは別に、ファイルの集合に保存されます。-genqcf オプションはこれらの品質管理データファイルの場所を制御します。Imagix 4D がこのデータを検索できるようにするには、genvdb データとディレクトリパスが同じ（-genvdb データと並列）である、qcm というディレクトリ内の場所を指定する必要があります。-genqcf オプションが使用されていない場合は、品質管理データは生成されず、また特定のメトリックスやレポートが作成されません。

オプションは、データソースを追加するときに自動的にセットされます。

-genvdb*Dirname* ディレクトリ *Dirname* のデータファイルに解析結果を書き込みます。

アナライザは明示的に解析されるソースファイル、及び直接的または間接的にそれらのソースファイルに含まれるヘッダファイルの個々についてエンティティ/関係/属性データを生成します。例外として考えられるのは、システムインクルードディレクトリのヘッダファイルです（-nosys オプション及び-S オプションの項参照）。

-genvdb オプションが使用されているときには、個々のソースまたはヘッダファイルに関するデータはディレクトリ Dirname の独立したファイルに格納されます。このため、増分解析が可能になるので、これがおすすめの方法です。ほかには-o オプションを用いる方法があります。

オプションは、データソースを追加するときに自動的にセットされます。

-gnu GNU 言語拡張を有効にします。

GNU gcc 及び g++コンパイラは、標準 C/C++言語へのいくつかのキーワード拡張を受け付けます。これらの拡張の多くは、GNU コンパイラ用コンパイラ設定ファイルの#define keyword 指令を通してサポートされます。ただし、一部の拡張は#define でサポートされず、追加のサポートを必要とします。

そのサポートを提供するのが-gnu オプションです。具体的には、インラインアセンブラコードを区別するためのキーワード__asm の使用、C コードへのキーワード inline の使用、C++コードのあらゆるグローバル識別子への std::の使用がサポートされます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-H*dirname* #include ファイルをカレントディレクトリで検索する前に *dirname* で検索します

アナライザは、#include "filename"前処理指令に出会うと、まず-H 及び-R オプションを通して指定されたすべてのディレクトリで filename を検索します。それで見つからない場合には、カレントディレクトリ（"."）で検索し、それでもまだ見つからない場合には、-I 及び-S オプションで指定されたディレクトリで検索します。#include "filename"ではなく、#include <filename>が使用されている場合には、カレントディレクトリ（"."）はスキップされます。

-H で指定されたディレクトリは、オプションとして指定されている順番で検索されます。これらのディレクトリはアプリケーションインクルードディレクトリとして、-I オプションで指定されたディレクトリと同じように扱われます。これらのディレクトリをシステムインクルードディレクトリとして扱うには、-R オプションを使用します。その場合の影響については、-nosys オプションの項を参照してください。

オプションは、Data Sources ダイアログの Include Dirs タブにある-I、-S Flags フィールド、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。Windows 版アナライザは/H も受け付けます。

-inc*Filename* ファイルごとに解析のスタートに *Filename* をインクルードします

-inc オプションは、解析する各ソースファイルの最初に#include<Filename>の行を加えることに相当します。つまり、ファイル自体を修正せずにその解析の仕方を変更することを可能にします。

このオプションはおもにコンパイラ設定ファイルをインクルードするために使用します。コンパイラ設定ファイルはマクロとインクルードディレクトリの場所を定義し、Imagix4D アナライザに、通常コードのコンパイルに使用するコンパイラをエミュレートすることを可能にします。

Filename は、Data Sources ダイアログの Source Files タブにある Comp/Trgt コンボボックスで ../imagix/user/cc_cfg のファイルから選択します。

-I*dirname* #include ファイルを検索するディレクトリのリストに *dirname* を追加します

アナライザは、#include "filename"前処理指令に出会うと、まず-H 及び-R オプションを通して指定されたすべてのディレクトリで filename を検索します。それで見つからない場合には、カレントディレクトリ(".")で検索し、それでもまだ見つからない場合には、-I 及び-S オプションで指定されたディレクトリで検索します。#include "filename"ではなく、#include <filename>が使用されている場合には、カレントディレクトリ(".")はスキップされます。

Imagix 4D アナライザが#include <filename>と#include "filename"を区別するのは、カレントディレクトリを検索するかどうかを判断するためにすぎません。インクルードディレクトリをシステムインクルードディレクトリとして宣言するには、-Idirname ではなく-Sdirname を使用します。その場合の影響については、-nosys オプションの項を参照してください。

オプションは、Data Sources ダイアログの Include Dirs タブにある-I、-S Flags フィールド、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。Windows 版アナライザは/I も受け付けます。

-kr C 言語の Kernigan & Ritchie ダイアレクトをサポートします。

Imagix 4D アナライザではほとんどの場合、コードが ANSI C またはより古い K&R C ダイアレクトに準拠するかどうかをユーザが指定しなくても、C コードを解析できます。ただし、特定の構成要素では、2 つのダイアレクト間の動作の違いが自動的に解決されないコメントがマクロに含まれています。デフォルトでは、アナライザは ANSI C をサポートしています。オプションを適用すると、K&R 仕様がサポートされます。このオプションは-traditional として起動して、GNU コンパイラとの一貫性を持たせることができます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-lc ファイル参照を小文字に変換します

Unix はファイル名について大文字小文字を区別しますが、Windows はそうではありません。このため、Unix 環境で Windows のコードを解析しようとすると、問題が生じることがあります。たとえば、Windows のコードで、あるファイルには#include <STDIO.H>、別のファイルには#include <stdio.h>の前処理指令が含まれていたとします。これらの指令は、Windows 環境では同じ意味で、どちらの指令でも同じファイル stdio.h がインクルードされます。

しかし、Unix 環境では、これらの指令の意味は違ってきます。STDIO.H または stdio.h のいずれかのファイルは見つからず、したがって、Windows のコードは正しく前処理されません。-lc オプションは、アナライザにすべてのファイル参照を小文字に変換させます。-lc が使用されているときには、アナライザは #include <STDIO.H>の指令でも stdio.h のインクルードディレクトリを検索し、これにより、#include <STDIO.H>と#include<stdio.h>はまた同じ意味になります。

Unix 環境で-lc を使用するときには、必ず実際のディレクトリ及びファイル名を小文字にしておいてください。

Imagix 4D は Windows と Unix の両方で実行できるため、Windows 環境でも大文字小文字の区別に関する問題があり得ます。Imagix 4D データベースでは大文字と小文字のファイル名を区別できるため、データベースの中で 1 つのファイルが 2 つの異なるファイルとして表示される場合があります。たとえば、あるファイルを#include <stdio.h>指令と#include <STDIO.H>指令の両方を通じてインクルードすると、データベースに 2 回表示されることになります。-lc オプションを Windows で使用することで、この大文字小文字の問題は効果的に解消されます。

-lc オプションを Windows で使用する場合は、ファイル名を変更する必要はありません。

オプションは、Data Sources ダイアログの Options タブで解析オプションのひとつとして選択します。

-lci #include 指令に対するファイル参照を小文字だけに変換します。

-lc オプションを使用すると、アナライザに渡されるすべてのディレクトリとファイル名が小文字に変換されます。#include 指令内の名前のほかに、-lc オプションによって、-H、-I、-R、または-S フラグで指定されたディレクトリ名と、解析用にアナライザに渡されるソースファイル名も変換されます。したがって、Unix 環境では、ソースコードとヘッダファイルに至る完全なディレクトリ構造では、ディレクトリとファイル名は小文字でなければなりません。

-lci オプションでは、#include 指令内の名前だけが変換されます。これは、Windows コードを Unix 環境で使用し、既存の上位の Unix ディレクトリの大文字小文字を変更するのが実際的でない場合に役立ちます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-locals ローカル変数に関するデータを収集します。

通常、Imagix 4D アナライザはグローバル変数及び静的変数に関するデータだけを収集し、ローカル変数は定義されているところで関数によってセットされるか、読み取られるだけです。つまり、ほとんどソフトウェア理解の助けになりません。しかし、場合によっては、ローカル変数に関するデータを収集したいと思うこともあるでしょう。たとえば、クラスカップリングのようなある種の品質メトリックスをしようと思っているときです。-locals オプションはローカル変数の定義及び使用法に関するデータをデータファイルに追加させます。

オプションは、Data Sources ダイアログの Options タブで解析オプションのひとつとして指定します。

-mark あらゆるアナライザメッセージに"imagix: "を挿入させます。

アナライザのエラーメッセージは、通常、それが発生したファイルの名前からはじまります。mark オプションを使用すると、すべてのエラーメッセージの先頭に文字列"imagix: "が挿入されます。これは、メイクファイルでアナライザを実行しているときのようにアナライザのエラーメッセージが標準エラー出力（stderr）への他のメッセージと混ざっているときには便利です。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-msc Microsoft 言語拡張を有効にします。

Microsoft Visual C/C++コンパイラは、標準 C/C++言語へのいくつかのキーワード拡張を受け付けます。これらの拡張の多くは、コンパイラ用のコンパイラ設定ファイルの#define keyword 指令を通してサポートすることができます。ただし、一部の拡張は、#define を使用しても与えることのできない追加のサポートを必要とします。このサポートを与えるのが-msc オプションです。

Microsoft Visual C++では、キーワード_asm、__asm はインラインアセンブラコードの始まりを示します。キーワード_segment、__segment、_base、__base はセグメント化されたメモリ及びオフセットを処理します。-msc オプションは Imagix 4D アナライザに、問題を避け、これらの言語拡張が使用されているソースコードを解析することを可能にします。

-msc オプションはまた、Microsoft Visual C++での COM プログラミングの型ライブラリの概念で使用される前処理指令#importもサポートします。-msc オプションで次のコードがあると、

```
#import <filename.ext>
#import "filename.ext"
```

アナライザは、-I、-S オプションで指定されているインクルードディレクトリを検索し、.tlh ファイルを見つけます。

さらに、-msc はアナライザによって収集されるシンボル及び関係情報に影響を与えない単純な修飾子なので、いくつかのキーワードを実質的に無視させます。無視されるキーワードは、cdecl、far、fortran、huge、near、pascal、_cdecl、_export、_far、_fastcall、_fortran、_huge、_interrupt、_loadds、_near、_pascal、_saveregs、__cdecl、__export、__fastcall、__far、__fortran、__huge、__interrupt、__loadds、__near、__pascal、__severegs です。これらのキーワードについては、コンパイラ設定ファイルに#define keyword の行を追加することによって、同じ解析結果を達成することができます。

ほかにも、少なくともこれらの Microsoft 拡張のサブセットをサポートするコンパイラがいくつかあるので、-msc オプションは、非 Microsoft コードにとっては有用であることがおわかりになるでしょう。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-msg*Filename* アナライザのメッセージを標準出力ではなく *Filename* に書き出します。

ソースコードを処理する際に、アナライザは処理中のファイルや遭遇した解析上の問題に関する一連のメッセージを出力します。通常、アナライザはこれらのメッセージを標準出力に送ります。 -msg オプションを指定すると、メッセージは標準出力ではなくファイルに直接書き出されます。

-msrc ソースコードの複数の解析パスの結果を格納します。

アナライザがソースコードを処理すると、単一のヘッダファイルが複数回処理され、複数の異なるファイルにインクルードされます。ヘッダファイルが解析されるたびに、マクロ定義の現在のステータスが使用されます。#ifdef、#ifndef、及び#if を使用すると、ヘッダを通じて各パスから異なる結果を得ることができます。たとえば、typedef は 1 つのパスで特定の方法で定義し、別のパスではまったく定義しないことが可能です。

ソースアナライザがコードを処理すると、下流の解析で正しいタイプ定義などを使用できるように、最新のパスのヘッダファイルについての情報が保持されます。ただしディスプレイやレポートで情報を提示するには、ファイルの内容を単一の表示で示す必要があります。

Imagix 4D の通常のデフォルト動作では、アナライザの最初のパスに基づいて、ヘッダファイルを通じてデータファイルが格納され、後で使用されます。これには、最初に条件付きで有効であったコードが反映されます。

場合によっては、ヘッダファイルのパスを通じて有効になったことがある、すべてのコードを調べることがあります。-msrc スイッチを使用することで、ヘッダファイルを通じたすべてのパスの結果がデータファイルに格納され、Imagix 4D のディスプレイとレポートの作成に使用されます。

オプションは、Data Sources ダイアログの Options タブで解析オプションのひとつとして選択します。
-msrc は、ソースアナライザの同じ呼び出しでのみ機能します。メイクファイルによる手法や、異なるプロジェクトが結合してコンボプロジェクトになるケースで起きることのある、異なるソースアナライザによる同じヘッダファイルの呼び出しは追跡されません。

-msrcf　ソースコードの複数の解析パスの結果を格納します。（拡張）

1 つのヘッダファイルを複数のパスで処理する場合、-msrc オプションを指定するとアナライザは既に解析された行がどれであるかを記録します。各パスについて、解析されていない行に関する情報が収集されます。多くの場合、条件付きでコンパイルされたコードの意味を捉えるにはこれで十分です。

しかし場合によっては、ある行の意味および得られるクロスリファレンスデータが、1 つの解析パスと次の解析パスとで変わってしまうことがあります。そのような場合、-msrcf は-msrc に代わる強力な機能を提供します。基本的に-msrcf オプションでは、ヘッダファイルがインクルードされるたびにヘッダファイルのすべての行に関する情報が集められます。このオプションを使用するとデータベースのサイズが大幅に増大してパフォーマンスが遅くなる可能性があるため、慎重に使用するべきです。 このオプションは、異なる定義設定に基づいて新しいコードを生成するためにヘッダファイルを故意に複数回にわたってインクルードする場合にのみ意義があります。

-msrcf のもう 1 つの呼び出し方法に、-msrcf*filename.c* という形式があります。この方法で呼び出されると、filename.c の解析中に限り、より強力なデータ収集が適用されます。他のソースファイルは-msrc が使用されているかのように解析されます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加するか、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。-msrcf オプションが使用されている場合、-msrc オプションは無効にするべきです。

-msrcf は、ソースアナライザの同じ呼び出しでのみ機能することに注意してください。メイクファイルによる手法や、異なるプロジェクトが結合してコンボプロジェクトになるケースで起きることのある、異なるソースアナライザによる同じヘッダファイルの呼び出しは追跡されません。

-nestcom　ネストされた C 型コメントをサポートします。

大半のコンパイラは入れ子になった C 型コメントの使用をサポートしません。次のようなコードでは、そうしたコンパイラは C をひとつのシンボルと見なし、C の次の*/で問題を指摘します。

```
/* A /* B */ C */ D = 1;
```

ただし、一部のコンパイラは C*/を/*A で始まるコメントのクローズ部として扱い、D までのすべてを無視します。-nestcom オプションを使用すると、Imagix 4D アナライザはこの後者のコンパイラのように動作します。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-neu　Neuron C 言語の使用をサポートします。

Neuron C 言語は C 言語を正規に修正したもので、Echelon ハードウェア用に Echelon クロスコンパイラでサポートされています。-neu オプションはアナライザに Neuron C 構文と意味を認識させ、サポートさせます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加するか、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-noendasm　#asm を単一行のプリコンパイラ指令としてサポートします。

一部のコンパイラは、C/C++ソースファイル内でのインラインアセンブラのコードをサポートします。これはさまざまな言語構成要素を通して行います（次項「言語拡張」参照）。一般的なコンパイラ規則は、プリコンパイラ指令#asm や#endasm を含めてアナライザによって自動的にサポートされ、インラインアセンブラ行が区別されます。ただしこの方法で#asm をサポートしないコンパイラもあります。コンパイラによっては、#asm を行頭に置き、その行の残りをインラインアセンブラコードで構成することもできます。この場合、#endasm 指令のクローズ部はありません。このオプションでは、対応する#endasm を置かない#asm の使用がサポートされています。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-nosys システムヘッダファイルのデータを生成しません。

インクルードディレクトリを指定するのに-I*dirname* ではなく-S*dirname* を使用すると、ひとつのディレクトリをアプリケーションディレクトリではなくシステムディレクトリとして指定することになります。-nosys オプションが使用されているか否かに関係なく、そのシステムディレクトリからのヘッダファイルはインクルードされるときに解析されます。ソフトウェアを正確に解析するためには、そうすることが必要だからです。ただし、-nosys オプションがセットされていると、システムディレクトリで見つかったヘッダファイルのデータは収集されません。

-nosys を使用すると、fprintf のようなシステムシンボルが宣言されているファイルのブラウズ及び調査ができなくなるという影響が出ます。一般に、stdio.h のようなファイルの内容を調べることにはほとんど価値がありませんから、通常は-nosys を有効にし、そうすることによって、自分にとってはほとんど価値のないデータをフィルタにかけて排除します。

オプションは、Data Sources ダイアログの Options タブで解析オプションのひとつとして選択します。

-nosysbodies システムヘッダファイルに定義された関数のボディに関するデータを生成しません

インクルードディレクトリを指定するのに-I*dirname* ではなく-S*dirname* を使用すると、ひとつのディレクトリをアプリケーションディレクトリではなくシステムディレクトリとして指定することになります。-nosysbodies オプションが使用されているか否かに関係なく、そのシステムディレクトリからのヘッダファイルはインクルードされるときに解析されます。ソフトウェアを正確に解析するためには、そうすることが必要だからです。ただし、-nosysbodies オプションがセットされていると、システムディレクトリに定義された関数の内容に関するデータは生成されません。要するに、これは-nosys の重要度の低い形式です。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-o*Filename* 解析結果をデータファイル *Filename* に書き込みます

アナライザは明示的に解析されるソースファイル、及び直接的または間接的にそれらのソースファイルに含まれるヘッダファイルの個々についてエンティティ/関係/属性データを生成します。例外として考えられるのは、システムインクルードディレクトリのヘッダファイルです（-nosys オプション及び-S オプションの項参照）。

-o オプションが使用されているときには、すべてのソースファイル及びヘッダファイルに関するデータが単一のファイル *Filename* に格納されます。このオプションは、従来の互換性を確保するために利用できるようになっています。-genvdb オプションを使用することをおすすめします。

オプションは、Imagix 4D のユーザインターフェイスを通してはセットされません。

-R*dirname* #include ファイルをカレントディレクトリで検索する前に *dirname* で検索します。

アナライザは、#include "filename"前処理指令に出会うと、まず-R 及び-H オプションを通して指定されたすべてのディレクトリで filename を検索します。それで見つからない場合には、カレントディレクトリ（"."）で検索し、それでもまだ見つからない場合には、-I 及び-S オプションで指定されたディレクトリで検索します。#include "filename"ではなく、#include <filename>が使用されている場合には、カレントディレクトリ（"."）はスキップされます。

-R で指定されたディレクトリは、オプションとして指定されている順番で検索されます。これらのディレクトリはシステムインクルードディレクトリとして、-S オプションで指定されたディレクトリと同じように扱われます。これらのディレクトリをアプリケーションインクルードディレクトリとして扱うには、-H オプションを使用します。その場合の影響については、-nosys オプションの項を参照してください。

オプションは、Data Sources ダイアログの Include Dirs タブにある-I、-S Flags フィールド、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。Windows 版アナライザは/R も受け付けます。

-S*dirname* #include ファイルを検索するディレクトリのリストに *dirname* を追加します

アナライザは、#include "filename"前処理指令に出会うと、まず-H 及び-R オプションを通して指定されたすべてのディレクトリで filename を検索します。それで見つからない場合には、カレントディレクトリ（"."）で検索し、それでもまだ見つからない場合には、-I 及び-S オプションで指定されたディレクトリで検索します。#include "filename"ではなく、#include <filename>が使用されている場合には、カレントディレクトリ（"."）はスキップされます。

Imagix 4D アナライザが#include <filename>と#include "filename"を区別するのは、カレントディレクトリを検索するかどうかを判断するためにすぎません。インクルードディレクトリをアプリケーションインクルードディレクトリとして宣言するには、-S*dirname* よりも-I*dirname* を使用します。その場合の影響については、-nosys オプションの項を参照してください。

オプションは、Data Sources ダイアログの Include Dirs タブにある-I、-SFlags フィールド、あるいはメイクファイルで指定します。Windows 版アナライザは/S も受け付けます。

-sfx -gencfd 及び-genqcf データに関連付けられたファイル名に接尾語を付けます

通常、-gencfd 及び-genqcf スイッチで生成されるデータは、名前の最後が対応するオリジナルのソースファイル及びヘッダファイルと同じ接尾語で終わるファイルに格納されます。このため、一部の設定管理システムで障害が生じる可能性があります。-sfx オプションは、.c ソースファイルから生成されたデータが.c_sfx の接尾語を持つファイルに格納されるように、ファイル名の最後に_sfx を追加します。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-sysincfirst < >によって指定されるヘッダファイルについて、-I ディレクトリの前に-S ディレクトリを検索します。

インクルードされたファイルの場所を特定するとき、デフォルトの動作では-I オプションと-S オプションが指定されている順序でインクルードディレクトリを検索します。アプリケーションヘッダファイルが#include "appfile.h"のようにインクルードされ、システムインクルードファイルが#include <sysfile.h>のようにインクルードされている場合、-sysincfirst オプションを使用するとファイルの検索がインクルードディレクトリの構成要素の適切な集合から開始されるので、名前の衝突を減らすことができます（*ユーザガイド*の「システムインクルードディレクトリとアプリケーションインクルードディレクトリ」のセクションをご覧ください）。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加するか、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-traditional C 言語の Kernigan & Ritchie ダイアレクトをサポートします。

Imagix 4D アナライザではほとんどの場合、コードが ANSI C またはより古い K&R C ダイアレクトに準拠するかどうかをユーザが指定しなくても、C コードを解析できます。ただし、特定の構成要素では、2 つのダイアレクト間の動作の違いが自動的に解決されないコメントがマクロに含まれています。デフォルトでは、アナライザは ANSI C をサポートしています。オプションを適用すると、K&R 仕様がサポートされます。このオプションでは -kr を呼び出すこともできます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-trkmem 構造体/共用体メンバの実体化をそれぞれ別の変数として追跡します。

-trkmem オプションは、集合変数(構造体や共用体などの集成体型の変数)のメンバが imagix-csrc によってどのように解析され、その後のユーザインターフェイスにおいてどう表現されるかに影響します。デフォルトの動作では、集合変数そのものは別々の変数として追跡しますが、集合変数間で同等のメンバを 1 つの変数に結合して追跡します。例えば、var1 と var2 という 2 つの変数が structA という同じ型を持ち、structA にメンバ memA が含まれているとします。ツールのデフォルトの動作では var1 と var2 を別々に追跡しますが、var1.memA と var2.memA を一緒にして、単に memA として考えます。これによってツールの多くの利用局面において、相応の代償として結果となるデータベースに特異性と複雑性が生じることになります。 しかしながら、memA の実体化をそれぞれ別に追跡したい場合もあるでしょう。-trkmem オプションによりそれが実現でき、var1.memA は var2.memA とは異なる変数であるとみなされます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加するか、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-U*macroname* マクロ *macroname* を未定義にします

これはコンパイラのオプション-U と同じです。マクロ置換は、アナライザによるソースコードの前処理の一部です。マクロが未定義の間、条件付きコンパイル指令#ifdef *macroname* によって定義されたブロックのコードは解析されません。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールド、あるいはメイクファイルで指定します。Windows 版アナライザは/U も受け付けます。

-uaggr 異なるファイルに定義された最上位のクラスを別々のクラスとみなします。

コンパイラの標準的な動作では、グローバル命名規則が適用され、同一コンパイル単位（実行可能ファイル、DLL またはライブラリ）にある同じ名前を持つすべての最上位クラスは同じクラスであると見なされます。 しかし、複数のコンパイル単位にまたがったプロジェクトを定義したい場合もあるでしょう。そのような場合、別のコンパイル単位において偶然、実際には別々であるクラスに同じ最上位クラスの名前が再利用されてしまうことがあります。このような名前の衝突を-uaggr オプションによって防止することができます。-uaggr が使用されると、異なるファイルに定義されたすべての最上位クラスは、たとえ同じ名前であっても別々のクラスであるとみなされます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加するか、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-unique 異なるファイルに定義された typedef を別々の typedef とみなします。

imagix-csrc ソースアナライザのデフォルトの動作では、重複した型定義（複数のファイルに同じ名前で定義された typedef）はプログラミング上の利便性をはかったものであり、同じ typedef を意図していると

想定します。例えば、大きいヘッダファイルをインクルードしないようにするために、単純な typedef がソースファイルに追加されることがあるかもしれません。共通の型をヘッダファイルに定義することを要求するコーディング・ガイドラインを適用している場合は、-unique オプションを使用することにより、同じ名前を持つ別々に定義された型が異なる typedef として扱われるようにすることができます。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加するか、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-vms VMS 言語拡張を有効にします

大半のコンパイラは、標準 C/C++言語へのいくつかのキーワード拡張を受け付けます。これらの拡張の多くは、コンパイラ設定ファイルの#define *keyword* 指令を通してサポートされます。ただし、一部の拡張は#define でサポートされず、追加のサポートを必要とします。一部のキーワードについて、そのサポートを提供するのが-vms オプションです。

具体的には、_align（識別子）及び__align（識別子）宣言指定子の使用についてサポートを追加します。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-vuid 解析する各シンボルについて VUID を生成します。

プロジェクトに関するドキュメントを作成する場合、通常は 1 つのパスにすべてのドキュメントを置きます。この場合、データベースがロードされると、ハイパーテキストリンクで使用されるシンボル識別子が割り当てられます。ドキュメントは 1 つのセッションで作成されるため、各 HTML ページで同じ識別子が使用されます。

-vuid オプションでは、プロジェクトがロードされたときではなく、ソースコードの解析時にビジュアライザ固有の識別子が生成されます。これによりプロジェクトセッション間の識別子の整合性がとられ、部分的及び増分の HTML ドキュメント作成が可能になります。部分的及び増分のドキュメント作成の詳細については、ファイル../imagix/user/doc_gen/sample.dg_を参照してください。

オプションは、Data Sources ダイアログの Source Files タブにある PP Flags フィールドで追加、あるいはコンパイラ設定ファイルまたはメイクファイルで指定することができます。

-warn すべてのアナライザ警告を返します

ソースコードの解析中、Imagix 4D アナライザはソースコードそのもの、あるいはコード解析に使用されるマクロ定義及びインクルードパスによって生じた構文または意味エラーに出会うことがあります。

デフォルトでは、アナライザは、インクルードファイルが見つからない、あるいはアナライザを再同期するためにコードを何行かスキップする必要があるといった重要な問題についてのみエラーメッセージを返します。-warn オプションを使用すると、アナライザは出会う問題すべてについてメッセージを返します。

場合によっては、あとで解析上の重大なエラーを引き起こす問題を、初期のうちに隔離するために-warn オプションを使用してもよいでしょう。

オプションはつねに有効になっています。警告メッセージは、Show Warning Messages チェックボックスに従って（LOCAL メニュー［Display］>［Show Warning Messages］）Analysis Results ウィンドウで表示またはフィルタされます。

サポートされていないオプション

Imagix 4D アナライザは、サポートしないオプションを渡されると、それをマクロ定義に変換します。たとえば、-ansi はマクロ__IMGX_ansi__を定義させ、-Xc は__IMGX_Xc__の定義をもたらします。オプショ

ンのなかの非英数字はそれぞれ$<ASCII-value>に変換されます。たとえば、アナライザにオプション-xc++を渡すと、マクロ__IMGX_xc$43$43__が定義されます。

この変換は Imagix 4D のコンパイラ設定ファイルでのカレントのコンパイル関連オプションの使用に柔軟性を持たせます。

-H、-I、-R または-S で開始するオプションは、インクルードパスを示すと解釈され、サポートされていないオプションとは見なされません。

言語拡張

確立された ANSI C プログラミング言語の規格と、より新しい C++言語の規格があります。ただし、多くの既存の C/C++ソフトウェアはこれらの規格を満たしていません。これは、ひとつには（C の K&R の時期のように）言語の規格が最終的にかたまる前にソフトウェアが書かれているからで、ひとつには、コンパイラが正式言語への拡張をサポートしているからです。

Imagix 4D アナライザはこうした既存のコードを幅広く処理するために開発されました。

多くの場合、このサポートは自動的に生じます。Imagix 4D アナライザはソフトウェアの解析中に数多くの歴史的言語構成要素を認識します。たとえば、古い K&R 型条件付きコンパイル指令（#if 0）も新しい C++型コメント（//）も C コード解析中にサポートされます。また、Imagix 4D アナライザは数多くのコンパイラ特定言語拡張もサポートします。たとえば、一部の VAX コンパイラ及びクロスコンパイラによって認められている、シンボル名のなかでの文字$の使用（na$me）も処理します。

キーワード拡張の場合には、Imagix 4D プリプロセッサがソースファイルを適切な C/C++ソースコードに変換するようにマクロ定義を使用する必要があります。たとえば、数多くの C クロスコンパイラはキーワード near 及び far を、ANSI C では定義されていなくてもサポートします。この場合には、near 及び far を空の文字列で置き換えるマクロ定義を生成すればよいでしょう。そうすれば、実質的に、Imagix 4D アナライザはコードのなかにあるそれらを無視するでしょう。

こうしたマクロ置換を定義するには、それらをコンパイラ設定ファイルのなかに置くのがいちばんです。そうすれば、サポートするコンパイラごとに、それを一度定義するだけですみ、修正するのも容易になります。例を見るには、../imagix/user/cc_cfg のディレクトリの.inc ファイルを参照してください。とりわけ、bor_win.inc 及び msvc_win.inc ファイルはそのようなマクロ定義を数多く含んでいます。

一部の言語拡張は、このような方法では定義することができず、-gnu、-msc 情報のように、アナライザオプションを通した特別な処理を必要とします（前項「アナライザの構文とオプション」参照）。これらのオプションは、#pragma cmdflag -msc のような行を追加することにより、コンパイラ設定ファイルに常に組み込むことができます。

インラインアセンブラ

特別な処理を必要とする言語拡張には、必ずインラインアセンブラコードが含まれます。多くのコンパイラは、C/C++ソースファイルのなかへのアセンブラコードの記述をサポートします。Imagix 4D アナライザはインラインアセンブラを理解しようとしませんが、周辺の C/C++コードを解析する上でインラインアセンブラが問題を起こさないように、特別な処理が行われます。

ひとつには、インラインアセンブラが言語規格の一部ではないこともあって、コードの特定の行が C/C++ではなくインラインアセンブラであることを指示するために、数多くの異なった構成要素が生成されています。これらの構成要素の一部は、いかなる状況においても Imagix 4D アナライザによって自動的に処理されます。ほかの構成要素は潜在的に言語の他の部分と矛盾する可能性があるので、特定のアナライザオプションが使用されているときにのみサポートされます。

以下のフォーマットの単一行は、つねに Imagix 4D アナライザによってスキップされます。

```
#asm (....)

#asm <...>

#asm "..."
```

以下の複数行の構成要素も、つねにアナライザによってスキップされます。

```
#asm
... 任意の数の行...
#endasm

#pragma asm
... 任意の数の行...
#pragma endasm

asm {
... 任意の数の行...
}
```

-noendasm オプションを使用しない限り、

```
#asm anything else on this line
```

はスキップされますが、#asm と#endasm の間の行はスキップされません。

多くのコンパイラはソースファイルに、関数のボディがインラインアセンブラのなかに実装されている関数定義を含むことを可能にします。このためによく使用されるフォーマットは次の通りです。

```
ASM declarator_part declaration_list {
... 任意の数の行...
}
```

こうした関数定義のサポートは、関数名 declarator_part の認識も含め、-asm、-gnu、-msc アナライザオプションを通して使用可能です。キーワード ASM は、-asm オプション使用時は asm、-gnu オプション使用時は__asm、-msc オプション使用時は_asm または__asm になります。このキーワードサポートは、-D オプションを通して、あるいはコンパイラ設定ファイルの#define statements を介し、マクロ定義を使用することによって拡張することができます。たとえば、オプション-D_asm=asm と-asm オプションを組み合わせて使用すれば、キーワード ASM は_asm でも asm でもかまいません。

もうひとつ、数多くのコンパイラによってサポートされていて、いくつかのフォーマットで使用可能な、よく用いられる構文があります。

```
ASM OPTvolatile {
... 任意の数の行...
}

ASM OPTvolatile (
... 任意の数の行...
)
```

この構成要素は単一行のフォーマットでも使用されます。

```
ASM OPTvolatile ...
```

Imagix 4D アナライザのこの構成要素のサポートは、関数定義に使用されているのと同じオプション、-asm、-gnu、-msc を通して有効になります。キーワード ASM は、上記のようにオプションによって asm、_asm、または__asm を表します。OPTvolatile キーワードは volatile または省略する必要があります。

前処理

コンパイラの場合と同じで、コード解析の第 1 段階では、コードを前処理し、純粋な C/C++ソフトウェアに変換します。この段階では、#define、#ifdef、#include のような前処理指令が展開され、/*と*/ではさまれたコメントは無視されます。

ソースコードを正確に前処理するため、Imagix 4D アナライザはコンパイラが使用するのと同じ情報の一部を必要とします。アナライザはヘッダファイルを探す場所を知る必要があります。また、(#if、#ifdef などの)条件付きコンパイル指令で使用されるマクロが定義されているかどうか、そして、定義されているとしたら、どのように定義されているかも知る必要があります。

通常、この情報はプリプロセッサオプションを通してコンパイラに渡されます。Imagix4D アナライザは同じ引数を使用します。-I*dirname*(Windows システムでは/I*dirname*)はインクルードディレクトリの指定に使用されます。これらのディレクトリでは、リストされている順に、#include ステートメントで参照されているヘッダファイルが検索されます。

同様に、引数-D*macroname* または-D*macroname=macrovalue* はコンパイラの場合同様にマクロを定義するために使用されます。これらのマクロ定義は、さらに、#ifdef などの条件付きコンパイル指令の処理方法を計算するために使用されます。逆のオプション-U*macroname* もアナライザによって認識され、マクロを未定義にします。

前処理及び実処理の両方を行うことにより、Imagix 4D アナライザはマクロに関する完全な情報を生成することができます。次のような例を考えてみましょう。

```
/* ファイルの第 1 行 */
#define macroB(x) funcC(x)

void funcA(int x)
{
int y;
y = macroB(x);
}
```

アナライザは、funcA が macroB を使用する(呼び出す)という情報とともに、funcA が funcC を呼び出すという情報を生成します。データファイルにはまた、funcA による macroB の呼び出しと funcA による funcC の呼び出しがともに 7 行目で起こるという情報も含まれます。この後者の、シンボルが参照される場所に関する情報は、数多くのアプリケーションを持ち、とりわけ Use Browser ディスプレイでシンボルが使用されているすべての場所を表示するときに使用されます。

コンパイラ設定ファイル

Imagix 4D アナライザはコンパイラから独立しています。ソフトウェアをコンパイラと同じように解析するには、コンパイラをエミュレートするように Imagix 4D をセットアップする必要があります。そのセットアップの手段となるのがコンパイラ設定ファイルです。

/imagix/user/cc_cfg で見つかるこの設定ファイルは、Imagix 4D アナライザをコードのコンパイルに使用するコンパイラと同様に動作するようにセットアップします。c ソースコードを含むこのファイルは、アナライザに必要な 3 つのものを提供します。第 1 に、stdio.h のようなシステムヘッダファイルを探す場所をアナライザに指示します。これは、次のように指定されます。

```
#pragma cmdflag -S/usr/include
```

第 2 に、コンパイラ設定ファイルはコンパイラによってサポートされる言語拡張を補正します。たとえば、多くの C コンパイラがキーワード near をサポートします。これはコンパイラによるメモリ割付に影響します

が、ソフトウェアの構造には何の意味もありません。もうひとつの例はキーワード byte です。これは標準 C 言語型ではありませんが、一部のコンパイラによって事前定義されています。コンパイラ設定ファイルは、マクロ定義と typedef を生成し、これらの拡張を次のように補正することを可能にします。

```
#define near
#define byte char
typedef char byte;
```

コンパイラエミュレーションの第 3 の領域は、コンパイラがシステムヘッダファイルを前処理するときに使用するマクロ定義に関係します。しばしば、コンパイラのヘッダファイルはいくつものターゲットシステム用にソフトウェアをビルドするようにセットアップされます。そのようなヘッダファイルには、コンパイラがビルドされるターゲットにふさわしいヘッダファイルのソースコードのセクションを使用するように、#ifdef _I386 のような条件付きコンパイル指令が含まれます。そのような暗黙的マクロ定義はコンパイラ設定ファイルのなかで次のように明示的にすることにより、コンパイラ設定ファイルによってサポートされます。

```
#define __I386
```

上記のキーワード拡張及び暗黙的マクロ定義は、しばしばコンパイラのドキュメントのなかに記述されます。キーワード、言語拡張、事前定義マクロ、あるいはマクロ定義に関する情報を探してください。

Imagix 4D には、主なコンパイラ用にいくつものコンパイラ設定ファイルがあらかじめセットアップされています。該当するコンパイラを使用する場合には、そのコンパイラ設定ファイルをビルド環境におけるシステムヘッダファイルの場所を反映するように少し修正してください。該当しないコンパイラを使用する場合には、使用するコンパイラをエミュレートするためのコンパイラ設定ファイルを必ず生成しなければならないわけではありませんが、生成することを強くおすすめします。

アナライザの起動

Imagix 4D C/C++アナライザは独立した実行ファイルであり、アナライザの構文に従い、コマンドラインを介して起動することができます。ただし、操作を簡単にするため、通常、起動は Imagix 4D ユーザインターフェイスの内部から制御されます。

Imagix 4D ツールは、カレントプロジェクトに関する知識を利用して、生成されるデータファイルの格納場所に関するアナライザスイッチをセットします。残りの必要な情報――どのソースファイルを解析し、どのインクルードディレクトリを検索し、どのマクロ定義を使用するか――は Data Sources ダイアログ（メニュー［Project］> ［Data Sources］）で指定します。ダイアログの左側で選択された各データソースについて、特定のデータソースの設定はダイアログの右側で行います。右側の上部にある Type メニューボタンで、メニューの Source Files セクションにある 4 つの項目のいずれかによって Imagix 4D C/C++アナライザが起動されます。ここで選択した解析方法により、ダイアログのあとの表示は変わってきます。ダイアログを完成させる方法については、このマニュアルの「*はじめに*」の項、及び状況連動型ヘルプ画面を参照してください。

Options ダイアログ（メニュー[Tools] > [Options] > ［Data Collection］）の設定はローカル変数に関するデータを収集するかどうかの選択など、Data Sources ダイアログの各 Options タブに表示されるアナライザオプションのデフォルト設定のために使用します。

C/C++ - ダイアログを通して解析を行う

この方法では、アナライザは Data Sources ダイアログの設定に従い、ユーザインターフェイスから直接起動されます。ダイアログでは、解析するソースファイル、検索するインクルードファイルの場所、使用するマクロ定義などを指定します。

このダイアログは、ある特定のディレクトリのソース(.c 及び.cpp)ファイル、あるいは、ある特定のディレクトリ及びそのすべてのサブディレクトリのソースファイル解析をサポートします。ソフトウェアが複数のディレクトリにまたがっている場合には、各ディレクトリにアナライザを複数回、起動する必要があります。その場合には、そのつど、Data Sources ダイアログでデータソースを追加することになります。

ただし、このディレクトリの問題はヘッダ(.h または.hpp)ファイルには適用されません。インクルードファイルの検索場所は Include Dirs タブで指定します。1 つは-I*dirname* を-I, -S Flags フィールドに挿入する方法があります。これは任意の数のディレクトに対して行うことができます。

また、Directory フィールドで指定したディレクトリのすべてのサブディレクトリにあるヘッダファイルを検索するときは、Search subdirectories for header files のチェックボックスを利用することができます。ただし、このチェックボックスを利用した場合には、サブディレクトリの検索する順序を制御することはできなくなります。このため、同じ名前のヘッダファイルが複数ある場合には、不正確になる可能性があります。

メイクファイルを使用して解析を行う

この選択肢はメイクファイルにターゲットのセットを追加します。したがって、これを選択する場合には、メイクファイルを使ってソフトウェアをビルドする必要があります。この方法では、新しいメイクファイルターゲットがさらに実際の Imagix 4D アナライザを起動します。

メイクファイルにはすでに、コードをコンパイルするときにコンパイルするソースファイル、ヘッダファイルを検索するインクルードディレクトリ、及び使用するマクロ定義をリストするメイクファイルマクロが含まれています。Imagix 4D によって追加されるターゲットは、これら既存のマクロ上にビルドされます。おそらく、Imagix 4D の関連ターゲットを編集する必要があるでしょう。このため、メイクファイルにターゲットを追加したときには、エディタに修正されたメイクファイルそのものが、新しい Imagix 4D ターゲットの修正に関する指示とともに表示されます(「*はじめに*」参照)。

ターゲットがメイクファイルに追加されると、Data Sources ダイアログの Options タブの設定がメイクファイルの設定にインクルードされます。このオプションをあとで変えるには、プロジェクトデータを再生する前に、単に Options タブで設定を変更するだけでなく、メイクファイルを修正する必要があります。

Data Collection Options ダイアログの Make タブの設定によって、Imagix 4D ターゲットが最初に追加される方法について、一部の要素を制御できます。さらに、ターゲットの追加に使用するテンプレートのカスタマイズも可能です。テンプレートは、使用方法と合わせて../imagix/user/mk_trgts に置かれています。

MSVC Project または MSVC Workspace/Solution

この方法では、Imagix のターゲットをメイクファイルに追加するのではなく、Microsoft Visual Studio で生成されたプロジェクトまたはソリューションから必要なファイル、インクルードディレクトリ、及びマクロ定義に関する情報を抽出します。プロジェクトまたはソリューション(以前のバージョンではワークスペース)を作成すると、Visual Studio ではこの情報がファイルで格納されます。これらのファイルのフォーマットと拡張子は、Visual Studio のバージョンによって異なります。MSVC を使用するこの方法は、Windows 及び Unix の両方のバージョンの Imagix 4D で使用できます。ただしバージョン 5 以前の.mak ファイルタイプを使用する場合は Windows バージョンが必要になります。

Microsoft Visual Studio のバージョン 5 及び 6 では、.dsp ファイルではなく.mak ファイルを使用してインポートを行う場合は、プロジェクト用にメイクファイルは自動的に生成されません。MSVC の Project メニューで Export Makefile メニューを選択すると、Visual Studio にメイクファイルを生成させることができます。メイクファイルには通常、*project*.mak という名前が付けられます。ただし、*project* は MSVC プロジェクトの名前です。

ときおり、MSVC で生成されたメイクファイルには、その動作を阻むエラーが含まれていることがあり、そのため、MSVC Project を利用してデータをインポートしようとすると、エラーが表示されることがあります。

MSVC Project を利用していて問題にぶつかったときには、DOS ウィンドウのコマンドラインから nmake -n -p -k -f *project*.mak を実行することにより、それがエラーのあるメイクファイルによって生じたものかどうかを確かめることができます。MSVC Project を利用しようとしていて表示されたのと同じエラーメッセージが表示されれば、問題はメイクファイルにあります。この場合はメイクファイルを修正するか、別の方法を選択します。

アナライザの起動方法（解析方法）の選択

すでに MSVC を使用している場合には、MSVC Project または MSVC Solution の方法がいちばん簡単でしょう。また、その場合には Imagix 4D プロジェクトを MSVC プロジェクトまたはソリューションのあらゆる変更に同期させておくことも容易です。Imagix 4D プロジェクトデータを再生するだけでよいでしょう。

C/C++コードでダイアログを通して解析を行う方法も、メイクファイルを使用した方法に比べると、通常ははるかに利用が容易です。ただし、夜間ビルドなどのためにメイクファイルを多用する実働環境で作業をしている場合には、その標準的なビルド環境で Imagix 4D のデータ収集も制御するほうが、メイクファイルに Imagix 4D ターゲットを追加するためによけいな手間をとられても、都合がよいこともあるでしょう。

解析上の注意点

ヘッダファイルの解析

一般に、Imagix 4D アナライザに渡すファイルのリストにヘッダファイルをインクルードする必要はありません。.c、.cpp ファイル及びインクルードディレクトリを指定することにより、ヘッダファイルが#include ステートメントを通して直接的または間接的にインクルードされれば、それらは自動的に解析されます。ヘッダファイルをリストするのは、たとえ C または C++ソースファイルにインクルードされなくても、プロジェクトに追加されるようにするためにすぎません。

システムインクルードディレクトリとアプリケーションインクルードディレクトリ

コンパイラの場合と同様に、Imagix 4D アナライザでも、インクルードディレクトリのリストを指定します。アナライザは、#include 前処理指令に出会うと、インクルードディレクトリを指定されている順に、#include ステートメントで指定された名前と適合するファイルが見つかるまで検索します。デフォルトでは、次の 2 つのインクルードステートメントに対する Imagix 4D アナライザの動作は、1 点だけ異なります。

```
#include "fileA.h"
#include <fileA.h>
```

ファイル名がクォーテーションマークで囲まれている上のステートメントでは、アナライザはまずカレントディレクトリで fileA.h を検索します。それを除けば、あとはどちらのステートメントでもアナライザの動作は同じで、指定されたインクルードディレクトリの検索を続けます。この検索は、最初の fileA.h が見つかるまで続きます。

特定のインクルードディレクトリがシステムインクルードディレクトリと見なされるように指定できます。それには、imagix-csrc 起動時に-I*dirname* オプションではなく-S*dirname* を使用します。

-I ではなく-S を使用したときに考えられる影響は 2 つあり、両方ともソースアナライザのオプションに関係します。第一の影響は-nosys オプションに関係します。-nosys オプションがセットされていると、システムインクルードディレクトリとして指定されたディレクトリのヘッダファイルは、解析されますが、データファイルは生成されず、その内容を調査することはできません。このため、-S は、stdio.h のように、一般には関心のないヘッダファイルを含むインクルードディレクトリの指定に使用することをおすすめします。そうすれば、プロジェクトのデータベースを意味のないデータでふさがずにすむでしょう。

-S と -I との第二の違いは、-sysincfirst オプションが適用されている場合に現れます。 -sysincfirst オプションを指定すると、< >によって指定されるヘッダファイルについて、-I ディレクトリの前に-S ディレクトリのすべてが検索されます。サイトでの慣例として、

```
#include "appfile.h"
```

によりアプリケーションヘッダファイルを指定し、

```
#include <sysfile.h>
```

によりシステムヘッダファイルをインクルードしている場合は、-sysincfirst オプションが使用できます。一部のアプリケーションヘッダファイルと一部のシステムヘッダファイルが同じ名前を持っている場合にヘッダファイル名の衝突を解決するうえで、このオプションが有効であるかもしれません。

パス名

Windows はパス名のなかでのスペースの使用をサポートし、また、大文字小文字の区別がありません。Imagix 4D アナライザの起動にあたっても、その点から注意しておかなければならないことがあります。次のような例を考えてみましょう。

```
(1) imagix-csrc -IC:\Program Files\Include -ox File.c
(2) imagix-csrc "-IC:\Program Files\Include" -ox File.c
(3) imagix-csrc "-IC:\PROGRAM FILES\INCLUDE" -ox File.c
```

インクルードディレクトリ C:＼Program Files＼Include には、スペースがひとつ含まれています。このようにスペースを含む引数はダブルクォーテーションマークでくくる必要があります。このため、(1)では解析エラーが生じ、(2)では正しく解析が行われます。Windows は大文字小文字の区別がないので、(2)と(3)は同じものと見なされます。

関数ポインタ

C 及び C++は関数ポインタ、つまり、関数を呼び出すための、物理メモリにおける関数のエントリアドレスのポインタ使用をサポートします。Imagix 4D のソースコード解析が標準的な解析を超えるひとつが、アナライザが関数ポインタの静的側面について捕捉する情報です。

関数名を含むデータとともにセットされる変数はすべて関数ポインタとしてマークされます。関数は、その静的初期化子においてはすべて、関数ポインタから呼び出される（可能性のある）ものとして記録されます。ある関数が関数ポインタを通して呼び出しを行うとすれば、その関数は関数ポインタ変数を呼び出すものとして表示されます。その結果、その関数から関数ポインタ変数へ、さらにはその変数に代入される可能性のあるすべての関数へと呼び出し関係をたどることによって潜在的なコールツリーを見ることができます。アナライザはひとつの関数ポインタから別の関数ポインタへの代入も追跡し、仮引数として渡される関数ポインタまたは関数を認識します。

この方法は、静的に初期化される関数ポインタ変数、関数ポインタの配列、または関数ポインタメンバを持つ構造体/共用体/クラスにとっては充分です。アナライザは代入も追跡します。ただし、変数が動的に更新される場合（すなわち、別の関数名がステートメントに代入される場合）、または関数ポインタ変数にたどりつくまでに関与するポインタがある場合、あるいは関数名が仮引数を通して渡される場合には、グラフは一般化し、呼び出される可能性のあるすべての関数及び関数ポインタを表示します。一部の if 文によって除外される関数を排除するためにコントロールロジックを評価しようとはしません。アナライザはまた、関数または関数ポインタを返す関数を処理することもできません。

次の例に、何が処理されるかを示します。

```
// 関数ポインタへの関係
```

```
int foo1();
int foo2();
int foo3();

typedef int (*fpT)();

fpT x1 = foo1;                    // 処理される: 関数ポインタへ直接代入
fpT *x4 = &x1, x2 = x1, x3;       // 処理される: 関数ポインタへのポインタ
int *v;

fpT a1[] = {foo1, foo2, foo3};  // 処理される: 関数ポインタの配列
fpT (*a2)[] = &a1;                // 処理される: 関数ポインタの配列へのポインタ
struct s1 {
fpT fp;
double w;
struct {
       int cnt;
       fpT fp2;
} si[3];
} s1 = {foo3, 2.0},                // 処理される: 関数ポインタを含む構造体
s2 = {foo2, 3.0, {{2, foo1}}}, // 処理される: 関数ポインタの配列を含む構造体
*sp1 = &s1;              // 処理される: 関数ポインタを含む構造体へのポインタ
struct s1 s3 = {foo1, 0.4, {{3, foo2},{4,foo3}}};  // 処理される

fpT goo1()
{
fpT r = &foo1;
if (v) return r;      // 処理されない: 戻り値
else   return foo2;  // 処理されない: 戻り値
}

void goo2(fpT &fp)
{
fp = foo3;            // 処理されない: 外部パラメータ
}

int bar1()
{
fpT locfp;
(*x1)();              // 処理される
goo1()();             // 処理されない
*v = 2;
(s1.fp)();            // 処理される
x3 = s1.fp;           // 処理される
*x4 = foo3;           // 処理される
(*a2)[1]();           // 処理される
goo2(&locfp);
(*locfp)();           // 処理される/処理されない
}

int bar2(fpT p, fpT q)
{
if (v) p = foo3;      // 処理される
if (*v) p = x1;       // 処理される
(*p)();               // 処理される
x1 = foo2;            // 処理される
x2 = sp1->fp;         // 処理される
s1.fp = foo1;         // 処理される
```

```
x3 = goo1();          // 処理されない
q();                  // 処理される
}

int bar3()
{
(*a1[*v])();          // 処理される
(*sp1->fp)();         // 処理される
bar2(foo1,foo2);      // 処理される
(*x2)();              // 処理される
(*x3)();              // 処理される
s2.si[0].fp2();       // 処理される
s3.fp();              // 処理される
(*s3.si[1].fp2)();    // 処理される
(**x4)();
}
```

なお、どの関数が関数ポインタによって呼び出される可能性があるかを指示すれば、Imagix 4D のデータベースにさらに多くの知識を追加することができます。そのためには、ひとつの方法として、そういう情報を含むデータファイルを生成する方法があります(*本ユーザガイド*の「データの追加」の項参照)。もうひとつは、ソースコードを修正する方法です。関数ポインタによってどの関数が呼び出されるかがわかっている場合には、次のようなソース行を追加しておけば、Imagix 4D がコールツリーを完成させることができます。

```
#ifdef IMAGIX_ONLY
FunctionPointerType fp = {func1, func2, ..., funcN};
#else
FunctionPointerType fp;  // fp の通常定義
#endif
```

また、構造体の場合には、関数はメンバで関連付けられることにも注意してください。したがって、構造体型が同じで完全に異なるふたつの構造体変数がいっしょにマージされます。同じ宣言クロスリファレンスは捕捉されません(たとえば、FPTR x = foo, y =x;)。

Microsoft Visual C++のサポート

タイプライブラリと#import

Microsoft Visual C++では、COM 型の定義をタイプライブラリに格納することができます。これらの定義は、Microsoft 固有のプリプロセッサディレクティブである#import を使用すれば、ソースコードで利用できます。例を示します。

```
#import <libraryname.tlb>
```

MSVC コンパイラは#import ディレクティブによっていくつかのファイルタイプをサポートしています。最も一般的なのがタイプライブラリバイナリ(.tlb)です。その他にサポートされているファイルタイプには、実行可能モジュール(.exe)、ダイナミックリンクライブラリ(.dll)、および以下のように progid または libid によって識別されるタイプライブラリなどが含まれます。

```
#import "progid:my.prog.id.1.5"
```

または

```
#import "libid:12341234-1234-1234-1234-1234" version("4.0") lcid("9")
```

Imagix 4D のソースアナライザが #import コマンドを処理するとき、コンパイラにより生成された対応する主要ヘッダファイルをインクルードしようとします。libraryname.tlb または libraryname.exe を指定している #import ディレクティブに遭遇すると、ソースアナライザは libraryname.tlh という名前のヘッダファイルを探します。ファイルが見つからない場合、アナライザは次の警告メッセージを生成します。

```
cannot open file "libraryname.tlh"
```

.tlh が検出されるためには、-I オプションおよび-S オプションによって Imagix 4D ソースアナライザに渡されたインクルードパスにそれが格納されている必要があります。したがって、そのようなメッセージを解決するには、まずファイルが存在するかどうかをチェックします。ファイルが存在しない場合、MSVC コンパイラを実行してファイルの生成を試みる必要があります。.tlb ファイルタイプに対し、コンパイラは通常、対応する.tlh ファイルを生成します。.exe や.dll ファイルなどの他のタイプについては、.tlh ファイルは作成される場合と作成されない場合があります。作成されない場合は、コードを Imagix 4D にロードする際に警告メッセージが残りますが、実際のところ.tlh ファイルは.exe ファイルには必要ないことがほとんどです。タイプライブラリ名の形式に progid および libid を使用すると、対応する.tlh ヘッダファイルが生成されないため、それらの名前は Imagix 4D ソースアナライザではサポートされていません。これに関して詳しくは、Microsoft のマニュアルを参照してください。

.tlh ファイルが存在するにもかかわらず警告メッセージが出力される場合は、-I オプションを追加してファイルが格納されているディレクトリを指定する必要があります。通常は、MSVC プロジェクトまたは MSVC ワークスペース/ソリューションのデータソースをロードする際、-I オプションおよび-S オプションが自動的に生成されます。追加の-I オプションは、Data Sources ダイアログの Project タブにある Options フィールドで指定することができます。

多くの場合、主要ヘッダファイル(.tlh)には#include ディレクティブが含まれており、コンパイラが生成したメンバ関数が定義されている 2 次ヘッダファイル(.tli)がそこに示されています。警告メッセージが.tlh ファイルではなく.tli ファイルが見つからないことを示している場合、.tli ファイルが存在するかを再度チェックし、MSVC コンパイラを実行して、ファイルの生成を試みる必要があります。多くの場合、見つからない.tli ファイルは Imagix データソースに情報を加えないため、無視しても構いません。

Imagix は警告が生成されていてもソースコードをロードすることに注意してください。#import ファイルが見つからないという警告以降にメッセージが報告されなければ、警告を無視して、ソースコードの解析が正常に完了したとみなすことができます。さらに、見つからないファイルに関する警告をなくしたい場合は、空のテキストファイルを適切な名前で作成し、そのファイルを通常のヘッダファイルとともにインクルードディレクトリの 1 つに配置します。

クラス参照と#using

Microsoft Visual C++では、他の実行モジュールに定義されているシンボルの使用をサポートするためのもう 1 つの手法が提供されています。#using プリプロセッサディレクティブは、.dll、.exe、.netmodule、および.obj の各ファイルに定義されたシンボルへの参照を可能にします。例を示します。

```
#using <mscorlib.dll>
```

Imagix 4D ソースアナライザが #using コマンドを処理する際、対応するヘッダファイルのインクルードを試みます。file.dll を指定する#using ディレクティブに遭遇すると、ソースアナライザは file.dll.inc というファイルを探します。ファイルが見つからない場合に生成される警告メッセージは以下のとおりです。

```
cannot open file "file.dll.inc"
```

#import ディレクティブの場合とは異なり、MSVC コンパイラは #using ディレクティブに遭遇したとき、自動的にヘッダファイルを生成しません。ユーザ自身がそのようなファイルを手動で作成し、得られたファイルを-S オプションまたは-I オプションで指定されたディレクトリに置く必要があります。

このように参照される最も一般的な MSVC 実行モジュールは mscorlib.dll です。Imagix 4D の環境にはこれに対応するヘッダファイルとして mscorlib.dll.inc が含まれています。このファイルはコンパイラ構成ファイルのディレクトリである../imagix/user/cc_cfg にあります。このディレクトリは常に暗黙的な-S オプションによって指定されます。したがって、#using <mscorlib.dll> という特定のディレクティブが既にサポートされています。この例にならって、その他の#using ディレクティブに対するサポートを追加することができます。

複数の DLL とクラスのエクスポート/インポート

コンパイラの標準的な動作では、グローバル命名規則が適用され、同一コンパイル単位（実行可能ファイル、DLL またはライブラリ）にある同じ名前を持つすべての最上位クラスは同じクラスであると見なされます。 Microsoft Visual C++もこのルールに従います。実行モジュールまたは DLL のそれぞれに対し Imagix 4D のプロジェクトを作成している限り、この動作はデフォルトの Imagix 4D ソースアナライザの動作でサポートされます。

しかし、複数のコンパイル単位にまたがったプロジェクトを定義したい場合もあるでしょう。そのような場合、別のコンパイル単位において偶然、実際には別々であるクラスに同じ最上位クラスの名前が再び使われてしまうことがあります。このような名前の衝突はソースアナライザのオプション、-uaggr（ユニークな集合）によって防止することができます。-uaggr が使用されると、異なるファイルに定義されたすべての最上位クラスは、たとえ同じ名前であっても別々のクラスであるとみなされます。

これによって最上位クラスの名前の衝突に関するほとんどの問題に対処できます。しかし Microsoft Visual C++では、最上位クラスをコンパイル単位の境界を越えて共有することが許されています。MSVC では拡張機能の dllexport および dllimport によって、特定のクラスを拡張 DLL として定義することが可能です。そのようなクラスは、1 つの DLL によって定義され、それらが使用される他の DLL 内で宣言されます。その一方で、通常のクラスは単にひとつの DLL として引き続き拡張されます。

Imagix 4D はそのような状況に対応することができます。通常のクラス同士の名前の衝突を除去するには、-uaggr オプションを使用しなければなりません。拡張機能の dllexport および dllimport によって指定されたクラスが DLL を拡張する単一のクラスとして見なされるためには、Imagix 4D のコンボプロジェクト機能を使用することが推奨されます。この技法では、各 DLL に別々の Imagix 4D プロジェクトが作成されます。その後、個々のプロジェクトがコンボプロジェクトに結合されます。

この技法のもう 1 つの利点は、1 つの DLL だけを調べたい場合に、結果として個々のプロジェクトがフォーカスされることです。そして、複数の実行モジュールと DLL にまたがったシステム全体を調べたい場合には、コンボプロジェクトによってそれが可能です。コンボプロジェクトに関して詳しくは、当ユーザガイドの「プロジェクトとデータ収集」の「コンボプロジェクト」の項を参照してください。

エラー処理とアナライザメッセージ

Imagix 4D アナライザは、まさにコンパイラのように完全な意味解析を行いますが、正規のコンパイラより寛容に設計されています。このため、コードは完全にコンパイル可能でなくても解析されます。

Imagix 4D アナライザには、言語問題についてのビルトインのエラー訂正規則があります。このような問題は不完全または構文的に不正確なソースコードによって生じると考えられます。しかし、たとえソースコードがコンパイル可能でも、ヘッダファイルの欠落、インクルードディレクトリの不正確な指定、あるいは前処理指令の評価に使用するマクロ定義のエラーによって問題が生じることもあります。

アナライザは、解析しようとしているソースコードのなかで構文または意味上の問題に出会うと、その問題を克服するためにエラー訂正規則を起動し、解析を続けます。もっと重大な問題の場合には、これらの規則では充分ではなく、アナライザは何行かをスキップして、あらためてソースコードと同期する必要があるかもしれません。

アナライザは、こうした問題が起こると、エラーメッセージを返します。メッセージは Analysis Results ウィンドウに表示されます。デフォルトでは、より重大なメッセージだけが表示されるようになっていますが、LOCAL メニューを設定すれば（[Display] > [Show All Messages]）、警告メッセージも表示されるようにすることができます。

特定のエラーの原因を解析する場合には、Analysis Results ウィンドウでそのエラーの上にマウスポインタを合わせて左クリックすると、ウィンドウがメッセージの表示からエラー解析の表示に切り換わります（LOCAL メニュー[Display] > [Show Error Analysis]）。Analysis Results ウィンドウには、解析上の問題の原因解明に役立つように、データベースとソースファイルの検索によって得られた詳細な情報が表示されます。

Java コードの解析

Imagix 4D は、みなさんのシステムの数多くのソフトウェア成果物から情報を収集することができますが、そのなかで最も重要なのはソースコードファイルそのものです。ソースコードの解析は、Imagix 4D のアナライザによって処理されます。C/C++ のアナライザについては前のセクションで説明してあります。ここでは Java のコードのアナライザについて説明します。

Imagix 4D の Java のアナライザは、コンパイラとよく似た動作をします。コンパイラのように、ソースファイルの完全な意味解析を行い、コード中のすべてのシンボルを解析します。ただし、コンパイラはリンカによって実行ファイルに組み込まれるオブジェクトファイルを生成するのに対し、Imagix 4D のアナライザは Imagix 4D のデータベースにロードされ、ほかのデータファイルと統合されるデータファイルを生成します。

Imagix 4D Java アナライザは Java 仕様の Second Edition および Third Edition をサポートします。アナライザは Java ソースファイル、クラスファイル、および jar ファイルの解析が可能です。また、アナライザは宣言の情報のみをクラスファイルから抽出し、メソッド本体や初期化子式は抽出しません。jar ファイルが入力として指定された場合、アナライザは jar ファイルにあるすべてのクラスファイルについて宣言の情報を抽出します。jar ファイル内の解凍済みまたは zlib 圧縮されたクラスファイルからも抽出されます。その他の圧縮方法はサポートされていません。

Imagix 4D のアナライザは、Imagix 4D のデータベース/ユーザインターフェイスから独立した実行ファイルです。これは Imagix 4D のユーザインターフェイスから起動するか、または個別にコマンドラインから起動できます。

アナライザの構文とオプション

Imagix 4D の Java のアナライザは、Imagix 4D のデータベース及びユーザインターフェイスから独立した実行ファイルです。/imagix/bin のディレクトリに置かれていて、次の構文を使用します。

imagix-java [*option...*] *file...*

ただし、option はひとつ以上の imagix-java オプション（以下に説明）、file はひとつ以上のソースファイルの完全パス名をさします。通常は、関係があるのは.java のソースファイルだけです。解析対象となる *file* のリストに直接クラスファイルと jar ファイルを含めることができますが、通常これらは、ソースコード内の import classname ディレクティブの結果として、および/または imagix-java 起動時の-cp クラスパスオプションの結果として解析されます。

Imagix 4D のアナライザは以下のオプションをサポートします。

-cmmd*Filename* *Filename* からコマンドを追加します。

imagix-csrc ソースアナライザはコマンドライン実行ファイルです。-cmmd オプションを使用することで、コマンドラインオプションを間接的に指定することができます。*Filename* で指定されたオプションは、コマンドライン自体で指定したオプションに追加されます。*Filename* が複数行で構成されている場合、imagix-java は *Filename* 内の次の行から、オプションと合わせて複数回起動されたように動作します。

-cp *Dirname* *Dirname* をクラスファイルを検索するクラスパスのリストに追加します。
-cp *Filename* *Filename* をクラスファイルを検索する jar ファイルのリストに追加します。

アナライザがソースファイルにおいて import classname ディレクティブに出会うと、-cp オプションで指定されたすべてのディレクトリ内および jar ファイル内で classname.class というクラスファイルを探します。

-cp の後に *Dirname* が続く場合、そのディレクトリはクラスファイル検索の基準ディレクトリとなります。classname 内のパッケージ名は *Dirname* クラスパスのサブディレクトリとして扱われます。例えば、*Dirname* クラスパスが/some/dir であり、import classname が pkgA.pkgB.classC である場合、アナライザは/some/dir/pkgA/pkgB ディレクトリ内で classC.class を探します。

-cp の後に *Filename* が続く場合、jar ファイル *Filename* にあるクラスファイルが解析されます。デフォルトでは、jar ファイル内にあるすべてのクラスファイルの内容についてのデータが生成されます。-impmin オプションを使用すると、ソースコードに import ディレクティブがあるクラスファイルについてのみ、データが生成されます。

ディレクトリおよび jar ファイルのコンテナは、-cp オプションの使用時にコマンドラインに表示される順序で検索されます。

-genflow*Dirname* ディレクトリ *Dirname* のファイルに制御フロー及びデータフローデータを書き込みます。

アナライザはデータを収集して、解析されたソースコードの制御フローとデータフローに関する表示およびレポートを生成することができます。このフローデータはエンティティ/関係/属性データが格納されているところとは別に、ファイルの集合に保存されます。-genflow オプションはこれらのフローデータファイルの場所を制御します。Imagix 4D がこのデータを検索できるようにするには、-genvdb データとディレクトリパスが同じ(-genvdb データと並列)である、cfd というディレクトリ内の場所を指定する必要があります。-genflow オプションが使用されていない場合は、フローデータは生成されず、フローチャート、計算ツリー、制御フローグラフ、及び特定のレポートが作成されません。

オプションは、データソースを追加するときに自動的にセットされます。

-genqcf*Dirname* ディレクトリ *Dirname* のファイルに品質管理データを書き込みます。

アナライザは、ソースコードチェックを実行し、解析されたソースファイルのシンボルに関するメトリックスを収集することができます。この品質管理データは、エンティティ/関係/属性データが格納されているところとは別に、ファイルの集合に保存されます。-genqcf オプションはこれらの品質管理データファイルの場所を制御します。Imagix 4D がこのデータを検索できるようにするには、-genvdb データとディレクトリパスが同じ(-genvdb データと並列)である、qcm というディレクトリ内の場所を指定する必要があります。-genqcf オプションが使用されていない場合は、品質管理データは生成されず、また特定のメトリックスやレポートが作成されません。

オプションは、データソースを追加するときに自動的にセットされます。

-genvdb*Dirname* ディレクトリ *Dirname* のデータファイルに解析結果を書き込みます。

アナライザは明示的に解析されるソースファイル、及び直接的または間接的にそれらのソースファイルに含まれるヘッダファイルの個々についてエンティティ/関係/属性データを生成します。例外として考えられるのは、システムインクルードディレクトリのヘッダファイルです(-nosys オプション及び-S オプションの項参照)。

-genvdb オプションが使用されているときには、個々のソースまたはヘッダファイルに関するデータはディレクトリ Dirname の独立したファイルに格納されます。このため、増分解析が可能になるので、これがおすすめの方法です。ほかには-o オプションを用いる方法があります。

オプションは、データソースを追加するときに自動的にセットされます。

-impcj インポートされたクラスを最初にクラスファイルで検索し、次に .java ファイルで検索します。

import class ディレクティブがソースコード内に出現すると、アナライザはそのクラスの定義を探します。-impcj オプションが使用されると、アナライザは最初に -cp オプションによって位置を特定されたクラスファイル内を検索します。クラス定義の場所がクラスファイルに見つからない場合は、アナライザは解析対象の java ソースファイルに出現するクラス定義を調べます。

-impcj、-impco、および-impjc の各オプションは互いに排他的です。オプションは Data Sources ダイアログの Class Paths タブにある Import モードコンボボックスで選択されます。

-impco インポートされたクラスをクラスファイルでのみ検索します。

import class ディレクティブがソースコード内に出現すると、アナライザはそのクラスの定義を探します。-impco オプションが使用されると、アナライザはクラスファイルの中だけを検索します。アナライザは解析対象の java ソースファイルに出現するクラス定義をすべて無視します。

-impcj、-impco、および-impjc の各オプションは互いに排他的です。オプションは Data Sources ダイアログの Class Paths タブにある Import モードコンボボックスで選択されます。

-impjc インポートされたクラスを最初に.java ファイルで検索し、次にクラスファイルで検索します。

import class ディレクティブがソースコード内に出現すると、アナライザはそのクラスの定義を探します。-impjc オプションが使用されると、アナライザは最初に解析対象の java ソースファイルを調べます。クラス定義の場所が java ファイルに見つからない場合は、アナライザは-cp オプションによって位置を特定されたクラスファイル内でクラス定義を検索します。

-impcj、-impco、および-impjc の各オプションは互いに排他的です。オプションは Data Sources ダイアログの Class Paths タブにある Import モードコンボボックスで選択されます。

-impmin jar ファイル内のインポートされていないクラスに対してデータを生成しません。

jar ファイルは-cp オプションによってアナライザに識別されます。通常は、jar ファイル内のすべてのクラスファイルに関するデータはアナライザによって収集されます。-impmin オプションが使用されると、解析対象のファイルに import class ディレクティブがあるクラスファイルに関してのみ、データが生成されます。

-impwopkg import ディレクティブに対するパッケージ名を必要としません。

import ディレクティブを処理する際、ソースアナライザは完全なインポート名を持つ java ファイルまたはクラスファイルについて、パッケージ部分も含めて検索しようとします。-impwopkg オプションが使用されている場合に初回の検索が失敗すると、ソースアナライザは java ファイルまたはクラスファイルの検索においてパッケージ部分を無視します。

-inc*Filename* 各ファイルの解析開始時に *Filename* をインクルードします。

-inc オプションは、解析する各ソースファイルの最初に import *Filename* ディレクティブを追加したときと同じ効果があります。つまり、ファイル自体を修正せずにファイルの解析方法を変更することができます。

このオプションは主として、言語設定ファイルをインクルードするために使用されます。言語設定ファイルには通常、ご使用の環境の jar ファイルに対するクラスパスコマンドが含まれており、Imagix 4D アナライザはコード解析の際にご使用の Java 言語環境をエミュレートすることができます。

Filename は Data Source ダイアログの Source Files タブにある Language コンボボックスで、../imagix/user/java_cfg 内のファイルから選択されます。

-java4 Java 1.4 構文を使用して、ソースコードを解析します。

Imagix 4D Java アナライザのデフォルトの動作では、Java の 1.5 バージョンの構文に従ってソースコードを解析します。このオプションが適用されている場合、旧式の構文に基づいてソースコードが解析されます。

オプションは、Data Sources ダイアログの Source Files タブにある Options フィールドで追加するか、言語設定ファイルで指定することができます。

-locals ローカル変数に関するデータを収集します。

通常、Imagix 4D アナライザはグローバル変数及び静的変数に関するデータだけを収集し、ローカル変数は定義されているところで関数によってセットされるか、読み取られるだけです。つまり、ほとんどソフトウェア理解の助けになりません。しかし、場合によっては、ローカル変数に関するデータを収集したいと思うこともあるでしょう。たとえば、クラスカップリングのようなある種の品質メトリックスをしようと思っているときです。-locals オプションはローカル変数の定義及び使用法に関するデータをデータファイルに追加させます。

オプションは、Data Collection Options ダイアログで解析オプションのひとつとして指定します。

-mark あらゆるアナライザメッセージに”imagix:”を挿入させます。

アナライザのエラーメッセージは、通常、それが発生したファイルの名前からはじまります。-mark オプションを使用すると、すべてのエラーメッセージの先頭に文字列”imagix:”が挿入されます。これは、メイクファイルでアナライザを実行しているときのようにアナライザのエラーメッセージが標準エラー出力（stderr）への他のメッセージと混ざっているときには便利です。

オプションは、Data Sources ダイアログの Source Files タブにある Options フィールドで追加するか、言語設定ファイルで指定することができます。

-sfx -genflow 及び-genqcf データに関連付けられたファイル名に接尾語を付けます。

通常、-genflow 及び-genqcf スイッチで生成されるデータは、名前の最後が対応するオリジナルのソースファイル及びヘッダファイルと同じ接尾語で終わるファイルに格納されます。このため、一部の設定管理システムで障害が生じる可能性があります。-sfx オプションは、.java ソースファイルから生成されたデータが .java_sfx の接尾語を持つファイルに格納されるように、ファイル名の最後に_sfx を追加します。

オプションは、Data Sources ダイアログの Source Files タブにある Options フィールドで追加するか、言語設定ファイルで指定することができます。

-vuid 解析する各シンボルについて VUID を生成します。

プロジェクトに関するドキュメントを作成する場合、通常は 1 つのパスにすべてのドキュメントを置きます。この場合、データベースがロードされると、ハイパーテキストリンクで使用されるシンボル識別子が割り当てられます。ドキュメントは 1 つのセッションで作成されるため、各 HTML ページで同じ識別子が使用されます。

-vuid オプションでは、プロジェクトがロードされたときではなく、ソースコードの解析時にビジュアライザ固有の識別子が生成されます。これによりプロジェクトセッション間の識別子の整合性がとられ、部分的及び増分の HTML ドキュメント作成が可能になります。部分的及び増分のドキュメント作成の詳細については、ファイル../imagix/user/doc_gen/sample.dg_を参照してください。

オプションは、Data Sources ダイアログの Source Files タブにある Options フィールドで追加するか、言語設定ファイルで指定することができます。

-warn すべてのアナライザ警告を返します。

ソースコードの解析中、Imagix 4D アナライザはソースコードそのもの、あるいはコード解析に使用されるマクロ定義及びインクルードパスによって生じた構文または意味エラーに出会うことがあります。

デフォルトでは、アナライザは、インクルードファイルが見つからない、あるいはアナライザを再同期するためにコードを何行かスキップする必要があるといった重要な問題についてのみエラーメッセージを返します。-warn オプションを使用すると、アナライザは出会う問題すべてについてメッセージを返します。

場合によっては、あとで解析上の重大なエラーを引き起こす問題を、初期のうちに隔離するために-warn オプションを使用してもよいでしょう。

オプションはユーザインターフェイス全体でつねに有効になっています。警告メッセージは、Show Warning Messages チェックボックスに従って（LOCAL メニュー［Display］>［Show Warning Messages］）Analysis Results ウィンドウで表示またはフィルタされます。

言語構成ファイル

Imagix 4D java アナライザは環境から独立しています。ソフトウェアを環境において動作しているときと同じように解析するには、Imagix 4D にソースコードで使用するものと同じクラスパスをセットアップする必要があります。

クラスパスのアプリケーション固有部分は Data Sources ダイアログで指定するのが最適ですが、言語構成ファイルはお使いの環境での標準のクラスパスを指定する方法を提供します。構成ファイルは./imagix/user/java_cfg にあります。Data Sources ダイアログにおいて、特定のプロジェクトでどの構成ファイルを使用するかを選択することができます。

アナライザの起動

Imagix 4D Java アナライザは独立した実行ファイルであり、アナライザの構文に従い、コマンドラインを介して起動することができます。ただし、操作を簡単にするため、通常、起動は Imagix 4D ユーザインターフェイスの内部から制御されます。

Imagix 4D ツールは、カレントプロジェクトに関する知識を利用して、生成されるデータファイルの格納場所に関するアナライザスイッチをセットします。残りの必要な情報――解析するソースファイル、およびクラスファイルと jar ファイルのクラスパス――は Data Sources ダイアログ（メニュー［Project］>［Data Sources］）で指定します。ダイアログの左側で選択された各データソースについて、特定のデータソースの設定はダイアログの右側で行います。右側上部の Type メニューボタンで、メニューの Source Files セクションにあるメニュー項目のいずれかによって Imagix 4D Java アナライザが起動されます。ダイアログを完成させる方法については、このマニュアルの「*はじめに*」の項、及び状況連動型ヘルプ画面を参照してください。

Options ダイアログ（メニュー[Tools] > [Options] >［Data Collection］）の設定はローカル変数に関するデータを収集するかどうかの選択など、Data Sources ダイアログの各 Options タブに表示されるアナライザオプションのデフォルト設定のために使用します。

Java - ダイアログを通して解析を行う

この方法では、アナライザは Data Sources ダイアログの設定に従い、ユーザインターフェイスから直接起動されます。ダイアログでは、解析するソースファイル、検索するクラスファイルの場所、解析する jar ファイルなどを指定します。

このダイアログは、ある特定のディレクトリのソース(.java)ファイル、あるいは、ある特定のディレクトリ及びそのすべてのサブディレクトリのソースファイル解析をサポートします。ソフトウェアが複数のディレクトリにまたがっている場合には、各ディレクトリにアナライザを複数回、起動する必要があります。その場合には、そのつど、Data Sources ダイアログでデータソースを追加することになります。

ただし、このディレクトリの問題はクラスファイルには適用されません。インポートされたクラスファイルの検索場所は Class Paths タブで指定します。1 つは-cp *dirname* を'Additional -cp flags'フィールドに挿入する方法があります。これは任意の数のディレクトに対して行うことができます。

また、Class Paths タブを使用して、解析に含める jar ファイルの場所と名前を指定することができます。複数のプロジェクトの設定を簡略化するために、日常的に使用される jar ファイルは言語設定ファイルに指定し、Class Paths タブはプロジェクト固有の jar ファイルを解析に追加するために使用すべきです。

プロファイルデータ - tcov、gprof、プロファイラ

Imagix 4D によって収集され、表示される情報の多くはソースコードの静的解析に基づくものですが、このツールは C/C++ソフトウェアの実行時間パフォーマンスについてもある種の情報を収集し、表示することができます。プロファイルデータソースから収集されるこれらの実行時間パフォーマンス測定値は、コードに含まれる個々の関数と関連付けられた個別のメトリックスとして表示することができます。また、Structure ビューの Graph ウィンドウでは、それらをまとめてビューすることもできるので、プログラムの全体的な呼び出し構造のコンテキストのなかでの個々の関数及び呼び出しの実行時間パフォーマンスを見ることができます。

プロファイルデータソースのプロジェクトへのインポートのプロセスは、必要なデータソースの判断、ソースコードの適切なコンパイル、実行ファイルの実行、データソースのロードから成ります。オプションとして、Imagix 4D を用いてプロファイルデータソースを管理し、それによって増分の測定をサポートすることもできます。

プロファイルデータのソース

Imagix 4D は C/C++ソースコードに対して、いくつかの実行時間プロファイルデータのソースをサポートします。Unix の実行ファイルについては、最もネイティブな Unix コンパイラを使用して tcov 及び gprof データベースを生成することができます。Windows の実行ファイルについては、.NET より前のバージョンの Microsoft Visual C++コンパイラを通して生成されたプロファイラデータベースを受け付けます。実行ファイルを実行することによってこのデータベースが埋まり、実行ファイルが実行されるプラットフォーム上で一部の初期の事後処理が行われると、Imagix 4D は Unix 環境下か Windows 環境下かに関係なく、tcov、gprof、またはプロファイラデータをロードすることができます。

生成及びロードするデータソースは、使用するメトリックスによって異なります。

Coverage（カバレッジ）

個々の関数がどれだけ完全に実行されたかを表示します。

Unix 実行ファイル - tcov 必要、gprof はオプション
Windows 実行ファイル - プロファイラ（ラインカバレッジまたはラインカウンティング型）必要

Time（実行時間）

個々の関数に費やされた時間を表示します。

Unix 実行ファイル - gprof 必要
Windows 実行ファイル - プロファイラ（関数タイミング型）必要

Frequency（実行頻度）

個々の関数が実行された頻度を表示します。

Unix 実行ファイル - gprof 必要
Windows 実行ファイル - プロファイラ（任意の関数型）必要

Unix 実行ファイルのプロファイルデータを有効にするコンパイル

tcov と gprof の両データソースの場合にも、ソースコードをコンパイルするときに適切なオプションをセットする必要があります。それではじめて、コンパイラは実行可能コードをツール化することにより、また、プ

ロファイル結果を捕捉するファイルを生成することにより、プロファイルデータを捕捉する態勢ができます。tcov の場合には、各ソースファイルに別々の結果(.d)ファイルがありますが、gprof の場合、ファイルは実行ファイル全体でひとつです(gmon.out)。

通常、対応する使用オプションは次の通りです。

tcov

C コンパイラ -xa
C++ コンパイラ -a または -xa

gprof

C コンパイラ -xpg
C++ コンパイラ -pg または -xpg

これらのコンパイラオプションの一部は、-g オプションがセットされていると、セットすることができません。コンパイラフラグの詳細な使用方法については、コンパイラのドキュメントを参照してください。

Windows 実行ファイルのプロファイルデータを有効にするコンパイル

.NET より前のバージョンの Microsoft Visual C++でプロファイラ情報を生成するには、プロファイリングを有効にしてプロジェクトをビルドする必要があります。プロファイリングを有効にするには、プロジェクトの設定ダイアログのリンクタブの一般カテゴリでプロファイルを行うチェックボックスにチェックを入れます。

前記した 2 つのライン型プロファイラ情報のいずれかを生成したい場合には、適切なデバッギングも有効にして実行ファイルをビルドしなければなりません。プロジェクトの設定ダイアログのリンクタブの一般カテゴリでデバッグ情報を生成するチェックボックスにチェックを入れ、同ダイアログの C/C++タブの一般カテゴリで行番号のみまたはプログラムデータベースを使用をデバッグ情報の型として選択します。

詳しくは Microsoft Visual C++のユーザガイドを参照してください。

プロファイルデータの生成

適切なオプションをセットして再コンパイルしたら、ソフトウェアを実行して Imagix4D が調べるプロファイルデータを生成する必要があります。

このときには、調べたいテストケースを使用します。実行ファイルを実行すると、結果は自動的に.d 及び gmon.out ファイル(Unix)または.pbt ファイル(Windows)に捕捉されます。

Imagix 4D が、実行ファイルを実行したプラットフォーム上で動作している場合、結果のデータファイルから直接プロファイルデータを収集することができます。その際には、データファイルの処理を助けるために gprof または Microsoft Visual C++ユーティリティの一部など、そのプラットフォーム上で使用可能なネイティブツールを利用します。

なお、プロファイルデータのクロスプラットフォーム解析を行いたい場合も、Imagix4D は、そのデータをロードすることができます。この場合には、実行ファイルのネイティブプラットフォーム上でランタイムデータファイルに幾つかの初期処理を行う必要があります。それにより、Imagix 4D は動作しているプラットフォーム上で中間結果をインポートすることができます。

必要な事後処理は、生成したプロファイルデータの型に依存します。適切なコマンドをコマンドラインから投入することができます。

gprof(gmon.out)の場合

```
gprof -b executable gmon.out > results_file
```

プロファイラデータ（executable_rootname.pbt）の場合

```
plist /t executable_rootname > results_file
```

tcov データ（.d ファイル）の場合には、事後処理は必要ありません。詳しくは gprof マニュアルページまたは Microsoft Visual C++のユーザガイドを参照してください。

プロファイルデータのインポート

データソースは Data Sources ダイアログ（メニュー［Project］> ［Data Sources］）を使用してロードすることができます。

データソースをロードする順序には、制約があります。プログラム構成情報はプロファイル情報がロードされる前に、Source Files による解析方法のいずれかを使用して追加されなければなりません。Imagix 4D はソフトウェアの関数及び呼び出し階層について維持する情報の上にプロファイル情報をビルドするからです。このため、たとえば、tcov のファイル関連の結果をより意味のある関数関連の結果に変換することができます。これにより、各関数のプロファイル結果をメトリックス表示で値としてビューすることもできます。

gprof またはプロファイラデータをインポートしようとしていて、しかも実行ファイルを実行したプラットフォーム上で Imagix 4D を動作させている場合には、プロファイルデータのロードの仕方は、ふたつあります。生データをロードしたい場合には、gprof Data Files または MSVC Profile Data File を使用します。すでに事後処理を行っている場合には、gprof Result File または MSVC Profile Result File を使用します。クロスプラットフォーム開発を行っている場合には、選択肢は自動的に事後処理された結果をインポートする方法だけに限られます。

Data Sources ダイアログの詳細を理解するには、状況連動型 Help 画面を起動してください（F1 キー）。

プロファイルデータファイルの管理

Manage Profile Data ダイアログの機能を利用すれば、一連のテストケースについて実行ファイルをプロファイルし、それぞれの実行結果を保存し、関心のある実行の結果、またはその組み合わせをビューすることができます。

これにより、さまざまなプロファイリングツールを扱う複雑さをある程度排除することができます。多重（またはインクリメンタル）実行の場合、tcov、gprof、及び MSVC プロファイラデータコレクタの動作は違ってきます。Gprof、プロファイラでは、実行ファイルを実行するたびにデータファイルがリセットされます。このため、gmon.out、executable.pdt には、いつでも直前の実行のプロファイルデータだけが含まれます。tcov の.d ファイルはリセットされません。実行ファイルの多重実行を行うと、その結果は累積します。

Manage Profile Data ダイアログには、テストケースを実行する前に tcov データファイルをすべて再初期化する機能があります。これを利用すると、tcov データファイルに以後のテスト実行の結果のみを反映させることができます。このため、カバレッジを個々のテストケースごとに関連付けて追跡し、解析することができます。ダイアログには、このほか、プロファイルデータまたは結果ファイルの現在の内容をセーブする機能もあります。

これらの機能を利用すれば、連続してテストケースを実行し、各実行の前に tcov データを再初期化し、それぞれの実行後に結果を記録することができます。その上で、Manage Profile Data ダイアログを利用し、見たい結果を指定して見ることができます。保存されている実行の結果は、個別でも、また、まとめても見ることができます。

これらの機能はさまざまな形で利用することができます。特定のソフトウェアの変更がコードのパフォーマンスに及ぼす影響を解析したい場合には、そのソフトウェアの異なるバージョンについて同じテストケ

ースの結果を比較すればよいでしょう。また、一連のテストケース全体のカバレッジをまとめて見れば、テストの完全さを解析することができるでしょう。または特定のテストケースによるインクリメンタルカバレッジを決定することで、テストを最適化することができます。

関係のビルド-メイクファイル（Unix のみ）

メイクファイルを用いてソフトウェアをビルドする場合には、そのなかにプロジェクトデータベースに追加したい情報を含めます。とりわけ、この情報はどのソースファイルをコンパイルすることによってどのオブジェクトファイル（.o または.obj）をビルドするか、またどのオブジェクトファイルをリンクすることによってどの実行ファイルをビルドするかを指示します。

この情報はメイクファイル解析を通して収集することができます。メイクファイル解析は、Data Sources ダイアログ（メニュー［Project］> ［Data Sources］）で Makefile Contents 型のデータソースを追加することによって指定します。make コマンドによってより高いレベルのメイクファイルからより低いレベルのメイクファイルを呼び出す場合には、通常は最高レベルのメイクファイルでデータ収集を指定するだけで充分です。

メイクファイルに含まれるコンパイラ及びリンカ依存関係は、Imagix 4D のビューでは Depends On 関係として記述されます。これらの Depends On 関係は Includes 関係を補完し、ソースコードの解析によって自動的に生成されます。Includes 関係は、どのヘッダファイルがどのソースまたはヘッダファイルに含まれるかを示します。Graph ウィンドウで Depends On または Includes 関係をビューするには、Files のシンボル型を見えるように指定する必要があります。

メイクファイル解析は、メイクファイルに使用する make コマンドのバージョンによって決まります。Imagix 4D はダイアログ（メニュー[Tools] > [Options] > ［Data Collection］）で make 呼び出しを定義し、make、gmake などのうち、どの make コマンドを使用するかを決めます。メイクファイル解析は Unix 版 Imagix 4D で使用可能であり、プラットフォームのネイティブバージョンの make 及び GNU バージョンの make をサポートします。

データの追加 - vdb ファイル

一般に、Imagix 4D データベースは Imagix 4D データコレクタによって生成されたデータファイルを使用します。ただし、ソフトウェアに関する手持ちのデータで、1）Imagix 4D データコレクタによってまだ捕捉されておらず、2）プロジェクトのコンテキストのなかで見る価値のあるものがあれば、自分で生成したデータファイルをインポートすることもできます。個々のデータファイルには、多少に関係になく、好きなだけデータを結合することができます。Imagix 4D の使用フォーマットに対応しているデータファイルであれば、Imagix 4D によってインポートすることができます。

追加したいデータを含むファイルを生成したら、Data Sources ダイアログ（メニュー［Project］> ［Data Sources］）で Vdb File の型を選択し、ファイルの完全パス名を指定します。パス名に制約はありません。これで、指定したファイルのデータがデータベースにロードされ、ユーザインターフェイスを通して見られるようになります。Update Project Data または Regenerate Project Data の機能を実行したときには、そのファイルはただ再び読み取られるだけです。変更があれば、自分でしなければなりません。

一般に、データファイルを生成するケースとしては、関数ポインタに関する情報を追加する場合、アセンブラコードまたは FORTRAN など、非 C/C++コードへのブラウジングについての情報を追加する場合などが考えられます。例は以下に掲げます。エンティティ/関係/属性型が定義されている../imagix/data/vdb/schema.vdb を参照することができます。Imagix 4D のデータモデル及びそのフォーマットについてさらに詳しい情報は、Imagix Toolkit で提供されます。

アセンブラコード

Imagix 4D では、FORTRAN、アセンブラなど、他言語で実装されたコードとともに使用される C/C++および Java のソースコードを調べることができます。

Imagix 4D アナライザはまさにその C/C++および Java のソースファイルを、それがインクルードまたはインポートしている関連ファイルとともに処理します。他言語を用いた他のファイルに実装された関数の呼び出しは、ライブラリ関数の呼び出しと同様に処理されます。関数の宣言、及びそれが呼び出されるあらゆるインスタンスを調べることができますが、関数そのものの定義、及び関数の下の階層を見ることはできません。

設定が適切に行われていれば（本ユーザガイドの「言語拡張」の項参照）、Imagix 4D C/C++アナライザはキーワード asm または_asm を通して識別されたインラインアセンブラを処理します。アナライザはアセンブラコードをスキップし、アセンブラブロックのエンドで解析を続けます。

アセンブラコードは通常、呼び出し階層のボトムでパフォーマンスがきびしい関数を実装するために使用されるので、これで普通は充分です。ただし、C/C++ソースコードから呼び出された関数のために実際のアセンブラコードにブラウズしたいと思うこともあるでしょう。その場合には、ブラウズに必要な情報を含むデータファイルを生成し、そのデータファイルを Imagix 4D にインポートすることができます。

次のようなアセンブラコードを考えてみましょう。

```
PFPROC        TRAP08
              PUSHF
              CALL    DWORD PTR CS:[OLD08]
              CLI
              PUSHM   <ES,BX>
              LES     BX,CS:InDOS
              CMP     BPTR ES:[BX],0
              POPM    <BX,ES>
              IRET
ENDPROC
```

```
PFPROC        TRAP28
              MOV    CS:TryPop,FALSE
              MOV    CS:InPop,TRUE
              STI
              CALL   START
              CLI
              MOV    CS:AllowPop,TRUE
              MOV    CS:InPop,FALSE
              JMP    DWORD PTR CS:OLD28
ENDPROC
```

ブラウズを有効にするには、アセンブラコードで関数定義が出現する場所がわかるようにファイル及び行番号を与える必要があります。TRAP08 及び TRAP28 が C コードで それぞれ_asm_Trap08 及び _asm_Trap28 として呼び出され、次のような行で構成される.vdb ファイルを作成するとします。

```
GD /#root/usr/examples
GD /#root/usr/examples/asm
GF /#root/usr/examples/asm/foo.asm
SP /usr/examples/asm
</#root/usr/examples/asm/foo.asm
Gf ./_asm_Trap08
Sl 2
Gf ./_asm_Trap28
Sl 13
>
```

1 行目は、新しいシンボルが生成（G）されること、そのシンボルがディレクトリ（D）であること、そのシンボルがカレントシンボル/#root/usr に含まれること、また、そのシンボルの名前が examples であることを示します。2 行目は、次のディレクトリレベル、asm について同じことを繰り返します。

3 行目も、シンボルがファイル（F）で、foo.asm という名前であることを除けば、同じことを繰り返します。

4 行目は、カレントシンボル foo.asm のパス（P）の属性を/usr/examples/asm にセット（S）します。

5 行目は、<で始まり、新しい置換セクションが始まること、そのセクションがシンボル /#root/usr/examples/asm/foo.asm に焦点を当てることを示します。/#root はパスの始まりを意味し、あとに通常のパス名が続きます。Windows では、/#rootc:/examples/func_ptr/foo.c のようなパスを使用することもできます。

6 行目は、新しいシンボルが生成（G）されること、そのシンボルが関数（f）であること、そのシンボルがシンボル foo.asm のカレントセクション（./）に含まれること、また、新しいシンボルの名前が_asm_Trap08 であることを示します。

7 行目は、カレントシンボル_asm_Trap08 の行番号（l）属性を 2 にセット（S）します。つまり、_asm_Trap08 はどのファイルにしろ、それが定義されるファイルの 2 行目で定義されます。

8 行目と 9 行目は 6 行目と 7 行目の繰り返しで、次の関数の定義及び行番号を指定します。

10 行目で現在のセクション foo.asm が終了します。

データの更新

プロジェクトにデータソースを追加すると、ソースコードが変更され、ソフトウェアと Imagix 4D データベースとの同期が外れることがあります。その場合には、行番号の変更により、File Editor ウィンドウのシンボルの色が消えるなどの症状が現れます。プロジェクトのデータベースを再びソースコードに同期させるときには、データベースの更新をソースコード全体にするか、変更された部分だけにするかを選べます。さらに、プロジェクトの情報を変化するソフトウェアの現実の状態とカレントに保つ方法にも、手動と自動の両方があります。

これらの方法は選択して組み合わせることができます。たとえば、日中に Update Project Data 機能を手動で呼び出して、コードに対するインクリメンタルな編集を捕捉し、夜間の cron ジョブとしてすべてのデータ収集コマンドを自動的に実行することで、データベース情報を完全に再構築することができます。

インクリメンタル ／ 完全更新

データソースが変更された場合にのみデータを再収集するインクリメンタルデータ収集は、メニューで［Project］を選択し、Update Project Data の機能を通して使用可能です。

ソースコードデータの場合には、このデータの再収集方法ではややきめが粗くなります。Java でのダイアログを使用した解析、C/C++でのダイアログを使用した解析、MSVC Project による解析方法または MSVC Workspace/Solution による解析方法を用いて収集されたデータソースの場合、Update Project Data を使用すると、データは前回の収集後に修正されたソースファイル、あるいはインクルードファイルが修正されたソースファイルについてのみ再収集されます。たとえば、ひとつのヘッダファイルにのみ変更が加えられたとすると、データはそのヘッダファイルを直接的または間接的に含む C/C++ソースファイルについてのみ収集されます。メイクファイルを使用した解析方法の精度は、Data Collection Options ダイアログの設定と、メイクファイルの構造によって異なります。

その他のデータソースにとっては、このデータの再収集法でもそれほどきめが粗いとはいえません。たとえば、プロジェクトデータソースのひとつが Makefile Contents の場合には、前回 Update Project Data または Regenerate Project Data を起動したとき以降にそのメイクファイルが変更されている場合にのみ、それが再び読み込まれます。

Source Files およびその他のデータソースについて少しでもデータが再収集されれば、Profile Data のデータソースが再びロードされます。

Regenerate Project Data はすべてのデータを再収集させます。これにより、プロジェクトは確実にあらゆるデータソースの最新の状態を捕捉し、データファイルから廃棄されたデータを排除することができます。ただし、すべてのデータソースをあらためて解析するので、Update Project Data より時間はかかります。

手動 ／ 自動更新

手動で Imagix 4D プロジェクトデータを更新するときは、Update Project Data 及び Regenerate Project Data の機能を使用します。これらはメニューで［Project］を選択すれば、使用可能です。また、データソースはリコンパイルまたは夜間ビルドのプロセスの一部として自動的に再収集することもできます。これを利用すれば、誰もいないときにオフラインで再収集を行うことができます。

コマンドラインからデータの再収集を行う方法はいくつかあります。その一部は、適切な Imagix4D アナライザをコマンドラインから実行ファイルとして直接起動する方法です。

これはまさに、imagix-csrc アナライザを起動するメイクファイルにターゲットを追加し、メイクファイルを通して C / C++コードの解析を行う方法と同じです。メイクファイルを通して解析を行う方法を選択している

場合には、imagix 及び imagix_update の形式のメイクファイルにいくつかのターゲットがあります。これらがそれぞれ、コードの完全な再解析及びインクリメンタルな再解析をもたらします。

この方法では、cron ジョブまたは夜間ビルドスクリプトに make -f makefile imagix のようなコマンドを追加します。

メイクファイルを通して解析を行う方法を使用していない場合でも、コマンドラインから直接 Imagix 4D アナライザを起動することができます。

しかし、Imagix 4D 自体を起動するために-cmmd オプションを使用して自動的にデータ収集を行うのがおそらく最も簡単な方法でしょう。この方法では、Imagix 4D はバッチモードで起動して動作し、指定された一連のコマンドを実行した上で、終了します。Imagix 4D にひとつ(一連)のプロジェクトを開き、すべてのプロジェクトデータを再生成させるコマンドのセットを指定することができます。これにより、プロジェクトデータは再びソフトウェアと同期します。

この方法では、cron ジョブまたは夜間ビルドスクリプトに imagix -cmmd cmmdfile のラインからコマンドを追加します。バッチモード、及び cmmdfile のフォーマットの詳細については、本ユーザガイドの *付録*を参照してください。

プロジェクトリソース

データ収集結果をロードして解析を実行するために、Imagix 4D は一連のデータベースを使用します。そのうちいくつかは、特定の解析が要求されているときに、要求に応じてロードされます。パフォーマンスを最も速くするために、Imagix 4D はデータベースをメモリにロードします。任意指定で、非常に大きいプログラムを解析する際のメモリオーバーフローを避けるために、データベースの一部をディスクの記憶領域に格納することもできます。データベースリソースの拡張のためにディスクの記憶領域を使用すると、速度が犠牲となり、数倍（2 倍から 20 倍）減速することがあります。

このようなメモリと速度のトレードオフは、Options ダイアログ(メニュー [Tools] > [Options...] > [Data Collection] > [Project Resources]) の設定により制御されます。さらに厳密に制御するには 4Ddefaults ファイルを使用します。(詳しくは../imagix/user/sample.4Ddefaults を参照してください。) このトレードオフは Imagix 4D のセッションの間継続します。設定変更を反映させるには、変更後に Imagix 4D をクローズして再オープンする必要があります。

コアエンティティと関係情報は常にメモリに保存されます。この情報は、ファイル、クラス、関数グラフ、相互参照情報、ソースブラウジングなどの、Imagix 4D の解析と結果に関する表示の大部分の基礎をなすものです。コアデータ用のメモリの必要量は、コード内でのシンボル(エンティティ)の数と用途によって異なります。Imagix 4D では通常、400 万から 1000 万までの文を含むソースコードを 1 つのプロジェクトで処理することができます。Imagix 4D が最初のプロジェクトロードでメモリを使い果たした場合、コアデータのサイズを減らす 1 つの方法は、ローカル変数に関する情報の収集を無効にすることです。それを行うと、実行可能な解析のいくつかが制限されます。特に、Flow Check レポートの一部はローカル変数のデータを必要とします。必要メモリを減らす 2 番目の方法は、ソースコードを分割し、個別に解析可能な別々のプロジェクトを作成することです。

メトリックスとソースチェック情報は第 2 のデータベースに保存されます。コアデータとは異なり、メトリックスデータは要求に応じてロードされます。デフォルトでは、結果となるメトリックスデータベースはメモリにロードされます。要望ベースで実行されるため、データサイズはどのメトリックスが要求されているかに依存します。一般的には、データサイズはおおよそコアデータの 1-2 倍のサイズであると想定されます。したがって、100 万から 500 万までの文を含むソースコードを 1 つのプロジェクトで処理できることになります。メモリの必要量は、Project Resource 設定により削減できます。Large または Very Large に設定すると、プロジェクトに 300 万から 800 万の文が含まれる場合もあります。これを削減するには、ディスクデータベースを使用します。メモリの代わりにディスクを使用すると、メトリックスデータの初期計算において 2-10 倍減速します。Very Large に設定されている場合、減速するのは当該プロジェクトに対して 1 度限りです。2 回目にメトリックス情報がプロジェクト用にロードされる際には、メトリックスは既に生データから計算済みであるため、実際にロード時間はメモリから読み込む場合よりも速くなります。

Flow Check レポートと Calculation Tree に使用されるデータフローデータは、第 3 のデータベースに保存されます。メトリックスデータと同様、データフローデータは要求に応じてロードされ、計算されます。基礎的なグローバルデータフロー解析は非常に複雑であり、相当な時間とメモリを必要とします。データフローデータのサイズは、非ローカル変数の数、関数と再帰の数、サイクロマチック複雑度などの要因によって、コアデータの 5 倍から 20 倍になります。したがって、Imagix 4D のプロジェクトは、メモリが集中的に使用される場合にデフォルトの構成で 20 万から 100 万の文を含むソースコードを処理できることになります。そのような規模のプロジェクトは、解析に何時間もかかる可能性があります。Large または Very Large に設定されていると、解析が 5 倍から 20 倍減速する可能性がありますが、プロジェクトの規模を 100 万から 500 万の文にまで増やすことができます。この規模のプロジェクトには、解析に何十時間もかかる可能性があることに注意してください。

Imagix 4D の使い方

Imagix 4D は、複雑であるか、サードパーティ製であるか、あるいは旧式である C、C++、Java のソフトウェアの解析、ドキュメント作成、および改善に役立ちます。コードのブラウジングと解析を自動化し、大規模なプログラム、複雑なプログラム、未知のプログラム、古いプログラムを、素早く理解することができるようになっています。このツールを利用すれば、高レベルのアーキテクチャから個々の機能の詳細なプログラムロジックまで、あらゆるレベルでのソフトウェアの素早いレビュー、あるいはシステマティックな調査が可能です。また、制御構造、データ用法、クラス継承など、ソフトウェアのさまざまな側面を幅広くビジュアル化して調査することもできます。プログラムをより早く、より正確に理解することができるようになる結果、生産性も高まり、ソフトウェアの欠陥も減らすことができます。

Imagix 4D の品質解析機能は、ソフトウェアの開発及び維持管理における潜在的問題の特定に役立ちます。変数の使用、タスクの連動、および割込みからの保護に関する一連の解析検証を行うことで、ユーザはリアルタイム組み込みの、マルチタスキング、マルチスレッド化されたシステムで起こりうる衝突を見極めることができます。ソースチェックを利用すれば、明瞭性および可搬性についての設計上及びコーディング上の例外を発見することができます。そしてソフトウェアメトリックスは、開発プロセスを管理し、テスト容易性および保守性に関する組織内の評価基準を満たさないソフトウェアの部分を特定するうえで役立ちます。

また、Imagix 4D は構成の解析、コードのレビュー、システム設計に関するドキュメントの作成に伴う面倒な作業を大幅にカットし、RTF や HTML などの多彩な形式でドキュメントを自動生成することにより、正確で最新の情報を提供します。

以下では、Imagix 4D がビルドされる基礎となる *データモデル* について説明します。このデータを、プログラムを素早く理解するための有益な情報に変換するのが、Imagix 4D ユーザインターフェイスのねらいです。

主要な Imagix 4D ディスプレイは 3 つのパネルに分かれています。Main パネルはディスプレイの上部右側にあります。Main パネルはおそらくユーザインターフェイスの最も重要な部分であり、一連のディスプレイウィンドウが含まれています。これらはそれぞれ、調査中のソフトウェアの特定の側面がより素早く理解できるように、最適化されています。ツールを使用する際は、さらに詳しく調査するために、これらのウィンドウの焦点を絞り直したり、追加のインスタンスを起動したりできます。

2 つの補助的なパネルとして、Main パネルの左に Project パネル、下に Symbol パネルがあります。Project パネルには解析されるソフトウェアの全範囲に関係する一連のタブによって、概観やナビゲーションが示されます。Symbol パネルには特定のシンボルに関する詳細情報が含まれており、そのシンボルについて多くの異なる側面を理解するのに役立つほか、ナビゲーションセンターとしての役割も果たします。

ここでは、使用可能な各種のディスプレイ及びクエリのメカニズムについても説明します。これらの説明は主としてそれらを全般的に理解するために盛り込まれたものであり、レポートについては、レポート内容に関する参考情報が示されています。ディスプレイおよびクエリの実際の使用方法を学習する最適な方法は、ディスプレイとそのコントロールに関するより詳細な情報を提供する Imagix 4D の状況連動型ヘルプ(F1 キー) をご利用になることです。

データモデル

既存のプロジェクトを開く場合、また、プロジェクトにデータソースを追加する場合には、Imagix 4D の基本データベースに自動的にデータがロードされます。全体として、このデータベースには、ソフトウェアに関する膨大な量のデータが一時保存されます。Imagix 4D のユーザインターフェイスは、そのデータをフィルタにかけ、みなさんにとって現在関心のある情報だけを容易かつ迅速に理解できるかたちで提示するメカニズムの集合と考えてよいでしょう。

この基本データベースはオブジェクト指向のエンティティ/関係/属性のデータモデルを使用します。Imagix 4D がみなさんのソフトウェアについて持つデータ――コードのなかのシンボル（エンティティ）、その相互の依存関係（関係）、個々のシンボルに関する固有の情報（属性）――を処理するにも、提示するにも、それがいちばん自然な方法として選択されました。

このデータベースはさまざまなシンボル型を幅広く認識します。たとえば、int のような基本的データ型に始まって、ファイル、ディレクトリといったものまで、あらゆるソフトウェアコンポーネントをシンボルとして扱います。

これらのシンボルはそれぞれひと組の関係の集合で他のシンボルと関連付けられます。たとえば、ひとつの構造体宣言は、それと、どのファイルがその定義及び宣言を含むか、どのような変数をそれがメンバとして含むか、それに基づいてどのようなグローバル及び静的構造体変数が定義されているか、どのような関数がそれをローカル構造体変数の定義に使用するかといった情報を関連付けているでしょう。

Imagix 4D はデータベースの個々のシンボルについて数多くの属性も収集します。これらの属性は変数の有効範囲や関数内の行数など、シンボルのさまざまな特性を記述します。Imagix 4D が収集する属性は、それらが記述するシンボルの型によって決まります。たとえば、ファイルについては、ファイルパーミッション、更新日などの属性を収集します。これに対して、関数の属性にはその範囲やサイクロマチック複雑度などが含まれます。

シンボル型

Imagix 4D は数多くの異なるシンボル型を認識します。このツールの多くの部分では、便宜上、個々のシンボル型をグループにまとめています。ある種のファイル型（メイクファイル、オブジェクトファイル、ターゲット）はメイクファイル解析を通して収集されます。その他のシンボルデータはすべて、ソースコード解析によって収集され、更新されます。

Directories（ディレクトリ）	ディレクトリのみを含みます。
Files（ファイル）	バイナリ、c、c++、実行、ヘッダ、ライブラリ、メイク、オブジェクトなどのファイルを含みます。
Namespaces / Packages（名前空間/パッケージ）	C++では名前空間と呼ばれ、Java ではパッケージと呼ばれます。
Classes（クラス）	テンプレート、クラス、構造体、共用体などを含みます。
Functions（関数）	関数及びバーチャル関数を含みます。
Macros（マクロ）	マクロ定義のみを含みます。
Variables（変数）	変数及び関数ポインタを含みます。
Data Types（データ型）	配列、列挙体、ポインタ、事前定義型、typedef などの型を含みます。

関係型

関係は、ふたつのシンボル間の依存関係を矢印で表します。Imagix 4D のデータベースは次のような関係型を認識します。別途の断りがないかぎり、関係データはソースコード解析によって収集され、更新されます。

Aggregates（集成） クラス 1 => クラス 2
クラス 2 がクラス 1 のメンバの宣言に使われていることを表します。

Base Class of（基底クラス） クラス 1 => クラス 2
クラス 2 がクラス 1 を継承していることを表します。

Calls（呼び出し） 関数 => 関数
関数 => マクロ
関数が前処理されるときにマクロが展開されることを表します。

Contains（包含） ディレクトリ => ディレクトリまたはファイル
ファイル => シンボル
クラス => メンバ
関数 => local_variable

Declares（宣言） ファイル => シンボル
クラス => メンバ

Depends On（依存） ファイル 1 => ファイル 2
ファイル 1 がファイル 2 からビルドされるという規則がメイクファイルに存在することを示します。これは makefile_contents データソースを用いて収集されます。

Has Friend（フレンド） クラス => クラス

Has Type（型） 変数=>data_type
関数=>data_type
関数がパラメータまたはローカル変数の宣言のために data_type を使用することを表します。

Includes（インクルード） ファイル 1 => ファイル 2
ファイル 1 が#include <ファイル 2>のような行を通してファイル 2 をインクルードすることを表します。

Links To（リンク） ディレクトリ 1 => ディレクトリ 2
ディレクトリ 2 への Unix リンクが存在することを表します。
ファイル 1 => ファイル 2
ファイル 2 への Unix リンクが存在することを表します。

Overridden By（オーバーライド） 関数 1 => 関数 2
クラス 2 がクラス 1 を継承するところでクラス 1 メンバがクラス 2 メンバによってオーバーライドされることを表します。

Reads（読み取り） 関数 => 変数
関数 1 => 関数 2
関数 1 が関数 2 を引数として渡すことを表します。

Sets（セット） 関数 => 変数

ソフトウェアメトリックス

Imagix 4D は、データベースの個々のシンボルについて数多くの属性を収集します。どのような属性が収集されるかは、それが記述するシンボルの型によって決まります。 別途の断りがないかぎり、属性データはソースコード解析によって収集され、更新されます。

収集される主要な属性に、ソフトウェアメトリックスがあります。ソフトウェアメトリックスはソースコード内にあるシンボルのある側面の定量的測定結果です。Imagix 4D が収集するメトリックスは、それらが記述するシンボルの型によって決まります。メトリックスを表示および解析するための主要な Imagix 4D のツールは、Reports メニューからアクセスできる Metrics ウィンドウです。

ディレクトリレベルのメトリックス

C Files（C 言語のファイル）	ディレクトリ内の C ファイルの数です。
C++ Files（C++言語のファイル）	ディレクトリ内の C++ファイルの数です。
Comment Ratio（コメント比）	ディレクトリ内のソースコードの行数に対するコメントの行数の比です。
Declarations in Directory（ディレクトリ内の宣言数）	型、変数、関数、マクロ定義など、ディレクトリ内の最高レベル宣言の数です。Java には適用されません。
Directory Size (bytes)(ディレクトリサイズ)	ディレクトリ内にあるファイルの合計サイズ（単位バイト）です。
Functions in Directory（ディレクトリ内の関数）	ディレクトリに定義されている関数の数です。
Halstead Intelligent Content（Halstead インテリジェントコンテント）	ディレクトリの内容量（複雑度）に関する、言語に依存しない測定値です(Halstead I)。
Halstead Mental Effort（Halstead メンタルエフォート）	ディレクトリの生成または理解に必要な基本的弁別数の測定値です(Halstead E)。
Halstead Program Volume (Halstead プログラム量)	ディレクトリの情報的な内容の測定値です(Halstead V)。
Halstead Program Difficulty（Halstead プログラム困難度）	ディレクトリがそのアルゴリズムをコンパクトに実装する度合の測定値です(Halstead D)。これは Halstead Program Level（抽象度）の逆です。
Header Files（ヘッダファイル）	ディレクトリ内の C および C++のヘッダファイルの数です。
Java Class Files（Java クラスファイル）	ディレクトリ内の Java クラスファイルの数です。
Java Files（Java ファイル）	ディレクトリ内の Java ファイルの数です。
Lines of Source Code（ソースコード行数）	ディレクトリ内のステートメントの行数です。
Lines of Comments（コメント行数）	ディレクトリ内のコメントの行数です。
McCabe Average Complexity	ディレクトリ内に定義されている関数のサイクロマチック複

(McCabe 平均複雑度)	雑度の平均値です。
McCabe Max Complexity (McCabe 最大複雑度)	ディレクトリ内に定義されている関数のサイクロマチック複雑度の最大値です。
McCabe Total Complexity (McCabe 合計複雑度)	ディレクトリ内に定義されている関数のサイクロマチック複雑度の合計値です。
Maintainability Index (保守容易性指標)	ディレクトリの保守容易性の測定値です (Welker MI)。
Number of Statements（ステートメント数）	ディレクトリ内のステートメント数です。
Total Files（ファイル合計）	ディレクトリ内の C、C++、および Java のファイルの数です。
Total Lines(総行数)	ディレクトリ内の行数です。
Variables in Directory（ディレクトリ内の変数）	ディレクトリ内に定義されている変数及びパラメータの数です。ローカル変数及びパラメータの数を数えるには、-locals オプションを有効にしてデータを収集しなければなりません。Java には適用されません。

ファイルレベルのメトリックス

Comment Ratio（コメント比）	ファイルのソースコードの行数に対するコメントの行数の比です。
Declarations in File（ファイルの宣言数）	型、変数、関数、マクロ定義など、ファイルの最高レベル宣言の数です。Java には適用されません。
Directly Included-By Files（直接インクルードしているファイル数）	ファイルを直接インクルードしているプロジェクトデータベースのヘッダファイルの数です。Java には適用されません。
Directly Included Files（直接インクルードされているファイル数）	ファイルに直接インクルードされているプロジェクトデータベースのヘッダファイルの数です。Java には適用されません。
File Size（bytes）（ファイルサイズ）	バイト数で表したファイルの大きさです。
Functions in File（ファイル内の関数）	ファイル内に定義されている関数の数です。
Halstead Intelligent Content（Halstead インテリジェントコンテント）	ファイルの内容量（複雑度）の言語に依存しない測定値です(Halstead I)。
Halstead Mental Effort（Halstead メンタルエフォート）	ファイルの生成または理解に必要な基本的弁別数の測定値です(Halstead E)。
Halstead Program Volume (Halstead プログラム量)	ファイルの情報的な内容の測定値です(Halstead V)。

Halstead Program Difficulty（Halstead プログラム困難度）	ファイルがそのアルゴリズムをコンパクトに実装する度合の測定値です(Halstead D)。これは Halstead Program Level（抽象度）の逆です。
Included Lines（インクルードされている行数）	ファイルに間接的にインクルードされているプロジェクトデータベースのヘッダファイルの行数です。Java には適用されません。
Lines of Source Code（ソースコード行数）	ファイルのステートメントの行数です。
Lines of Comments（コメント行数）	ファイルのコメントの行数です。
McCabe Average Complexity (McCabe 平均複雑度)	ファイル内に定義されている関数のサイクロマチック複雑度の平均値です。
McCabe Max Complexity (McCabe 最大複雑度)	ファイル内に定義されている関数のサイクロマチック複雑度の最大値です。
McCabe Total Complexity (McCabe 合計複雑度)	ファイル内に定義されている関数のサイクロマチック複雑度の合計値です。
Maintainability Index (保守容易性指標)	ファイルの保守容易性の測定値です (Welker MI)。
Number of Statements（ステートメント数）	ファイル内のステートメント数です。
Total Lines（総行数）	ファイルの行数です。
Trans Included-By Files(間接的にインクルードしているファイル数)	ファイルを間接的にインクルードしているプロジェクトデータベースのヘッダファイルの数です。Java には適用されません。
Trans Included Files（間接的にインクルードされているファイル数）	ファイルに間接的にインクルードされているプロジェクトデータベースのヘッダファイルの数です。Java には適用されません。
Variables in File（ファイル内変数の数）	ファイル内で定義されている変数及びパラメータの数です。ローカル変数及びパラメータの数を数えるには、-locals オプションを有効にしてデータを収集しなければなりません。Java には適用されません。

名前空間/パッケージレベルのメトリックス

Member Attributes（メンバの属性）	名前空間の属性(メンバ変数)の数です。
Member Classes(メンバクラス数)	名前空間のクラス(メンバクラス)の数です。
Member Methods（メンバメソッド）	名前空間のメソッド(メンバ関数)の数です。

Total Members of Class(クラスの総メンバ数)	名前空間のメンバの総数です。

クラスレベルのメトリックス

CK Class Cohesion(CK クラスの凝集性)	Lack of Cohesion of Methods(LCOM)は、クラスのメンバ関数の凝集性を表す測定値です(Chidamber と Kemerer による LCOM、1994 年版改訂定義)。また、ユーザはクラスの凝集性について別の定義を選択することもできます。選択肢の 1 つは、Li と Henry によるグラフ理論の計算に対する、Hitz と Montazeri による 1996 年の再定義です。もう 1 つは Henderson-Sellers の定義で、特定の属性(メンバ変数)の読み取りまたはセットを行わないメソッド(メンバ関数)の比として、クラスのあらゆる属性について平均して計算されます。
CK Class Coupling（CK クラスカップリング）	クラスのカップリング、あるいは依存関係の尺度です(Chidamber と Kemerer による CBO)。 これは使用される外部クラスの数を表します。
CK Depth of Inheritance (CK 継承の深さ）	クラスから基底クラスへの、階層の深さです(Chidamber と Kemerer による DIT)。
CK Number of Children(CK 子の数)	クラスから直接派生したクラス数です (Chidamber と Kemerer による NOC)。
CK Response for Class(CK クラスのレスポンス数)	クラスのメソッドに呼び出されるメソッドの数で、クラスのレスポンスの測定値です (Chidamber と Kemerer による RFC)。
CK Weighted Methods(CK 重みつきメソッド数)	クラスのメソッドに対するサイクロマチック複雑度の合計値です(Chidamber と Kemerer による WMC)。
Class Coupling（クラスカップリング）	クラスのカップリング、あるいは依存関係のもう 1 つの尺度です。計算は、継承されたクラスの数、ネストされたクラスの数、静的メンバ関数への呼び出しの数、及び、クラスメンバ関数によって使用され、ひとつのクラス型を持つパラメータ及びローカル変数の数に基づきます。正確には、データは-locals オプションを有効にして収集されなければなりません。この値が高くなればなるほど、クラスの理解、使用、修正のために多くの努力が要求されます。
Derived Classes（派生クラス）	クラスから派生クラスへの、階層の深さです。
Ext. Methods and Coupling(外部メソッドとカップリング)	使用されている外部クラス、およびクラスメンバメソッドにより呼び出される外部メソッドの総数です。これはクラス間でのカップリングと呼び出される外部メソッドの合計です。正確には、データは-locals オプションを有効にして収集されなければなりません。
Fan In of Inherited Classes（継	クラスによって直接継承された基底クラス数です。

承クラスの論理入力数)	
Halstead Intelligent Content (Halstead インテリジェントコンテント)	クラスの内容量(複雑度)の言語に依存しない測定値です(Halstead I)。
Halstead Mental Effort (Halstead メンタルエフォート)	クラスの生成または理解に必要な基本的弁別数の測定値です(Halstead E)。
Halstead Program Volume (Halstead プログラム量)	クラスの情報的な内容の測定値です(Halstead V)。
Halstead Program Difficulty (Halstead プログラム困難度)	クラスがそのアルゴリズムをコンパクトに実装する度合の測定値です(Halstead D)。これは Halstead Program Level(抽象度)の逆です。
Local Methods(ローカルメソッド数)	クラスのローカルメソッド(非公開メンバ関数)の数です。
McCabe Average Complexity (McCabe 平均複雑度)	クラスメソッドのサイクロマチック複雑度の平均値です。
McCabe Max Complexity (McCabe 最大複雑度)	クラスメソッドのサイクロマチック複雑度の最大値です。
McCabe Total Complexity (McCabe 合計複雑度)	クラスメソッドのサイクロマチック複雑度の合計値です(CK クラスの応答と同じ)。
Member Attributes(メンバ属性)	クラスの属性(メンバ変数)の数です。
Member Classes(メンバクラス数)	クラスのネストされたクラス(メンバクラス)の数です。
Member Types(メンバ型数)	クラスのメンバ typedef の数です。
Member Methods(メンバのメソッド数)	クラスのメソッド(メンバ関数)の数です。
Methods Called, External (呼び出されるメソッド-外部)	クラスのメソッドにより呼び出される外部メソッドの数です。
Methods Called, Internal (呼び出されるメソッド-内部)	クラスのメソッドにより呼び出される内部メソッドの数です。
Total Members of Class(クラスの総メンバ数)	クラスのメンバの総数です。

関数レベルのメトリックス

Coverage(カバレッジ)	実行されたコードのブロックのパーセンテージです(プロファイルデータ)。
Decision Depth in Function(関数のデシジョンの深さ)	関数内のネストされた判定のレベルまたは制御文のレベル数です。
Fan In of Function Calls(関数	プロジェクトにロードされるソースファイルに含まれ、直接ま

呼び出しの論理入力数)	たは関数ポインタを通して呼び出すか、読み出す関数の数です。
Fan Out of Function Calls(関数呼び出しの論理出力数)	関数に直接または関数ポインタを通して呼び出すか、読み出される関数の数です。
Frequency(実行頻度)	実行中に関数が呼び出される回数です(プロファイルデータ)。
Global Variables Used(使用されるグローバル変数)	関数によってセット/読み込みが行われるグローバル変数の数です。
Halstead Intelligent Content (Halstead インテリジェントコンテント)	関数の内容量(複雑度)の言語に依存しない測定値です(Halstead I)。
Halstead Mental Effort (Halstead メンタルエフォート)	関数の作成または理解に必要な基本的な心理的識別数の測定値です(Halstead E)。
Halstead Program Volume (Halstead プログラム量)	関数の情報的な内容の測定値です(Halstead V)。
Halstead Program Difficulty (Halstead プログラム困難度)	関数がそのアルゴリズムをいかにコンパクトに実装するかの測定値です(Halstead D)。これは Halstead Program Level(抽象度)の逆です。
Knots (ノット)	関数の複雑度および体系化されていない度合の測定値です(Woodward、Hennell、Hedley によるノット)。
Lines in Function(関数行数)	関数の行数です。
Lines of Source Code(ソースコード行数)	関数のステートメントの行数です。
McCabe Cyclomatic Cmplx (McCabe サイクロマチック複雑度)	関数のサイクロマチック複雑度 (McCabe v(G))。これは、関数のデシジョンポイントの数を表します。またこれは、関数内の線形に独立したテストパスの数も表します。ユーザはその他のサイクロマチック複雑度の計算方法として、Myer による 1979 年版 拡張定義 であるデシジョン内の述部をカウントする方法、または修正定義である並列の case を単一のデシジョンとしてカウントする方法を選択することもできます。
McCabe Decision Density (McCabe デシジョン密度)	関数のデシジョン密度(サイクロマチック密度)。関数のステートメント行数に対するサイクロマチック複雑度の比率として計算されます。
McCabe Essential Cmplx (McCabe 本質的な複雑度)	関数の本質的な複雑度(McCabe ev(G))。体系化されていない構成体を含む関数のデシジョンポイントの数を表します。
McCabe Essential Density (McCabe 本質的な密度)	関数の本質的な密度、または体系化されていない度合を表します。本質的な複雑度のサイクロマチック複雑度に対する比率として計算されます。
Static Variables Used(使用され	関数によってセット/読み込みが行われる静的変数の数で

る静的変数)	す。
Transitive Fan In of Funcs(関数の間接的な論理入力数)	プロジェクトにロードされるソースファイルに含まれ、関数ポインタを通じた呼び出しを含む、間接的に呼び出すまたは読み出す関数の数です。
Transitive Fan Out of Funcs(関数の間接的な論理出力数)	関数ポインタによる呼び出しを含め、関数によって間接的に呼び出されるまたは読み出される関数の数です。
Time(実行時間)	関数の実行に費やされる時間です(プロファイルデータ)。
Variables in Function(関数内変数の数)	関数に含まれるパラメータ及びローカル変数の数です。正確には(0 以外)、データは-locals オプションを有効にして収集されなければなりません。

変数レベルのメトリックス

Functions Reading(読み取る関数の数)	変数を読み取るプロジェクトにロードされる、ソースファイル中の関数の数です。
Functions Setting(セットする関数の数)	変数をセットするプロジェクトにロードされる、ソースファイル中の関数の数です。
Functions Using(使用する関数の数)	変数の読み取り及び/またはセットを行うプロジェクトにロードされる、ソースファイル中の関数の数です。

Source Checks(ソースチェック)

Imagix 4D によって収集される属性にはソースチェックが含まれます。これらファイルレベルのソースチェックは、設計上及びコーディング上の例外を指摘できるため、品質保証プロセスに役立ちます。ソースチェックの例外は、ソースチェックレポートにリストされ、File Editor ではアンダーラインを付けて表示されます。

ファイルベースのソースチェック

Built-In Operators(ビルトイン演算子)	1 つの式の中から、ビルトイン演算子の数がユーザ指定の閾値を超えることを示します。
Conversion Issue(型変換の問題)	精度を失う、あるいは算術またはポインタの移植不可能な解釈に左右される可能性がある変換をフラグします。これには、移植可能性の問題につながる可能性のある有符号及び無符号オペランドの混合も含まれます。また、有符号オペランドの右シフトも含まれます。(例: int x; unsigned y; x = y;)
Functions in Expression(式の中の関数)	1 つの式の中から、関数呼び出しの数がユーザ指定の閾値を超えることを示します。
Jump Statement(跳躍文)	コントロールフローの理解を難しくする goto 文、break 文、continue 文をフラグします。
K&R Style Declarator(K&R 型宣言子)	C++では有効ではない古い型の K&R 構文を使用する関数宣言子をフラグします。

Missing Default Case（デフォルト case の欠落）	デフォルト case のない switch 文をフラグします。
Missing/Mismatched Decl（宣言の欠落/不一致）	関数の宣言と関数呼び出しに使用されているパラメータの不一致を示します。これは主に、C で原型が欠落しているか不一致の場合に有効です。
Missing Return Type（戻り値型の欠落）	戻り値型を指定しない関数宣言をフラグします。
Needs Compound Statement（複文が必要）	欠落している可能性のある複文（ブレース）をフラグします。これは、ぶら下がっている else、または while、for から誤って分離された文によって生じる可能性のあるエラーも含まれます。
No Constructor（コンストラクタ無し）	クラスがコンストラクタを持たないことを示します。クラスオブジェクト初期化の際に問題を避けるため、明示的コンストラクタをつねに定義しておくことをおすすめします。
Omitted Lines（無視される行）	現在のプリプロセッサの設定によりアナライザに無視されるソース行をフラグします。
Old Style Allocator（古い型のアロケータ）	malloc、realloc、calloc、または free など、C++プログラムで古い型のメモリアロケータの使用をフラグします。
Potential Static Function（潜在的静的関数）	関数が定義されているファイルの内部からしか呼び出されていないのに静的と定義されていないことを示します。
Problematic Constructor（問題のあるコンストラクタ）	クラスコンストラクタの問題のある定義をフラグします。このなかには、コンストラクタからのバーチャル関数呼び出し、及び初期化順序問題などが含まれます。
Return Ignored（返却無視）	関数呼び出し、または計算が記憶されないあらゆる式の結果を無視する文をフラグします。これは通常、エラー結果が無視されることを示します。
Skipped Lines（スキップ行）	アナライザが解決できない構文問題によりスキップされるソース行をフラグします。ソースがオリジナルの開発環境でコンパイルする場合には、これはインクルードファイルの欠落、またはアナライザの誤った設定を示している可能性があります。
Suspicious Assignment（疑わしい代入）	条件式または引数リストで、代入が現実には比較を意図しているような場合に、疑わしい代入をフラグします。
Unclear Subexpression（不確かなサブエクスプレッション）	意図したように書かれていない可能性のあるサブエクスプレッションをフラグします。これには、符号無し変数及び負の数といった静的に評価することができる比較などがあります。疑わしい関係型演算子の使用、丸括弧の欠如の可能性などをよくとらえます。（例: x && y | z）
Unused Global Variable（未使用のグローバル変数）	プロジェクトにロードされたソースファイルのなかでグローバル変数が使用されていないことを示します。
Unused Local Variable（未使用	ローカル変数が使用されていないことを示します。

のローカル変数)	
Unused Static Variable(未使用の静的変数)	静的変数が使用されていないことを示します。
Unterminated Case(終了しない case)	break 文など、明示的な制御の移転で終了しない switch 文の case をフラグします。
Variable Number of Args(可変個の引数)	可変個の引数を持つ関数をフラグします。

その他の属性

ソフトウェアメトリックスやソースチェックに加えて、Imagix 4D では、コードに関するその他の情報が収集及び記録されます。これらその他の属性はユーザインターフェイスのさまざまな場所で表示され、コードについての理解が深まります。

一般シンボル属性

Kind(種類)	初期化、ライブラリなど、シンボル型の追加記述を指します。
Line Number(行番号)	ソースファイルのなかでシンボルの定義または宣言が始まる行を指します。
Scope(有効範囲)	グローバル、公開など、シンボルの有効範囲を指します。

他のファイルレベルの属性

Modification Date(更新日)	ファイルが最後に更新された日付です。
Owner(所有者)	ファイルの所有者の識別子です。
Path(パス)	ファイルのディレクトリ上の場所です。
Permissions(パーミッション)	ファイルに関する読み取り、書き込み、実行のパーミッションです。

グラフウィンドウ

Imagix 4D のデータベースには、みなさんのソフトウェアから収集される広範なデータが納められます。このデータを、プログラムの理解を促進する有用な情報に変換するのが、Imagix 4D のユーザインターフェイスの主な目的です。そのユーザインターフェイスで最も重要な部分を担うのが一連のディスプレイウィンドウであり、各ウィンドウはそれぞれ、調査しているソフトウェアの個々の側面理解を促進するように最適化されています。

そのようなウィンドウのうち最も強力であり、最もユニークであるのが Graph ウィンドウです。Graph ウィンドウは、個々のシンボル及びそれらの相互の関係を図示し、コードに内在する構造及び依存関係をビジュアル化することにより、ソフトウェアの理解を助けます。これにより、コードに内在する構造や依存関係をビジュアル化することができます。

シンボルはさまざまな形状で表現されます。個々のシンボル型は互いに区別できるように特定の形状と色で表現されます。関係は直線で表現されます。Imagix 4D が捕捉する関係はすべて方向性を持っており、これは矢印で表現されます。直線の色は関係の型を表しています。

最も基礎となる Graph ウィンドウコントロールは *View* であり、ここで現在のプログラム理解タスクのために最適化された全般的な表示方法を選択できます。View の選択により、Graph ウィンドウで可視にするシンボル型および関係型のほか、情報のグラフィカルな表示に使用されるフォーマットを制御します。Symbols 機能を利用すると(メニュー[Help] > [Symbols])、現在ビュー可能なシンボル型及び関係型の形状と色を一覧できるウィンドウが表示されます。Graph ウィンドウ上で特定のシンボルまたは関係にマウスポインタを合わせて Shift キーと Ctrl キーを押しながらマウスの左ボタンを押すと、押している間だけそのシンボルまたは関係に関する Information オーバレイが表示されます。

Graph ウィンドウのクエリ機能はグラフに表示するシンボルの完全な調整を可能にします。グラフに表示するものを指定することができるように、Select、Traverse、Group、Filter のクエリ機能がそろっています。また、グラフは Bookmark 機能を用いて格納し、検索することができます。

Analyze により、グラフィカルなクエリ処理の大部分が自動的に行われます。

ビュー

View の選択により、Graph ウィンドウで可視にするシンボル型および関係型が制御されます。 変数定義での型の使用という低いレベルから、ファイル間の関数呼び出しにおけるファイルレベルの抽象化に至るまで、コードの側面を調査するためのさまざまなビューが提供されています。

これらのビューは Graph ウィンドウの View メニューの下にビューの集合としてまとめられ、グラフの外観および動作に従って大きく 3 つのカテゴリに分けられています。特定のビューを選択することによって、グラフのコンテンツの管理に使用可能な機能だけでなく、グラフの外観も制御されます。

ビューの 1 番目の、そして最大のカテゴリは Structure ビューです。このビューは最も汎用的なビューであり、これによってソフトウェアの構造を調べ、ファイル、名前空間、クラス、関数、変数、および/またはデータ型の相互作用を理解することができます。

ときによっては、コードのなかでの関数呼び出し及び変数使用のシーケンス及び条件を理解することが重要な場合もあるでしょう。Control Flow ビューは従来の構造図と機能特定のフローチャートデータを結びつけ、詳細なコントロールフロー解析を可能にします。

UML ビューでは、統一モデリング言語（Unified Modeling Language）の表記を使用して既存のソフトウェアが表示されます。UML Class Diagram ビューおよび UML File Diagram ビューによって、クラスとファイルのレベルにおけるインターフェイス、コラボレーション、及び関係を調査することができます。UML

Task Cllaboration Diagram によって、タスクが共有変数を通してどのようにやり取りを行うかを解析することが可能です。

Structure ビュー

Structure ビューは、高レベルのアーキテクチャからビルド、手続き、データ、及びクラス依存関係の詳細まで、あらゆるレベルでのソフトウェアのシステマティックな調査を可能にします。

Imagix 4D のデータベースのデータは、どのようなソフトウェアプロジェクトでも、プロジェクト全体ではきわめて膨大な量になる可能性があります。グラフィックな解析では、Graph ウィンドウで現在関心のある特定の情報に焦点が当てられます。なかでも第 1 にあげられるのは View メカニズムで、これにより Graph ウィンドウに表示するシンボル及び関係の型を調整することができます。Structure ビューでは、どの関係型をビュー可能にするかを指定することができ、非常に多くのなかから細かくシンボル型を選択することができます。

関数の複雑度、またはテストケースがコードをどの程度完全に実行したかなど、ソフトウェアの定量的特性を調べることもできます。ソフトウェアメトリックスの情報は、各シンボルに関連するメトリックス値を色分けした状態で Graph ウィンドウに表示することができます。また、特定の Flow Check レポートによって識別された関数を示すために色分けすることも可能です。

グラフの外観及びレイアウトは Graph ウィンドウの下のチェックボックスで調整します。大きさと方向の設定によりグラフを調整して、現在のコンテンツをより見やすくすることができます。レイアウト及び起点の設定はグラフ上のシンボルの相対的な位置関係を変化させます。グラフに表現された内在する構造及び依存関係をいかに素早く、いかに完全に理解することができるかは、これらの設定によって違ってくる可能性があります。

ビューの範囲

Imagix 4D のデータベースはソフトウェアに含まれるさまざまなシンボル、及びそれらの相互の関係に関する情報を含みます。シンボルはソフトウェアのオブジェクトまたは要素です。たとえば、ヘッダファイル、クラス、マクロなどが含まれます。ある関数から別の関数への呼び出し、あるいは宣言にデータ型を使用している変数などの関連づけは、2 つのシンボル間の関連が持つ方向性に基づいています。

一般に、Imagix 4D によって収集されるシンボル及び関係の情報は、ファイルとファイルの間のインクルード依存関係に始まって、クラスの他のクラスからの派生、セットされ、読み取られる変数にいたるまで、ソフトウェアのさまざまな側面をカバーします。

Structure ビューでは、これらすべての情報は潜在的にビュー可能です。ソフトウェアの特定の側面解析を助けるために、ビューメカニズムによって Graph ウィンドウでビュー可能なシンボル及び関係型を調整することができます。その方法は 2 つあります。

より簡単な方法は、View メニューにリストされた既存のビューの 1 つを選択する方法です。Imagix 4D には、事前定義のシンボル型と関係型の組み合わせが多数用意されています。これらのビューを使用して、一般的に興味のあるソフトウェアの構造的な側面のうち多くを調べることができます。例えば、Function Call を Variables ビューで表示すると、ソフトウェアにおける関数の呼び出し階層を調べることができ、それと同時に、それらの関数の中でグローバル変数および静的変数の使用方法が確認できます。

Other メニュー項目からは Set View ダイアログが開き、より多くの選択肢とより詳細なコントロールが利用可能です。ここでビュー可能なシンボルと関係の組み合わせを追加で指定できます。特定の関係型は特定のシンボル型だけに適用されるため、一方を変更するともう一方に影響します。

たとえば、変数（variables）を無効にすると、セットされるまたは読み取られるビュー可能なシンボルが存在しないため、関数から変数へのセット及び読み取りの関係もグラフから消えます。Set View では、表示

される関係を明示的に制限したい場合もあるかもしれません。たとえば、ある関数によってセットされるすべての変数を見たいが、関数によって読み取られる変数には関心がない場合もあるでしょう。ビュー可能な関係の調整は、カレントグラフのレイアウトばかりでなく、Add 機能を通して関係クエリを行ったときに追加されるシンボルにも影響します。

頻繁に使用するシンボル型と関係型の組み合わせがある場合には、Set View の Save の機能を使用して、ビューを既存のビュー定義のリストに追加することもできます。

グラフのレイアウト

Graph ウィンドウでシンボルの配置を変えることにより、グラフを最適化して、調べているコードに内在する構造及び依存関係を理解しやすくすることができます。この調整は Graph ウィンドウの下部にあるチェックボックスで行います。

シンボル間の関係にはすべて方向性があります。たとえば、関数が変数に値を代入するときに生じるセットの関係には、関数から変数への方向性があります。可視のシンボルで、他の可視のシンボルから向かってくる関係を持たないものはすべてルートと見なされます。可視のシンボルで、他の可視のシンボルへ向かう関係を持たないものはすべてリーフと見なされます。

レイアウトの選択肢にはノーマルレイアウトとコンパクトレイアウト（Compact）があります。双方の違いを説明するため、ここでは、グラフは横の流れで表示されているものと仮定します。ノーマルレイアウトでは、ルートシンボルはすべて左端の列に置かれます。次の列には、まだ配置されていないシンボルが表示されれば、それらのシンボルに対してルートとなるシンボルが置かれます。これは残りのシンボルすべてが配置されるまで繰り返されます。このようにしてシンボルが配置される結果、関係の矢印はすべて左から右へ向くことになります。縦方向の矢印は、再帰的な関係で結ばれたシンボル間の再帰を示します。

コンパクトレイアウトでも、ルートはやはり、すべて左端の列に置かれます。次の列には、ルートシンボルと直接関係のあるすべてのシンボルが含まれます。3 列目には、2 列目のシンボルと直接関係があり、最初のルートの列とは直接関係のないすべてのシンボルが含まれます。以下、同様です。このレイアウトでは、シンボルがよりコンパクトに配置されますが、関係の矢は両方向に向くことがあります。

第 2 のレイアウトの選択肢は、ルートに当たるシンボルをすべて最初の列に置いてグラフを表示するか、リーフに当たるシンボルをすべて最後の列に置いてグラフを表示するかです。これらのレイアウトの違いは、コンパクトレイアウトが選択されているときに最大になります。

通常はルートを基準として最初の列に置くレイアウト(FromRoots)を選択するでしょう。いまかりに、ある関数呼び出し階層をビューしていて、関数 X のサブツリーを切り離し、そのサブツリーを調べようとしているとします。FromRoots のビューは、関数 X によって直接呼び出される関数にどのようなものがあり、それらの関数によって使用される追加の関数にどのようなものがあるかを明確に示します。

それでも、FromRoots のチェックを外したほうがよい場合もいろいろとあるでしょう。同じようにある呼び出し階層を見ていても、ひとつの変数がどのように使用されているかを理解しようとしている場合を考えてみましょう。FromRoots のチェックを外したビューでは、その変数に対して直接セットまたは読み取りを行う関数にどのようなものがあり、それらの関数がどのような関数によって呼び出されるかがよくわかります。

グラフの外観

また、グラフの外観は、データが見やすくなるように調整することができます。Graph ウィンドウの下部にある一連のチェックボックスの中に 3D および Vertical のペアがあり、実際のレイアウトを変えることなくグラフの外観を制御することができます。

シンボル及びそれらの関係のグラフは、3D のチェックボックスにチェックが入っているかどうかに応じて、3 次元または 2 次元で表示されます。Vertical のチェックボックスは、グラフを上から下への(縦<vertical>の)流れで表示するか、左から右への(横の)流れで表示するかの選択に使用します。

これらのグラフはみな同じものであり、外観だけが異なります。自分でいいと思う外観に設定してください。また、調べているデータが変化するうちに設定を変えたくなることもあるでしょう。

Graph ウィンドウでシンボルや関係を調べるときは、グラフのなかのとくに関心のある部分がよく見えるように、パン、ズームをすることもできます。3D モードでは、グラフを回転させることもできます。

メトリックスとレポート結果

Display 設定では、個々の関数のサイクロマチック複雑度など、ソフトウェアのさまざまな定量的特性を表示する色の調整が可能です。あるソフトウェアメトリックスについて色を有効にすれば、シンボルのメトリックスのアッパースレッショルド及びロウアースレッショルドに対する相対的な値が Graph ウィンドウの個々のシンボルについて表示されます。各メトリックスのスレッショルドは、Options ダイアログ(メニュー [Tools] > [Options] > [Metrics] > [Threshold])で設定します。このスレッショルドは、Metrics ウィンドウ及びその他のメトリックス値が表示されるディスプレイに適用されます。

また、Display 設定を使用して、ある Flow Check でレポートされた関数を特定することもできます。特に、様々な Task Flow Check によってレポートされた問題を調べるために Graph ウィンドウを使用してソフトウェアを解析する際、タスクで使用されていない関数を特定できると非常に役立つことがあるでしょう。

Control Flow ビュー

通常は、ソフトウェアの呼び出し階層に関心があるのであれば、Function Calls を示す Structure ビューの標準的な集合により必要な情報はすべてわかります。ただし、関数呼び出しのシーケンス及び条件、コード内での変数の使用方法を理解する必要があることもあるでしょう。そのような場合により詳細な制御フローの解析ができるように、Imagix 4D は Control Flow ビューを用意しています。

Control Flow ビューでは関数及び変数のみ表示可能であり、複雑なグラフィックスの代わりに関数及び変数のシンプルなアイコン(関数は立方体/四角形、変数はピラミッド形/三角形)が表示されます。その結果、個々の可視の非ライブラリ関数のなかで、他の可視関数がどこで呼び出され、他の可視変数がどこで読み取られ、セットされるかが表示されます。可視である関数および変数の集合が変化するのに伴い、個々の可視の非ライブラリ関数に対して表示される内部コンテンツは自動的に変化します。ビューする選択をした関数及び変数の間での制御のフローを素早く、正確に調査することができます。

UML Diagram ビュー

統一モデリング言語(Unified Modeling Language)は、オブジェクト指向システムのフォワードエンジニアリングの手法として圧倒的に受け入れられてきました。この手法では、クラス図がシステムの静的設計の一次的な表現です。UML Class Diagram ビューは、この表現を利用して、既存のソフトウェアのインターフェイス、コラボレーション、及び関係の理解を助けます。

結果となるグラフでは、各クラスの表現にそのクラスのメンバに関する情報が含まれます。 クラスとクラスの関係は、クラスレベル(Inheritance<継承>、Generalization<汎化>、Aggregation<集約>)でもメンバレベル(UML の分類でいう association<関連>)でも表示されます。

UML Class Diagram ビューは、C++ コードと Java コードのビューにのみ使用可能ですが、C のソフトウェアの解析にも使用可能な UML File Diagram という類似のビューもあります。 UML File Diagram モードでは同じ表現を使用して、システムの静的設計をファイルレベルで表示します。

UML Task Collaboration Diagram ビューはさらに焦点を絞り込んでマルチタスキング、マルチスレッド化されたシステムに対応し、特にタスクが共有変数を通してどのようにやり取りを行うかを示します。このビューは、Hassan Gomaa 氏が提唱する COMET 方法論をもとにしています。Task Flow Check レポートによって使用される同じタスク定義が、ここに適用されます(本ユーザガイドの「タスクフローチェック」の項参照)。

これらのビューでは、まず単純なアズビルトの UML 図を表示してからメンバと関係の情報表示を制御して、複雑なシステムを正確かつ包括的に理解することができます。

Display Format メカニズムにより、表示する関係の型を調整することができます。Class Diagram と File Diagram において、Imagix 4D はコンテナ（クラスまたはファイル）レベルでも、構成メンバのレベル(関連)でもシンボル対シンボルの関係を把握していて、表示することができます。Task Collaboration Diagram における関係の制御では、用法、クリティカル領域の保護、および関連するタスク、関数、および変数によるフィルタリングが可能です。表示する関係型を調整することにより、データ過多に陥らず、必要な情報を見ることができます。

また、Display Format では、クラス、ファイル、タスク、共有変数の各コンテナに内部的に現れるメンバの型を制御することも可能です。Class Diagram と File Diagram では、メンバの設定とメンバレベルの関係(関連)の設定が相互に影響し合って、実際に図の中に表示されるメンバと関係が決定されます。メンバレベルの関係は、表示されているメンバについてのみ表示できます。また、すべてのメンバではなく関連するメンバを表示するようにフォーマットが設定されている場合、メンバレベルの関係の設定では表示対象のメンバが制限されます。

クエリ

Graph ウィンドウのクエリ機能はカレントクエスチョンに関する情報を明瞭かつ簡潔に表示するグラフの生成を可能にすることにより、Imagix 4D に第 4 の D(ディメンション)を提供します。クエスチョンが時間とともに変化すれば、ただちにグラフを変更することができます。

クエリのアクティビティは大きくふたつに分けられます。関心のあるシンボルの識別と Graph ウィンドウに表示される関連情報の絞り込みです。 場合によっては、単にグラフ内の特定のシンボルまたはシンボルの集合を見つけるために識別操作を利用することもあるでしょう。また、現在可視のシンボルのどれが特定の基準を満たすかを調べるため、単純なクエリを実行することもあるでしょう。しかし、多くの場合、識別は何段階ものクエリの最初のステップとして行い、その識別結果を利用してグラフを修正することになります。

識別操作には、Select 及び Traverse があります。Select 機能はシンボルそのものの特性をもとにシンボルを識別するもので、Traverse は現在選択されているシンボルに対する関係をもとに新しいシンボルを識別するものです。

絞り込みは、Filter 及び Group 機能で行います。Filter 機能は、データベースに存在するシンボルの一部を隠すことにより、ビューの単純化を可能にします。逆に、グラフに関心のある情報がすべて含まれていない場合には、隠していた情報をまたディスプレイに追加することもできます。

Control Flow ビューおよび UML Diagram ビューでは使用できない Group 機能では、シンボルの集合をひとつのグループシンボルにまとめることにより、情報を圧縮することができます。 そのグループ以外のまとめられたシンボルの関係を見ることもできます。

Filter 及び Group 機能はプロジェクトのカレントデータベースに存在する情報から単純化されたグラフを生成することができます。プロジェクトそのものに含めるデータソースとそこから除外するデータソースを管理することにより、より核心的で普遍性のある結果を得ることができます。

Imagix 4D のクエリ機能はきわめて柔軟で広範囲に及ぶため、まだこのツールに慣れないうちはとっつきにくく感じるかもしれません。数回のマウスのクリックで一般的なクエリを素早く実行するには、Analyze 機能を使用できます。これはシンボル固有の機能で、マウスの右クリックポップアップメニューから利用可能です。

Select 機能

Select 機能は一定の基準を満たすシンボルを識別し、それによって、Graph ウィンドウでハイライト表示されるものを制御します。

Select 機能は、全ての選択をクリア(None)、または Graph ウィンドウの全てのものを選択(All)する単純なものから、極めて高度な検索機能まで、さまざまです。 Find 機能により、特定の属性を持つシンボルを識別することができます。

一部の機能はシンボルを、それが現在可視のグラフのどこにあるかをもとに識別します。Roots はグラフの最上位階層を識別し、Leaves は最下位階層にあるシンボルを突きとめます。Roots/Leaves は、ルートにして同時にリーフでもあるシンボル、つまり浮動性のシンボルを見つけます。

いつでも、Save を選択すれば、現在選択されているシンボルのリストを格納することができます。Restore 機能を用いてこのリストをリストアすると、そのリスト上にあり、まだ可視のすべてのシンボルが再び選択されたシンボルになります。

選択されているシンボルは現在の Graph ウィンドウ上で、黄色でハイライト表示されます。ある時点で現在の Graph ウィンドウとなるのは 1 つだけです。他の Graph ウィンドウ内の選択されたシンボルは白で表示されます。

Find

Find 機能では、単に Find ダイアログで指定された基準に合致するシンボルが現在の Graph ウィンドウ上でハイライト表示されるだけです。 Graph ウィンドウのコンテンツは変化しません。

Find 機能はシンボルの属性に基づいたシンボルの検索を可能にします。Imagix 4D はシンボル及び関係の情報を収集するとき、個々のシンボルについて数多くの属性を捕捉します。これらの属性には、名前、型などのシンボルの基本的記述から、シンボルのファイル位置、最後にファイルが変更されたのがいつかなど、あまり重要ではない特性まで、さまざまなものが含まれます。

Find 機能では、シンボルをその属性に基づいて検索することができます。Find は、指定した基準に合致する属性を持つ、現在可視であるすべてのシンボルを突きとめます。多数の属性を検索することが可能であり、名前など一部の属性については、glob スタイルの展開(*使用)を使用することができます。

また、検索したシンボル型のコンテナに当たるシンボル、あるいは検索したシンボル型をコンテナとするシンボルも突きとめることができます。たとえば、File Calls などのファイルレベルの Structure ビューで foo という関数を検索するとします。グラフ上では、関数 foo の定義を含むファイルが、すでに可視であった場合に限り、ハイライト表示されます。

Traverse 機能

シンボル間の関係にはすべて方向性があります。たとえば、関数が変数に値を代入するときに生じるセットの関係には、関数から変数への方向性があります。Graph ウィンドウでは、この方向性が矢印で示されます。

これらの関係を目で追っていると、時間がかかりますし、込み入ったグラフになると、目で追うこと自体がきわめて難しくなります。Traverse の機能は、この目で追う作業を自動化します。

Step 及び Full の機能はこうした関係を選択されているシンボルのなかへ(Up)、または外へ(Down)とたどります。Step は隣のシンボルまでたどり、Full はルートまたはリーフに到達するまで再帰的にステップアップまたはステップダウンします。Unique の機能は選択された単一のシンボルの中または外へと関係をたどります。またこの機能は隣のシンボルまで再帰的に関係をたどりますが、隣のシンボルのうち選択されていないシンボルからも到達できるものはたどりません。

Intersection 及び Difference の機能は複数のシンボルが選択されている場合に使用します。Intersection は選択されているすべてのシンボルに共通した祖先(Up)または子孫(Down)をたどります。Difference は選択されているシンボルのすべてではなく、一部について存在する祖先または子孫を選択します。これらを使用する場合の一例としては、ふたつのサブシステムのエントリ関数を選択し、Intersection Down を適用してそれらのサブシステムの共用変数を調べる場合などが考えられます。

これらの Traverse の機能は、あらかじめ Graph ウィンドウに表示されているシンボルの関係をたどります。まだグラフに表示されていないシンボルの階層を調べるには、Add 関数を使用します。

Group 機能

ソフトウェアを調べていると、Graph ウィンドウは複雑すぎて容易に理解できなくなることがよくあります。そのような場合には、通常は Filter 機能を用いて特定のシンボル及びそれらの関係をウィンドウから完全に削除し、グラフを単純化します。しかし、ときには、数多くのシンボルに関する情報をすっかり削除せずに、それらをまとめて圧縮するほうが望ましい場合もあるでしょう。Group 機能を用いれば、それができます。 Graph ウィンドウで Control Flow または UML Diagram ビューが表示されているときは、Group 機能は利用できません。

シンボルの集合をグループにまとめるときには、その集合の外部表現となる新しいシンボルを生成します。現在の Graph ウィンドウ、および Project Panel 内の関連する Graph Symbols タブには、グループ化されたシンボルが個別に表示されなくなります。グループ内のシンボルの関係もそうです。その代わり、グループ全体を表現するひとつのシンボルが表示されます。また、グループにまとめられたシンボルのいずれかがグループ外の可視のシンボルと持っている関係も全て見られます。つまり、グループの外部インターフェイスは調べられるわけです。

Group Definitions 機能により、グループの作成と削除が可能です。個々のグループメンバを、グループに追加するかグループから削除することができます。これらのグループはプロジェクト全体に適用されるので、主要メニューバーの Project メニューから Group Definitions ダイアログにアクセスできます。しかし、Graph ウィンドウのローカルのメニューバーにある Group メニューから Group Definitions ダイアログにアクセスするのが普通です。

グループの自動定義によってファイルとクラスをシミュレートする機能も提供されています。File Index および Class Index の右クリックメニューに、"Create group for members"という項目が含まれています。これを選択すると、あるファイルまたはクラスのすべての関数メンバと変数メンバを含むグループが作成されます。Function Call ビューでこれらのグループを使用すると、ファイルとクラスに対するパッケージ化およびカプセル化の方法を理解し、評価することができます。

グループはプロジェクト全体に対して定義されるので、グループをアクティブにするかどうかを *Activate Groups* 機能によって、特定の Graph ウィンドウごとに制御します。通常はこの機能を、特定の Graph ウィンドウのローカルメニューバーにある Group メニューから利用できる、Activate Groups ダイアログを通して制御します。以前に自動定義されたファイルとクラスのメンバグループをアクティブ化する場合も、File Index および Class Index の右クリックメニューを使用することができます。

Select、Traverse、Filter の機能はすべてグループにも適用されます。これらの機能はグループ全体に対して適用されます。たとえば、Find を実行し、あるグループ内のひとつのシンボルが識別された場合には、そのグループ全体が選択され、ハイライト表示されます。

Group 機能を用いるケースはいろいろと考えられます。高いレベルでアーキテクチャを調べるため、ソフトウェアをサブシステムまたは開発者に従って区分する長期的グループを生成することもあるでしょう。グループ化を what if 解析のひとつの手段として利用し、プログラム要素をあるファイルから別のファイルへ移したときの影響をシミュレートするグループを生成することもあるでしょう。また、既存の C ソフトウェ

アをオブジェクト化しようとしているときには、どのシンボルをどのクラスでまとめるのがよいかを判断するために、グループをカプセル化のモデルとして利用することもできるでしょう。

Filter 機能

ソフトウェアの構造をビジュアル化していると、現在隠れているシンボルを追加したいと思うこと、あるいは現在可視だが、関係のないシンボルを消去したいと思うことがあるでしょう。Filter 機能は Graph ウィンドウに表示するシンボルを制御します。

Filter 機能の一部分を使用して、非表示のシンボルおよびそれらの関係を Graph ウィンドウに追加して戻すことができます。その 1 つの方法は、他のディスプレイウィンドウにおいて手動でシンボルを選択し、Add Selected 機能を使用してそれらをグラフに追加する方法です。Add は、自動化された、ダイアログを利用する方法であり、データベースのシンボルの属性及び関係に関する情報を利用して追加するものを指定します。Restore All は、カレントビューにおいてビュー可能なシンボルで隠れているものをすべて追加します。この方法では、よけいな情報まで追加される可能性があるので、注意が必要です。

Isolate 及び Hide はグラフからシンボルを消去する機能です。Hide All はグラフを完全にクリアし、Hide Library はグラフからライブラリ機能を消去しますが、Isolate Selected 及び Hide Selected は現在選択されているシンボルだけを消去します。 消去したいシンボルは、最初に Select または Traverse の機能で選択し、ハイライト表示させておきます。その上で、そのシンボルを消去したいか、隔離しておきたいかによって、Hide Selected または Isolate Selected の機能を使用します。

シンボルを多く消去または追加しすぎた場合には、Undo 機能(LOCAL メニュー[Edit] > [Undo])で元に戻すか、さらに他の Filter 機能を利用してグラフを修正することができます。

Add

Graph ウィンドウでソフトウェアの構造を調べていると、現在表示されているシンボルに特定のシンボルを追加してビューしたくなることもあるでしょう。 Add 機能は、特定のシンボルをその属性及び関係に関する情報を利用して、グラフに追加することを可能にします。

また、シンボルを現在可視のシンボルに対する階層的関係を利用してグラフに追加することもできます。その場合には、まずカレントグラフでシンボルを選択します。その上で、Add ダイアログで step/full up/down のいずれかを利用して、追加したいシンボルの現在選択されているシンボルに対する関係を記述します。

Add 機能は、シンボルをその属性に基づいて追加することも可能にします。Imagix 4D は、ソースコードからシンボル及び関係の情報を収集するとき、個々のシンボルに関するさまざまな属性も捕捉します。これらの属性には、名前、型などのシンボルの基本的記述から、シンボルのファイル位置、最後にファイルが変更されたのがいつかなど、あまり重要ではない特性まで、さまざまなものが含まれます。

Add 機能では、シンボルをこれらの属性に基づいてフィルタにかけることができます。Add は、指定した基準に合致する属性を持つすべてのシンボルを突きとめます。名前など、一部の属性については、glob スタイルの展開(*使用)を使用することができます。

また、検索したシンボル型のコンテナに当たるシンボル、あるいは検索したシンボル型をコンテナとするシンボルも追加することができます。たとえば、Graph ウィンドウを関数だけが表示されるように設定しているときに bar というクラスを検索したとしましょう。 検索の結果、グラフには、クラス bar のクライアントメンバに当たるすべての関数が追加されます。逆に、ファイルレベルのビューで foo という関数を検索した場合には、関数 foo の定義を含むファイルがグラフに追加されます。

解析

Graph ウィンドウで使用可能なクエリ機能は、きわめて柔軟で広範囲に及ぶため、まだ Imagix 4D に慣れないうちはとっつきにくく感じるかもしれません。数回のマウスのクリックで一般的なクエリを素早く実行するには、Analyze 機能を使用するとよいでしょう。

その他の Main パネルのツール

Imagix 4D のデータベースには、みなさんのソフトウェアから収集される広範なデータが納められます。このデータを、プログラムの理解を促進する有用な情報に変換するのが、Imagix 4D のユーザインターフェイスの主な目的です。そのユーザインターフェイスで最も重要な部分を担うのが一連のディスプレイウィンドウであり、各ウィンドウはそれぞれ、調査しているソフトウェアの個々の側面理解を促進するように最適化されています。

Graph ウィンドウの内容を補足する Main パネル上の一連のウィンドウには、ソフトウェアのその他の側面が表示されます。Flow Chart ウィンドウは関数内部の制御フローを表示します。Calculation Tree には、変数の値に影響する、ソースコードの割り当てに関する情報が表示されます。File Editor ウィンドウはソースコードをそのまま表示しますが、ブラウジング及び理解を促進します。

Imagix 4D の、最大の特長の 1 つは、これらのディスプレイウィンドウによって個別に貴重な情報が得られるのに加え、これらのウィンドウがすべて相互に連動しているところです。これらのディスプレイを組み合わせて使用することで、ソフトウェアをすばやく解析して理解を深めることができます。

Flow Chart ウィンドウ

Flow Chart ウィンドウはソフトウェアの関数内で出現する制御フロー、すなわちプログラムロジックを図示します。数百行のソースコードから成るような複雑な関数の場合には、ルーチンの内部ロジックのより迅速な把握に役立つことがあります。

Help メニューで Symbols を選択すると、図で使用されているシンボルを説明するウィンドウが表示されます。switch 文のような分岐点は菱形で表現され、コードのインラインブロックは正方形で表示されます。実際の制御フローはシンボルをつなぐ直線で示されます。

Flow Chart ウィンドウは File Editor に密接にリンクされています。Flow Chart 上の特定のシンボルにマウスポインタを合わせてクリックすると、対応する File Editor のカーソルがソースコード上のそのシンボルの場所へ移動します。同様に、マウスポインタがソースコード上にある状態でクリックすると、Flow Chart ウィンドウの対応するシンボルが赤でハイライト表示されます。

Flow Chart ウィンドウで Vertical のレイアウトを選択すると、図そのものにソースコードの注釈を付けることができます。これは、Flow Chart をプリントする場合にはとくに便利です。

Flow Chart ウィンドウは個々の関数内の完全な制御フローを表示します。複数の関数にまたがって制御フローを調べたい場合には、Graph ウィンドウの Control Flow ビューを利用します。Control Flow ビューは抽象レベルが 1 段階上がり、Flow Chart のディテールの一部を失いますが、ソフトウェアのより広い部分に関する情報を提供します。

Calculation Tree

Calculation Tree には、変数の現在値に影響する、計算に関する情報が表示されます。特定の変数の値に影響するすべての変数の代入を調べることができます。File Editor の右クリックポップアップメニューから起動する場合は、Calculation Tree には選択したコード行の変数の値に影響する代入だけが表示されます。その他の場合、Calculation Tree では、出現する変数に影響するすべての代入が解析されます。

Calculation Tree ウィンドウは、計算全体のさまざまな側面を示す一連のビューで構成されています。Assignment Flow ビューには、計算に含まれるすべての代入の階層がグラフィカルに表示されます。グラフ内のある代入をクリックすると、その特定の代入に関する詳細がグラフの下の表に表示されます。

これらの代入は、すべて Assignment List ビューにも表示されます。この場合、各代入のすべてのソース行、または各代入の説明がリストに表示されます。リストの順序とインデントは、代入のフローを示します。

Variable Dependencies ビューには、計算全体のうちより高いレベルのグラフが表示されます。計算に関わるグローバル、ローカル変数、パラメータが表示され、これらの変数間の依存関係を調べることができます。Assignment Flow ビューでは、特定の変数または依存関係の詳細がグラフの下に表示されます。

セットされている変数は Variable Table ビューにも表示され、個々の代入は割り当てられている変数によってまとめられます。

表示される情報と他のディスプレイウィンドウとの相互関係のため、Calculation Tree はユーザガイドのこの項で説明されている他のディスプレイウィンドウと合わせて、[Tools]メニューに含まれています。ただし、Calculation Tree に最初に情報が表示される前に、大量のグローバルデータフローの解析が必要になります。プロジェクトの規模によっては、この解析で大幅な遅延が発生します。さらにこの解析ではローカル変数に関する知識が必要になるため、Calculation Tree ではソースコードを-locals オプションで解析する必要があります。このユーザガイドの「アナライザの構文とオプション」の項を参照してください。

File Editor ウィンドウ

File Editor は、ソフトウェアの効率的な読み取り、誘導、編集を助けるさまざまな工夫を凝らして、現実のソースコードを提示します。

プログラムへの理解は、色の使用によって促進されます。コメントは現実のコードそのものとは区別されます。シンボルは、その定義ばかりでなく、使用も、このツールの他のディスプレイと共通の色でコーディングされています。

ハイパーテキストのようなソースコード誘導がサポートされます。色の付いたシンボル上でダブルクリックすると、新しい File Editor ではそのシンボルが定義されているファイル及び行番号がオープンされます。この誘導は、ほかの Imagix 4D ディスプレイのいずれからも可能です。Graph ウィンドウ、Flow Chart、Metrics レポートおよびその他のレポートのウィンドウからダブルクリックするだけで、表示されているソースコードを見ることができます。

もうひとつ、Next Reference、Prev Reference のアイコンで誘導を行う方法もあります。アイコンバーにあるこれらのボタンを利用すると、ひとつのシンボルの型としての定義、宣言、呼び出し、読み取り、書き込み、あるいは使用が行われている場所をすべて順番に見ていくことができます。

File Editor では、Imagix 4D のレポートのいくつかによって特定されたソースコード行をフラグで示すことができます。ファイルベースのソースチェックに関する警告には、アンダーラインが付きます。本ユーザガイドの「データモデル」の項に列挙されたソースチェックは設計上及びコーディング上の例外を表現します。表示されるソースチェックの型を制御することはできません。アンダーラインの引かれたテキストの上にマウスポインタを合わせて、Shift キーと Ctrl キーを押しながらマウスの左ボタンを押すと、押している間だけ Information オーバレイが現れ、ソースチェックがフラグされているものに関する情報が表示されます。

特定の Flow Check によってレポートされた関数のソースコードについても、ソースコードの左のカラムの色付けによりフラグすることができます。特に、様々な Task Flow Check によってレポートされた問題を調べるために File Editor を使用してソースコードをレビューする際、タスクで使用されていない関数のソースコードを特定できれば非常に役立つことがあるでしょう。

File Editor はコードのビューばかりでなく、コードの修正にも利用することができます。Imagix 4D は変更を行番号で追跡します。ただし、変更を行うと Imagix 4D のデータベースとソースコードとの同期が外れることがあります。データベースは、Update Project Data もしくは Regenerate Project Data の機能を用いることによって、あらためてソースコードと同期させることができます。

ソースコードとの同期

Imagix 4D は、データベースの個々のシンボルについて、そのシンボルがソースコードのなかで置かれている場所に関する情報を格納します。このなかには、そのシンボルが定義または使用されている場所のパス、ファイル、行番号も含まれます。

この情報は、データをインポートするときにソースコードから捕捉されます。このため、それはソースコードが最後に解析されたときのシンボルの場所を反映します。ファイルが誤った行番号で開いているとき、また Imagix 4D エディタがシンボルをカラーコーディングしていないときには、データベース更新後にファイルを修正したことを示しています。データベースは、Update Project Data もしくは Regenerate Project Data の機能を用いることによって、あらためてソースコードと同期させることができます。

外部エディタの使用

デフォルトでは、Imagix 4D はソースコードの表示に File Editor を使用します。ただし、いろいろと込み入った編集を行う場合には、File Editor より他のエディタを使用したくなることもあるでしょう。その場合には、Options ダイアログ（メニュー[Tool] > [Options] > [File Editor] > [Alternative Editors]）でエディタを指定することができます。Unix 環境下では、EDITOR の環境変数でエディタを指定していれば、Environment Editor の選択でそれを選択することもできます。

Imagix 4D は Other フィールドに入力されたコマンドを使用し、command *filename* を起動します。ただし、command は Other フィールドに入力されたコマンド、*filename* はブラウズしようとしているシンボルを含むファイルを表します。エディタの起動プロセスはファイル../imagix/user/user_ed.tcl で修正して、シンボルの行番号を含めることができます。

解析の自動化

Graph ウィンドウは、制御構造、データ用法、クラス継承など、ソフトウェアのさまざまな側面を幅広く調査するための包括的な機能を備えています。グラフを拡大したり縮小したりすることで、関数間の依存関係を調べ、グローバル変数がそれを定義するファイルの外部で使用されている箇所を突き止め、データ階層を探索し、ファイルのインクルード関係を追跡することが可能です。ソフトウェアのさまざまな側面をグラフィカルに検索および表示する Imagix 4D の機能は、実質的に無限です。

Analyze 機能はこの機能を自動的に行うもので、簡単で迅速な手段によって、対象となるシンボルに関して特定の側面を明らかにする包括的なグラフを生成できます。

Analyze ダイアログでは、一度につき 1 つの特定シンボルにフォーカスします。そのシンボルについて、一連の強力なクエリがリストされています。これらのクエリはシンボルの型に固有であり、多くの場合シンボル自体にも固有です。例えば、クラスのクエリには、そのクラスの用法を調べるいくつかのグラフが含まれています。クラスに public の変数メンバがある場合、これらのグラフにはそれが含まれ、変数メンバがクラスの外部の関数によってセットされるか読み取られる場所を示します。

クエリのいくつかは、第 2 のシンボルとの関係を調べます。例えば、関数に対するクエリの 1 つでは、ターゲットのシンボルとユーザが選択した第 2 の関数との間にある、あらゆる呼出しパスを特定します。

時として、最初に得られたグラフによって、関心のある特定側面を理解するのに必要なすべての情報が提供されることがあります。そうでない場合には、自動生成されたグラフを、ソフトウェアをより深く解析するための出発点として利用します。

レポート及びメトリックス

［Tools］及び［Reports］メニューからは、いずれも複数の情報ディスプレイウィンドウにアクセスできます。Imagix 4D のデータベースには、ソフトウェアから収集された広範なデータが含まれており、これらのディスプレイウィンドウにより、ユーザインターフェイスを通じてデータを利用できます。この 2 つのセットの違いは、［Tools］メニューからアクセスできるディスプレイウィンドウが、このデータをある程度直接に視覚化するために使用し、コードの特定の部分を特定できることです。

これに対し、［Reports］メニューから使用できる Reports ウィンドウと Metrics ウィンドウでは、プロジェクトの全体についての情報が表示されます。これらの情報はフォーカスされるためのものではなく、多くの場合、最初のディスプレイが表示される前に、Imagix 4D がすでに収集された広範なデータの解析と処理を行う必要があります。これらの［Report］メニュー項目の目的は、一般的にソフトウェアの実装の質に関する測定及びチェックなど、特定の問題についてプロジェクト全体の観点からの情報を提供することです。

Metrics ウィンドウは、さらにレビューの必要な領域を特定することができるようにシンボルをそのソフトウェアメトリックスに従ってソートし、ランク付けします。File レポート及び Class Summary レポートは、データベース全体から抽出したデータを表にまとめ、ソフトウェアについて高いレベルの統計値を提供します。Source Check レポートは設計上及びコーディング上の例外を拾い出します。変数、関数、及びタスクフローチェックでは、特に組み込みのリアルタイムソフトウェアについて、潜在的なデータ及び制御フローに関する問題が特定されます。Include Analysis レポートでは、ファイルとヘッダファイルの依存関係に関する包括的な解析結果が表示されます。

これらのディスプレイウィンドウ自体ではコードの特定の部分へのフォーカスは行われませんが、Imagix 4D の標準ナビゲーションがサポートされており、Tool ディスプレイウィンドウで 関心のある領域をさらに調べることができます。Import Report 機能により、このナビゲーションを以前に保存したレポートに対して適用することができます。

Metrics ウィンドウ

Metrics ウィンドウは、Imagix 4D がソフトウェアについて行う定量的測定の結果を解析する主なディスプレイウィンドウです。ここに表示されるディレクトリ、ファイル、名前空間、クラス、関数、変数レベルのメトリックスについては、*本ユーザガイド*の「データモデル」の項に説明があります。

Metrics ウィンドウでは、特定の型の全シンボルを、解析したいと思っているソフトウェアメトリックスの集合によってリスト、ソート、比較、ランク付けすることができます。メニュー項目 Format の選択肢では、行おうとしている解析に適した表示形式を選択することができます。Graph を選択したときに表示される色は、それぞれのメトリックスが上位及び下位の閾値に対してどの範囲にあるかを示しています。

これらの閾値は Metrics ウィンドウでも（LOCAL メニュー［Display］> ［Threshold］）、メインの Graph ウィンドウの Options ダイアログ（メニュー［Tools］> [Options] > [Metrics] > [Threshold]）でも設定することができます。ここで設定した閾値は、Graph ウィンドウを始め、Imagix 4D のあらゆるメトリックス表示に適用されます。自社の基準に合わせてこれらの閾値を設定しておくとよいでしょう。

ソフトウェアメトリックスはまず第 1 に、コードのサイズ、設計、または複雑度が理解、向上、試験を難しくしているところなど、コードの潜在的問題領域を特定するために利用することができます。開発期間中、時間をかけてメトリックスを追跡していれば、開発の進捗度をはかり、潜在的な品質の問題を早期に発見し、それをより容易に修正することができます。ソフトウェアメトリックスは、ソースチェックと組み合わせて利用することにより、ソフトウェア及びプロセスの全体的品質の向上に役立ちます。

File Summary 及び Class Summary

File Summary はプロジェクトに含まれるファイルの概要を提供します。型及びディレクトリによって集計したデータを表にまとめ、コード行数及びメンバ数の 2 点からファイルのサイズに関する情報を表示します。

Class Summary はプロジェクトに含まれるクラスの概要を提供します。型及びディレクトリによって集計したデータを表にまとめ、メンバ数の点からクラスに関する情報を表示します。

Source Checks（ソースチェック）

Source Checks では、本ユーザガイドの「データモデル」の「ソースチェック」の項で説明した、ファイルベースのソースチェックの各項目に対応するレポートを選択することができます。これらのソースチェックは設計上及びコーディング上の例外を拾い出します。各レポートは、プロジェクトの全ファイルにまたがって、特定のソースチェックの全例外をリストします。

これらとは別に、File Editor メニュー及びファイル上でのマウスの右クリックポップアップメニューで利用可能なソースチェックは、特定のファイルについて、すべてのソースチェックの例外をリストします。File Editor ウィンドウでも、ソースチェックの例外はアンダーラインを付けて表示されます。ソースチェックは、ソフトウェアメトリックスと組み合わせて利用することにより、ソフトウェア及びプロセスの全体的品質の向上に役立ちます。

Variable Flow Checks（変数フローチェック）

変数フローチェックは、ソフトウェアにおける変数の使用に関する潜在的な問題を特定する、レポートのセットです。関数内（ローカルデータフロー解析）及び関数呼び出し（グローバルデータフロー解析）におけるプログラム制御フロー、さらに変数の代入及び渡されるパラメータが追跡され、チェックされた変数の完全なデータフローが判断されます。

Unused Variables（未使用変数）

未使用変数レポートは、3 つのレポートをまとめたものです。未設定の、読み込まれていない、または完全に使用されていない変数をレビューするかどうかを選択できます。これら 3 つの選択肢により、コード内の異なる種類の潜在的な問題を特定できます。たとえば、次のようなソースコードがあるとします。

```
int globalA, globalB;

int func2(int paramX, int paramY) {

paramY = globalB;
globalA = paramY;
return globalB;
}

int func1(void) {
int localW = 1;
int localX, localY, localZ;

return func2(localX, localY);
}
```

レポートウィンドウのメニューバーの〝Display〟を、変数について never used、never read、never set のどれに設定したかによって、生成されるレポートの内容は大きく異なります。never used（未使用）に設定すると、ソースコードで読み込みも書き込みもされていない変数が特定されます。レポートには、宣言された場所に関する情報と合わせて、各変数がリストされます。これらはにせの宣言である場合もあります

が、レビューを行うことで、チェックしたソフトウェアでの名前の不一致など重大な問題が発見される場合があります。

```
Variables Which Are Never Used

Settings:
       Usage Type:         never Used
       Global Variables:   displayed
       Static Variables:   displayed
       Local Variables:    displayed

Summary
  Variables         Total   Set Only  Read Only   Unused
Global                  2        1        1        0
Static                  0        0        0        0
Local                   6        2        2        1


Variable                                              File (Line)

localZ                                                unused_vars.c (13)
```

次の選択肢であるセットのみの変数は、重要なデータが使用されていない状態を示している場合があります。この場合、変数の範囲によってフィルタをかけることができます。ここでは、システムの一部だけをチェックする場合のように、レポートからグローバル変数が除外されます。

```
Variables Which Are Set Only

Settings:
       Usage Type:         set only
       Global Variables:   omitted
       Static Variables:   displayed
       Local Variables:    displayed

Summary
  Variables       Total   Set Only Read Only   Unused
Static                0        0        0        0
Local                 6        2        2        1


Variable                                              File (Line)

localW                                                unused_vars.c (12)
paramX                                                unused_vars.c (4)
```

レポートに読み込まれているがセットされていない変数が表示されるように設定することで、読み込まれたときに予期しない結果が発生しうる、初期化されていない変数などの問題を発見することができます。

```
Variables Which Are Read Only

Settings:
        Usage Type:        read only
        Global Variables:  displayed
        Static Variables:  displayed
        Local Variables:   displayed

Summary
  Variables       Total   Set Only Read Only   Unused
Global                2        1        1        0
Static                0        0        0        0
Local                 6        2        2        1
```

```
Variable                              File (Line)

globalA                               unused_vars.c (2)
local W                               unused_vars.c (12)
paramX                                unused_vars.c (4)
```

Uninitialized Variables Read（未初期化読み取り変数）

Uninitialized Variables Read レポートは、グローバル変数、静的変数およびローカル変数のうち、初期化またはセットされる前に読み取られている変数を示します。このような状態は単に変数の初期化失敗によるものであるか、ロジックの欠陥やロジックの不足を示している可能性があります。次のような例を考えてみましょう:

```
int globalA, globalB, globalC;

int func2(int paramW, int paramX) {
        int localA;
        localA = globalA;
        localA = globalB;
        localA = globalC;
        return localA;
}

int func1(void) {
        int localW = 1;
        int localX, localY, localZ;
        int decision = 1;
        globalA = localW;
        if (decision) {
            globalB = localW;
            globalC = localW;
        } else {
            globalC = localW;
        }
        localY = func2(localW, localX);
        return localZ;
}
```

localX および localZ は初期化されないまま func1 のなかで読み取られているため、結果となるレポートにリストされています。同様に変数 globalB も、その初期化が if 条件に依存するため報告されますが、globalC は if 条件に合致するかどうかにかかわらず初期化されます。

```
Uninitialized Variables Read

Settings:
        Global Variables:          displayed
        Static Variables:          displayed
        Struct Container Summary:  omitted
        Union/Bitfield Members:    separate

Task Definitions
Name                Members  Root
autotask 0 - func1        2  func1

Variable                              File (Line)
    Function                              File (Line)
       Assignment
```

```
globalB                              uninit_vars_read.c (1)
    func2                                uninit_vars_read.c (3)
        6 localA = globalB;

localX                               uninit_vars_read.c (12)
    func1                                uninit_vars_read.c (10)
       22 localY = func2(localW, localX);

localZ                               uninit_vars_read.c (12)
    func1                                uninit_vars_read.c (10)
       23 return localZ;
```

タスク定義がこのレポートに含まれていることに注意してください。レポートはデータフローの出発点を決定する際にタスク定義を考慮に入れます。ただし、このレポートは Task Flow Check レポートに属するものではないため、手動でタスクを定義する必要はありません(ユーザガイドの Task Flow Check セクション参照)。

Useless Assignments（無用の代入）

Useless Assignments レポートには、グローバル、静的、ローカルの各変数への代入が示されます。これらの代入値は、変数が読み込まれる前に再度セットされたか、または設定後に変数が読み込まれていないために、使用されません。無用の代入は、誤ったまたは欠落したロジックがあることを示す場合があります。次のような例を考えてみましょう。

```
int globalA, globalB;

int func2(int paramX, int paramY) {

    paramY = globalB;
    globalA = paramY;
    return paramX;
}

int func1(void) {
    int localX, localY, localZ;

    localX = 1;
    localY = 1;
    localZ = 1;
    globalB = 2;

    localZ = func2(localX, localY);

    return localZ;
}
```

デフォルトでは、レポートには読み込まれる前に変数自体が再度設定されるか（localZ）、読み込まれることがない（globalA）、特定の変数の代入 が示されます。これらは直接的に無用な代入であると見なされます。

```
Useless Assignments

Settings:
        Global Variables:        displayed
        Static Variables:        displayed
        Local Variables:         displayed
        Transitively useless:    not displayed
        Union/Bitfield Members:  separate
```

```
Variable                                                    File
    Function                                                    File
       Assignment

globalA                                                     useless_asgn.c
    func2                                                       useless_asgn.c
        8 globalA = paramY;

localZ                                                      useless_asgn.c
    func1                                                       useless_asgn.c
       17 localZ = 1;
```

オプションとして、間接的に無用な代入がチェックされるようにレポートを設定できます。この場合、paramY 及び globalB の代入がレポートに含まれます。func2 における globalA の代入には、直接的に無用です。paramY の代入は globalA の無用の代入にのみ使用されるため、上流におけるこの代入は間接的に無用と見なされます。さらに 1 段階上流では、paramY の値の設定での globalB の使用も同様になります。

```
Useless Assignments

Settings:
        Global Variables:         displayed
        Static Variables:         displayed
        Local Variables:          displayed
        Transitively useless:     displayed
        Union/Bitfield Members:   separate

Variable                                                    File
    Function                                                    File
       Assignment

globalA                                                     useless_asgn.c
    func2                                                       useless_asgn.c
        8 globalA = paramY;

globalB                                                     useless_asgn.c
    func1                                                       useless_asgn.c
       18 globalB = 2;

localZ                                                      useless_asgn.c
    func1                                                       useless_asgn.c
       17 localZ = 1;

paramY                                                      useless_asgn.c
    func2                                                       useless_asgn.c
        7 paramY = globalB;
```

Function Flow Checks（関数フローチェック）

関数フローチェックには、コード内の関数呼び出し階層を解析する、2 つのレポートが含まれます。これらのチェックによってレポートされた結果は、必ずしもソフトウェアの問題を示すものではありません。代わりに、ソフトウェアの制御フローにおける潜在的な関心領域が特定され、その後のレビュー対象の決定に役立ちます。

Recursive Functions（再帰関数）

再帰関数レポートでは、直接またはその他の関数呼び出しを通じて再帰的に呼び出される、すべての関数が検出されます。再帰関数を使用することで実装が簡単になりますが、無限ループが発生しないように注意し、再帰終了条件が適切であることを確認する必要があります。組み込みソフトウェアについては、スタックのオーバーランの危険性があるため、再帰関数は特に注意して使用することが重要です。レポート結果をレビューすることで、再帰の最大数が予測及びテストされ、十分なスタックスペースが確保されるようにすることができます。次のような例を考えてみましょう。

```
void funcD() {
    return;
}

void funcC() {
    funcA();
    funcD();
}

void funcB(int paramX) {
    while (paramX < 10) paramX = funcB(paramX);
    funcC();
}

void funcA() {
    funcB(1);
    funcC();
}
```

このソースコードの再帰を表示する場合、再帰関数レポートには、直接または間接的に再帰するすべての関数と、再帰が発生するパスがリストされます。このリストには表形式が使用されます。それぞれの再帰パスのリストでは、再帰に関係する関数だけが表示されます。それぞれの再帰関数について、複数の再帰パスがある場合があります。

```
Recursive Functions

Function                                        File (Line)
          Recursive Path

funcA                                           recursive_funcs.c (16)
      0   . funcA
      1   . . funcB
      2   . . . funcB                                   see (1)
      3   . . . funcC
      4   . . . . funcA                             recursion
      5   . . funcC                                 see (3)

funcB                                           recursive_funcs.c (11)
      0   . funcB
      1   . . funcB                                 recursion
      2   . . funcC
      3   . . . funcA
      4   . . . . funcB                             recursion
      5   . . . . funcC                             see (2)

funcC                                           recursive_funcs.c (6)
      0   . funcC
      1   . . funcA
      2   . . . funcC                               recursion
      3   . . . funcB
      4   . . . . funcC                             recursion
```

```
5   . . . . funcB                                   see (3)
```

Shared Functions（共用関数）

Shared Functions レポートでは、ハイライトされた複数の関数から直接的または間接的に呼び出されたすべての関数がレポートされます。これにより、ハイライトされた関数の依存関係を確認することができます。これは、さまざまなタスクのルート関数からではなく選択した関数から解析が開始することを除き、タスクフローチェックの再入関数レポートに類似しています。このレポートを実行するには、最初にグラフウィンドウのルート関数をハイライトする必要があります。

Unused Code(未使用のコード)

Unused Code レポートでは、プロジェクトのルート関数がすべてリストされます。これらは、コード内の他のどの関数からも呼び出されない関数であり、ユーザが定義したどのタスクのエントリ関数でもないことから、デッドコード領域への入り口となる可能性があります。これらの関数のいくつかは、その実行モジュール自体への入り口であるかもしれません。あるいは、静的に決定できない関数ポインタを通して呼び出されている可能性があります(本ユーザガイドの「解析上の注意点」の項参照)。そのため手動によるレビューが必要ですが、レポートにはデッドコードへの入り口となりうるすべての箇所がリストされます。

また、レポートには、各ルート関数から到達した未使用のコードの有効範囲が示されます。Total Functions と Total Lines の値には、ルート関数自体とそれが唯一呼び出す関数の両方が含まれます。これらは、その 1 つのルート関数からの呼び出しによってのみ到達する、さらに他の関数です。この有効範囲は、レポート内のリンクを通してグラフィカルにも表示されます。

Task Flow Checks（タスクフローチェック）

タスクは、組み込みのリアルタイムソフトウェアシステムの操作で、重要な役割を果たします。タスクフローレポートは、ソフトウェアのタスクの実装と相互関係についてチェックする、一連のレポートです。タスクの連動は複雑で、問題の原因になる場合があります。これらのチェックによってレポートされた結果は、必ずしもソフトウェアの欠陥を示すものではありません。代わりに、タスク間の潜在的な矛盾が特定され、その後のレビュー対象の決定に役立ちます。

タスクは C/C++プログラムで明示的に指定されるものではないため、これらのレポートでは Task Definitions ダイアログからタスクを定義できます。各タスクについてルート関数を指定すると、それぞれのタスクはルート関数、及びそのルート関数から直接または間接的に呼び出されたすべての関数で構成されると見なされます。Task Definitions ダイアログでタスクを定義しない場合、プロジェクト内のルート関数ごとにタスクが自動的に定義されます。

このレポートでは、割り込み禁止またはセマフォによって保護されている、ステートメントのクリティカルな領域や範囲も示されます。これらについても、ソースコードでは明示的に指定されていません。Critical Region Definitions ダイアログで、クリティカルな領域に入る（割り込み禁止/セマフォ）、またはクリティカルな領域を出る（割り込み許可/セマフォ）ために使用される関数を指定できます。Mismatched Critical Regions レポートでは、クリティカルな領域に入る／出るための関数を定義する必要があります。その他のレポートでは、クリティカルな領域に入る／出るための関数の定義は任意です。

マルチタスクシステムにおける、同期のためのイベントの使用は、2 つの特定のタスクフローチェックレポート、Event Calls in Tasks および Event Transition Between Tasks で解析されます。タスクやクリティカルな領域と同様に、イベントメカニズムも C/C++プログラムで明示的に示されていないため、これらのレポートを実行する前に特定しておく必要があります。それは、Event Definitions ダイアログから実行します。このダイアログで、オペレーティングシステムのイベント関数を指定します。タスクからの呼び出しにより、これらの関数を使用して待機(pend)、解放(post)、およびクリアのアクティビティが処理されます。またこの

ダイアログではイベントと、イベント間通信で使用されるオペレーティングシステム共有リソースに対するソースコードレベルの識別子(マクロ、列挙リテラル、および定数変数)を指定します。

Task Flow Check レポートを実行した後は、Imagix 4D の全機能を利用して、ソフトウェアを調査し、レポートでフラグされている問題を理解することができます。特別な設定により、タスクで使用されていない関数を特定することによって、この機能は Graph ウィンドウおよび File Editor の両方で利用できます。

Variables Set in Multiple Tasks レポート

Variables Set in Multiple Tasks レポートでは、複数のタスクによるグローバル変数及び静的変数の使用方法が報告されます。これは一組のレポートとして考えることもできます。セットのアクセスのみをレビューするか、セットおよび読み取りのアクセスをレビューするかを選択できます。

レポートの主要な利用目的は、セットのアクセスのみを調べることです。そのような変数への代入により、それらの変数を使用するタスクがクリティカルな領域で保護されていない場合に、予期しない動作が発生することがあり、それが不具合となる可能性があります。

```
int globalA, globalB, globalC;

int subX(int paramX) {
    globalB = paramX;
    /* some calculations */
    paramX = globalB;
    return paramX;
}

void taskX() {
    int localX = 1;
    globalA = localX;
    /* some calculations */
    localX = globalA;
    DisableInt();
    localX = subX(localX);
    EnableInt();
    globalC = localX;
}

void taskY() {
    int localY = 1;
    globalA = localY;
    DisableInt();
    globalB = localY;
    EnableInt();
    localY = globalC;
}
```

セマフォ、または DisableInt や EnableInt などの割り込み制御関数を使用して、クリティカルな領域を保護することで、グローバル変数値書き換えという他のタスクからの予期しない干渉から保護されます。たとえば、subX で読み込まれる globalB の値は、subX が taskX の DisableInt で保護されるため、subX で先に globalB に代入された値と同じになります。その結果、taskY による割り込みによって、subX の globalB の設定と読み込みの間で発生する計算中に globalB が変更されることはありません。

これに対して、taskX における globalA の設定と読み込みは保護されません。書き込みと読み込みの間で taskY による割り込みが発生した場合は、globalA の値が修正され、taskX の予定された動作に干渉します。

Variables Set in Multiple Tasks レポートでは、変数の代入がクリティカルな領域で発生するかどうかの解析が行われ、その情報がディスプレイに表示されます。保護されている変数の使用をフィルタで除外す

ることもできます。この場合は、保護されている変数の集合と保護されていない変数の集合の両方が表示されます。指定された用法でのアクセス状況は、U(保護されていない)または P(保護されている)によって示されます。保護されたアクセスについては、クリティカルな領域の名前が示されます。ここでは 1 つのクリティカルな領域が Critical Region Definitions ダイアログで定義され、int という名前が付けられています。

```
Variables Set in Multiple Tasks

Settings:
        Critical Region:           int ( DisableInt / EnableInt )

        Usage Type:                set only
        Protected Variables:       displayed
        Unprotected Variables:     displayed
        Global Variables:          displayed
        Static Variables:          displayed
        Struct Container Summary:  displayed
        Union/Bitfield Members:    separate

Variable                           File
          Task
             Line Number of Usage
                 Critical Region
                           User of Variable

globalA                            multi_set_vars.c
          Task X
              13 U         taskX
          Task Y
              24 U         taskY

globalB                            multi_set_vars.c
          Task X
               5 P   (int) subX
          Task Y
              26 P   (int) taskY
```

レポートのもう 1 つの用途は、複数のタスクにおける変数へのセットおよび読み取りのアクセスを解析することです。この用途での解析結果は、必ずしも潜在的な問題を示すとは限りませんが、タスク間で共有される変数の用法と保護の状態について、有用で総合的な見識を示します。

```
Variables Used in Multiple Tasks

Settings:
        Critical Region:           int ( DisableInt / EnableInt )

        Usage Type:                set or read
        Protected Variables:       displayed
        Unprotected Variables:     displayed
        Global Variables:          displayed
        Static Variables:          displayed
        Struct Container Summary:  displayed
        Union/Bitfield Members:    separate

Variable                           File
          Task
             Usage Type
                 Line Number of Usage
                      Critical Region
                                User of Variable
```

```
globalA                                multi_set_vars.c
          Task X
             S     13 U          taskX
             R     15 U          taskX
          Task Y
             S     24 U          taskY

globalB                                multi_set_vars.c
          Task X
             S      5 P    (int) subX
             R      7 P    (int) subX
          Task Y
             S     26 P    (int) taskY

globalC                                multi_set_vars.c
          Task X
             S     19 U          taskX
          Task Y
             R     28 U          taskY
```

セットと読み取りの両方が解析されると、アクセスの用法タイプに関する情報がレポートに追加されます。アクセスのタイプは S(セット)または R(読み取り)によって示されます。globalC などその他の変数は、あるタスクでのみ設定されて他のタスクで読み取られる場合にはリストに表示されます。

C/C++ の構造体と共用体、および C++ のクラスで構成されている変数を集計します。このレポート及び以下のいくつかのレポートには、集計された変数のレポート方法を制御するオプションがあります。メニューバーの Display→Include Struct Container Summary にチェックを付けると、これらの集計のいずれかがコンテナ変数の要約で使用されます。たとえば、構造体のメンバが 1 つのタスクに割り当てられ、別のメンバが別のタスクに割り当てられた場合、各メンバは Variables Set in Multiple Tasks レポートに表示されませんが、構造体変数は表示されます。

これに対して配列は、インデックスが動的に表されることから、通常はどの配列要素が割り当てられているかを識別できないため、これらのレポートでは配列全体を単一の変数として見なします。

静的な解析によって可能な範囲で、Imagix はどのオブジェクトがポインタに参照されているかを追跡します。ポインタまたは参照パラメータは、実際のパラメータ変数を追跡し、潜在的な変更を特定します。配列に設定されて配列のインデックスに使用されるポインタにより、配列が変更されます。Imagix ではポインタを「クリーンに」使用することが想定されています。この場合ポインタは特定の変数に設定され、その変数上だけで機能し、隣接するメモリ領域に割り当てられている個々の変数に副次的にアクセスすることはありません。

Reentrant Functions レポート

複数のタスクから呼び出される関数は、Reentrant Functions レポートにリストされます。グローバル、静的、またはスタティックローカルなどの、メモリ上に静的に割り当てられる変数を書き込むまたは読み込む関数を呼び出すと、同じ関数を呼び出すその他のタスクとの予期しない連動が発生することがあります。

```
int globalZ1, globalZ2;

void funcC() {}

void funcB() {
    int localB;
    globalZ2 = 2;
    /* some calculations */
    localB = globalZ2;
```

```
}

void funcA2() {}

void funcA1() {
    int localA1;
    globalZ1 = 1;
    /* some calculations */
    localA1 = globalZ1;
}

void funcA() {
    funcA1();
    funcA2();
}

void taskX() {
    funcA();
    DisableInt();
    funcB();
    EnableInt();
    funcC();
}

void taskY() {
    funcA();
    DisableInt();
    funcB();
    EnableInt();
    funcC();
}
```

Reentrant Functions レポートでは、メンバ関数、再入関数、呼び出し先関数という用語が使用されます。メンバとは、タスクルート関数、及びそのルートから直接または間接的に呼び出されるすべての関数など、タスクを構成する一連の関数です。共用関数(複数のタスクで共有されるメンバ関数)は、再入関数または呼び出し先関数のいずれかに分類されます。再入関数とは、1 つまたは複数の共有されないメンバ関数によって呼び出される共用関数です。これらの関数は、ルート共用関数と見なされる場合もあります。呼び出し先関数とは、他の共用関数によって呼び出される共用関数であり、そのため共用関数の呼び出し階層にエントリが示されません。

レポートでは、各再入関数のリストで、それぞれの再入関数を呼び出すタスクだけでなく、その関数がクリティカルな領域内から呼び出されたかどうかを示します。この解析では、タスク間の潜在的な連動のレビューが可能です。

```
Reentrant Functions

Settings:
        Critical Region:              DisableInt / EnableInt

        Protected Functions:          displayed
        Unprotected Functions:        displayed
        Functions not Using Globals:  displayed
        Library Functions:            displayed

Task Definitions
Name            Root                              Members  Reentr Callees
Task X          taskX                                   8       3       2
Task Y          taskY                                   8       3       2

Reentrant Function                                      File (Line)
```

```
    Callees
          Task
             Line Number of Usage
                 Protected / Unprotected Usage
                   User of Reentrant Function

funcA                                                  reentr_funcs.c (22)
    funcA1                                             reentr_funcs.c (15)
    funcA2                                             reentr_funcs.c (13)
          Task X
              28 U   taskX                               reentr_funcs.c (27)
          Task Y
              36 U   taskY                               reentr_funcs.c (35)

funcB                                                  reentr_funcs.c (6)
          Task X
              30 P   taskX                               reentr_funcs.c (27)
          Task Y
              38 P   taskY                               reentr_funcs.c (35)

funcC                                                  reentr_funcs.c (4)
          Task X
              32 U   taskX                               reentr_funcs.c (27)
          Task Y
              40 U   taskY                               reentr_funcs.c (35)
```

ここでは、レポート結果にフィルタをかけて、問題が発生する可能性が高い再入関数にフォーカスすることができます。グローバル変数などのメモリ上に静的に割り当てられる変数を使用しない funcC などの関数は安全であり、デフォルトではレポート結果から除外されます。

グローバル変数が使用される可能性は、クリティカルな領域を使用することで保護されます。たとえば、funcB への呼び出しに対する割り込み保護は、globalZ2 が設定から使用までの間に異なるタスクによって変更されまたは読み込まれないように保護します。一般的には、保護されているクリティカルな領域内で再入関数を使用するほうが、保護されていない領域で使用するよりも安全であり、同様にフィルタで除外できます。これら 2 つのフィルタが適用された、同じレポートの例を示します。

```
Reentrant Functions

Settings:
        Critical Region:              DisableInt / EnableInt

        Protected Functions:          omitted
        Unprotected Functions:        displayed
        Functions not Using Globals:  omitted
        Library Functions:            displayed

Task Definitions
Name            Root                              Members  Reentr Callees
Task X          taskX                                   8       1       2
Task Y          taskY                                   8       1       2

Reentrant Function                                     File (Line)
    Callees
          Task
             Line Number of Usage
                 Protected / Unprotected Usage
                   User of Reentrant Function

funcA                                                  reentr_funcs.c (22)
    funcA1                                             reentr_funcs.c (15)
```

```
Task X
    28 U   taskX                          reentr_funcs.c (27)
Task Y
    36 U   taskY                          reentr_funcs.c (35)
```

Functions Not Used in Tasks レポート

Functions Not Used in Tasks レポートには、現在定義されているタスクで呼び出されていない、プロジェクトデータベース内の関数がリストされます。

```
void funcA() {return;}
void funcB() {return;}
void funcC() {return;}
void funcD() {return;}
void funcE() {return;}

void Task1() {
    funcA();
    funcB();
}

void Task2() {
    funcB();
    funcC();
}

void Task3() {
    funcC();
    funcD();
}
```

このレポートは、タスクが[Task Definitions]ダイアログで明示的に定義されていることを必要とします。定義されていないと、すべてのルート関数は自動的にあるタスクのルート関数となり、どの関数も未使用であると見なされなくなります。この解析では、タスクは関数の Task1 および Task2 に対してのみ定義されています。したがって、関数 Task3 からのみ呼び出された関数はすべて未使用であると見なされます。

```
Functions Not Used in Tasks

Task Definitions
Name        Members  Root
Task 1            3  Task1
Task 2            3  Task2

Function                          File

funcD                             test.c
funcE                             test.c
Task3                             test.c
```

実際の Functions Not Used in Tasks レポートに加えてこの解析の結果も、オプション設定により、Graph ウィンドウおよび File Editor にある関数を特定するために利用できます。タスク解析の対象外であった関数が識別できることは、これらのウィンドウを使用してソフトウェアを調査し、様々な Task Flow Check レポートにフラグされた問題を理解するうえで役立つでしょう。

Mismatched Critical Regions レポート

タスク内での、割り込み呼び出しまたはセマフォの許可/禁止に関係する潜在的な問題は、Mismatched Critical Regions レポートによって検出されます。解析では、クリティカルな（保護された）領域の開始と終

了の不一致が指摘されます。またこのレポートでは、コード内の特定の行がクリティカルな領域にあるかどうかが不明確な箇所が特定されます。このような問題によって、保護されていない割り込みに対してソフトウェアシステムが無防備になり、またロジックに関するその他の問題が発生する場合があります。クリティカルな領域は、関数呼び出しの開始と終了を指定することで定義します。プロジェクト内のそれぞれの開始関数呼び出しについて、レポートはプログラムまたはタスクの終点まですべての実行パスをチェックして、マッチする終了関数呼び出しを検索します。欠落しているものがある場合は、レポートには開始呼び出しの場所と、終了呼び出しが欠落した最も近いパスが示されます。次に、レポートは 終了関数呼び出しからすべての実行パスを逆行して調べ、プログラムまたはタスクの先頭に戻るまで、マッチする開始関数呼び出しを検索します。見つからない場合は、レポートは終了関数呼び出しの場所と、開始呼び出しが欠落した最も近いパスを示します。

当該行へのパスによってソースコード内の行の保護状態が異なる場合、不明確な箇所が生じます。レポートでは、この解析の一部として、呼び出し先または呼び出し元となるすべての関数がチェックされます。

```
int globalX;

void funcC() {
    int localC = 1;
    if (localC) {
        globalX = 1;
        EnableInt();
    } else {
        globalX = 2;
    }
}

void funcB() {
    int localB = 1;
    if (localB) {
         DisableInt();
        globalX = 1;
    } else {
        globalX = 2;
    }
}

void funcA() {
    funcB();
    funcC();
}
```

上記の例では、クリティカルな領域への出入りでは特定の条件パスしか保護されていないため、いずれも問題があります。結果のレポートには両方の問題が表示され、開始または終了が欠落している最も近いパスが特定されます。これを基に、不一致のレビューを開始することができます。

```
Mismatched Critical Regions

Settings:
        Critical Region Critical Region:         CR ( DisableInt /
EnableInt )

        Regions Missing Starts:  displayed
        Regions Missing Ends:    displayed
        Ambiguous Functions:     displayed
        Ambiguous Regions:       displayed

Start of Critical Region            Missing End of Critical Region
Function         Line File          Function          Line File
funcB              17 mm_cr.c       funcB               22 mm_cr.c
```

```
Missing Start of Critical Region    End of Critical Region
Function         Line File          Function          Line File
funcC               7 mm_cr.c   funcC                    8 mm_cr.c

Ambiguous Functions
Function         Line File
funcC               4 mismatched_cr.c

Ambiguous Regions
File             Lines
mismatched_cr.c  (all) 5 6 7 10 11 21 25 26
```

Variable Flow Between Tasks レポート

Variable Flow Between Tasks レポートでは、タスク間の通信に使用されるグローバル変数と静的変数がレポートされ、変数を複数回設定または読み込むなど、潜在的に疑わしい使用方法が検出されます。疑わしい使用方法の 1 つに、フィードバックループがあります。フィードバックループは、あるタスクが次のタスクによって読み込まれるグローバル変数を設定する、タスクとグローバル変数の相互作用です。この 2 番目のタスクが次に別の(または同じ)変数を設定し、それが次のタスクによって読み込まれます。これは、最初のタスクが再度関係するまで続行されます。

他のフローチェックレポートと同様に、この解析によって必ずしも問題が提示されるとは限りませんが、タスクの相互作用のレビューに有益な情報が得られます。次のようなコードが考えられます。

```
int globalA, globalB, globalC, globalD;
int globalE, globalF, globalG;

void taskX() {
    int localX1, localX2 = 1;
    globalA = localX1;
    globalB = localX1;
    globalC = localX1;

    localX2 = globalG;
    globalE = localX2;
}

void taskY() {
    int localY1, localY2 = 1;
    globalA = localY1;
    localY1 = globalB;
    globalC = localY1;

    localY2 = globalE;
    globalF = localY2;
}
```

この例では変数がタスク間で共有されていますが、レポートに表示されるように、実際にフィードバックループはありません。

```
Variable Flow Between Tasks

Key:
        R or RR:          variable is read once (R) or multiple times
                               (RR) in task
        S or SS:          variable is set once (S) or multiple times
                               (SS) in task
        R,S or RR,SS:     variable is always read before being set
```

```
                              in task
        R,*:                variable is read first in at least one path;
                              it is set and read in various orders
        S,*:                variable is set first in all paths where it
                              is read; it is set and read in various
                              orders thereafter
        Fn or Fn..:         variable and task involved in feedback loop
                              labeled n (and others if ..)

Settings:
        Critical Region:                no critical regions defined

        Single Set / Single Read:       displayed
        No or Single Set / Multi Read:  displayed
        Multi Set / No or Single Read:  displayed
        Multi Set / Multi Read:         displayed

        Global Variables:               displayed
        Static Variables:               displayed
        Struct Container Summary:       omitted
        Union/Bitfield Members:         separate
        Feedback Loops:                 displayed

Variable                        Task1     Task2     Task3 ...

                                Task X
                                   .      Task Y
-------------------------------   =         =
globalA ......................  S         S
globalB ......................  S         R
globalC ......................  S         S
globalE ......................  S         R
```

次に示すコードでは、より多くのタスク間で多数の変数が共有されており、フィードバックループの可能性があります。

```
int globalA, globalB, globalC, globalD;
int globalE, globalF, globalG;

void taskV() {
    int localV1 = 1;

    globalD = localV1;
    localV1 = globalD;
}

void taskW() {
    int localW1 = 1;

    globalD = localW1;
    localW1 = globalD;
}

void taskX() {
    int localX1, localX2 = 1;
    globalA = localX1;
    globalB = localX1;
    globalC = localX1;

    localX2 = globalG;
    globalE = localX2;
```

```
}

void taskY() {
    int localY1, localY2 = 1;
    globalA = localY1;
    localY1 = globalB;
    globalC = localY1;

    localY2 = globalE;
    globalF = localY2;
}

void taskZ() {
    int localZ2;

    localZ2 = globalF;
    globalG = localZ2;
}
```

コードに潜在的なフィードバックループがある場合は、レポートに表示され、どのタスク/変数の使用方法がどのフィードバックループに対応するかが、F(x)で示されます。

```
Variable Flow Between Tasks

Variable           Task1     Task2     Task3 ...

                   Task V
                      .      Task W
                      .         .      Task X
                      .         .         .      Task Y
                      .         .         .         .      Task Z
-------------------   =         =         =         =         =
globalA ...........   .         .         S         S         .
globalB ...........   .         .         S F2      R F2      .
globalC ...........   .         .         S         S         .
globalD ...........  S,R F1    S,R F1     .         .         .
globalE ...........   .         .         S F2      R F2      .

globalF ...........   .         .          .        S F2      R F2
globalG ...........   .         .         R F2       .        S F2
```

Out of Step (Z)変数

通常、タスクでは変数は書き込んでから読み込みます。Out of Step 変数(Z 変数)は、読み込んでから設定するという実行順序になっているグローバル変数と静的変数を特定します。タスクベースのソフトウェアでは、計算で使用される変数が陳腐化する場合があります。

次のようなコードが考えられます。変数 globalA が初期化され、taskA が最初に呼び出された場合でも、globalA に値が与えられます。ただし taskA によって、完了前に globalA の値が更新されます。次に taskA が呼び出されると、globalA に読み込まれる値は前回の呼び出しの最後に存在した値になります。これが望ましい動作であることもありますが、本来は最初に globalA に(センサーの読み取りなどによって)代入される場合のプログラミングエラーであることもあります。

```
int globalA = 1;

int subX() {
    int localX;
    localX = globalA;
    return localX;
```

```
}

void subY(int paramY) {
    globalA = paramY;
}

void taskA() {
    int localA, localB;
    localA = subX();
    localB = localA;
    subY(localB);
}
```

Out of Step レポートではこれらの読み込みが検出され、タスク内のグローバル変数または静的変数の使用が設定されます。

```
Out-of-Step (Z) Variables (Read Before Being Set)

Key:
        r or R (letter):      variable is read
        s or S (letter):      variable is set
        S or R (upper case):  variable is used in current function
        s or r (lower case):  variable is only used in called functions
        r,s (order):          variable is read in at least one path
                                   before being set
        s,r (order):          variable is set in all paths before
                                   being read
        r,* (order):          variable is read first in at least one path
                                   it is set and read in various orders
        s,* (order):          variable is set first in all paths where it
                                   is read; it is set and read in various
                                   orders thereafter

Settings:
        Global Variables:          displayed
        Static Variables:          displayed
        Unset Variables:           displayed
        Struct Container Summary:  omitted
        Union/Bitfield Members:    separate


Variable                           Func1 Func2 Func3 (in order of calls)

Task:  Task A
                                   taskA
                                     .   subX
                                     .     .   subY
-----------------------------------  =     =     =
globalA ........................... r,s    R     S
```

Event Calls in Tasks レポート

Event Transition Between Tasks レポートと同様に、Event Calls in Tasks レポートは、マルチタスクシステムにおける、同期のためのイベントの使用をチェックするために使用されます。

イベントメカニズムは C/C++プログラムで明示的に示されていないため、両レポートでは最初にイベントメカニズムを特定する必要があります。イベント関数およびイベント自体は Event Definitions ダイアログで指定します。イベントは、イベント間通信で使用される、オペレーティングシステム共有リソースに対するソースコードレベルの識別子(マクロ、列挙リテラル、および定数変数)です。オペレーティングシステム

固有のイベント関数を使用して、イベントの待機(pend)、イベントの通知(post、現在のタスクに共有リソースへのアクセスを提供させる)、イベントのクリア(イベントに使用可能な通知をすべて削除し、すべてのタスクを再度待機状態にする)が行われます。クリアのイベント関数は、通知のイベント関数の変種とみなされ、特殊なパラメータ値によって、通知ではなくクリアを実行することが要求されます。

以下に示す、taskX が taskY に動作開始が可能になったことを単に通知する例を見てみましょう。

```
#define EVENTA 1

extern void PostEvent(int event);
extern void WaitEvent(int event);

void taskX() {
   // 割り込みで開始、各種の設定を実行
   PostEvent(EVENTA);
}

void taskY() {
   while (1) {
    WaitEvent(EVENTA);
    // 続いて動作を開始
   }
}
```

Event Calls in Tasks レポートでは、どのタスクがイベント関数の呼び出しを実行したかがイベント識別子によってまとめられています。この情報を使用して、各イベントが一連の正しいタスクを同期するために使用されていること、イベント通知に過不足がないことが確認できます。

```
Event Calls in Tasks
Key:
       P:                task posts this event
       W:                task waits for this event
       C:                task clears this event
Settings:
       Post event function:       PostEvent
       Wait event function:       WaitEvent
       Number of events defined:  1
Event                              File
         Task
            Line Number of Usage
                Usage
                  User of Event
EVENTA                             c601a.c
         taskX
              9 P taskX
         taskY
             14 W taskY
```

Event Transition between Tasks レポート

タスク間の同期における主要な問題の 1 つはデッドロック、つまりすべてのタスクが他のタスクによるイベント通知を永遠に待ちつづけることです。 Event Transition between Tasks レポートは、潜在的なデッドロックおよびその他の問題を識別するために役立ちます。このレポートは、タスクが相互に待機している条件を検証するためにプログラムをブラウズする出発点となります。

```
Event Transition Between Tasks
Settings:
       Post event function:       PostEvent
       Wait event function:       WaitEvent
       Number of events defined:  1
```

```
Events                                Task1     Task2     Task3 ...

                                    taskX
                                       .       taskY
-----------------------------------    =         =
EVENTA ..........................      P         W
```

レポートでは、マトリックス形式で、どのタスクが通知、待機、クリアを実行するかが示されます。このレポートを調べるにあたって、注目すべき点がいくつかあります。最低 1 つの通知呼び出しを持たないイベント(行)は、明らかにデッドロックの条件として考えられます。もう少し複雑なデッドロック発生要因は、2 つのタスク(列)が複数のイベント(行)を使用して通信を行っており、通知と待機が各行の異なるタスクにおいて起きている場合です。この場合、それらの制御フローおよび条件によって、タスクが互いに待機していないことが保証されるかをチェックしたいと考えるでしょう。Event Calls in Tasks レポートを使用して、個々の待機の呼び出し箇所をブラウズし、デッドロックが発生していないことが確認できます。

このイベントレポートには、待機イベントおよび通知イベントの関数呼び出しに使用されるイベントパラメータは、静的に決定可能である必要があるという制限があることに注意してください。これはつまり、パラメータがイベント自体(マクロ、列挙リテラル、または定数変数)、イベントを示す単純な式("EVENTA | EVENTB"など)、または現在の関数内にある一連のイベントに対して設定されるローカル変数のいずれかである必要があるということです。グローバル変数や現在の関数に渡されるパラメータをイベントパラメータにすることはできません。さらに、以下に示すとおり、関数ポインタの処理方法に関連する制限があります。

Flow Check レポート - 使用方法及び制限

タスクの指定

Task Flow Check レポートには、タスクはルート(またはエントリ)関数によって定義され、プログラムの中から明示的に起動することなく、プログラム内において実行可能とする制御フローが維持されます。通常は、タスクは同時実行(プログラムの別の部分の実行中に実行される)または準同時実行(不明な数のステップを実行し、別のタスクへの転送を制御する)にすることができます。

組み込みのリアルタイムのシステムについては、以下のケースをタスクとして考慮します。

-- 割り込み発生時に非同期に起動が可能な関数

-- 基盤となるスケジューラによって開始され、制御フローの完了前に停止が可能で、基盤のスケジューラの制御を別のタスクに移すことができる関数

-- プログラムの他の部分と(準)同時実行される可能性のある開始(またはルート)関数

次の関数は、単一のタスクとして、または別のタスクに分けて定義することができます。

-- スケジューラによって常に同じ順序で実行され、プログラムの他の部分による停止や割り込みが起こらない一連の関数

-- 常に起動時に他のタスクの開始前に実行される、システムを初期化する関数。初期化子関数は、オペレーティングシステムを定義する関数に含めることができます。

この定義に従うとプログラムに 1 つしかタスクがない場合は、いくつかの Task Flow Check レポートがあまり意味を持たないものになります。それでも、いくつかのレポートには価値があります。特に、Mismatched Critical Regions レポートはタスクのクリティカルな領域での問題を報告し、Out of Step(Z) 変数レポートは循環タスクでの潜在的な順序付けの問題を示します。

イベントの指定

イベント関連の Task Flow Check では、イベント定義に待機(pend)、解放（post）、およびクリアの操作を取り扱うために使用されるオペレーティングシステムのイベント関数の仕様が含まれます。通常、ポストとクリアは同じ関数により処理され、マクロまたは列挙リテラルがその関数に渡されて、ポストの操作ではなくクリアの操作が行われます。この機能に対応するため、Event Definition ダイアログが設定されています。

ポストとクリアに別々の関数が使われているオペレーティングシステムを使用している場合は、対処方法として、マクロをもう 1 つの引数に取るよう変更したクリア関数を定義します。新しいクリア関数を宣言した後、実際のクリア関数を新しい関数へと定義します。例えば、実際のクリア関数の名前を ClearEvent とし、1 つのパラメータを取るものとします。使用しているコンパイラ構成ファイルに以下の行を追加します。

```
void IMAGIX_ClearEvent(int event, int mode);
#define IMAGIX_CLR 1
#define ClearEvent(E) IMAGIX_ClearEvent(E, IMAGIX_CLR)
```

これを行うのに最適な場所は、新しい単独のファイルです。このファイルを作成した後、-inc オプションを使用して、すべてのソースファイルが実質的にインクルード(#include)されるようにします（上述した「アナライザの構文とオプション」の項を参照）。

これらの#define を追加すると、ソースアナライザが ClearEvent 宣言を処理する際に構文警告メッセージが出力されることがあります。しかし、結果となるデータベースには問題は生じません。

その後 Event Definitions ダイアログにおいて、IMAGIX_ClearEvent をポストイベント、IMAGIX_CLR をクリアイベントのパラメータとして指定します。これらの変更により、イベント関連の Task Flow Check ではポストの操作とクリアの操作とを区別することができるでしょう。

データ型の処理

C/C++の構造体と共用体は、C++のクラスと共に、集約変数というシンボル型のセットを構成します。Task Flow Check レポートの一部には、これらの集約変数がレポートされる方法を制御するオプションがあります。レポートオプションによって Container Summary をインクルードすると、集約変数のメンバがコンテナ変数の要約で使用されます。たとえば、構造体のメンバが 1 つのタスクに設定され、同じ構造体の別のメンバが別のタスクに設定された場合、Variables Set in Multiple Tasks レポートにはメンバが表示されませんが、構造体変数は表示されます。

これに対して配列は、レポートによって単一の変数として扱われます。その理由は、配列指標が動的な式となる場合があるからです。したがって、通常はどの配列コンポーネントが割り当てられているかを識別することは不可能です。

"Analyze Union/Bitfield Members Separately"オプションにより、解析において共用体またはビットフィールドの一部であるメンバをどう扱うかを制御します。このオプションを有効にすると、構造体の場合と同じように、共用体メンバは別の独立した変数として追跡されます。このオプションを無効にすると、配列の場合と同じように、メンバは同じ変数であるとみなされます。後者の場合、共用体メンバ un.member1 への代入は、un.member2 などのすべての共用体メンバへの代入として解釈されます。また、このオプションはビットフィールドにも適用されます。典型的なコンピュータ・アーキテクチャでは、ビットフィールドを割り当てると、メモリ内の語またはバイト全体を更新する必要が生じます。したがって、別のタスクから同時に別のビットフィールドへの割り当てがあると、この割り当てによって隣接する他のビットフィールドに影響を及ぼす可能性があります。

静的な解析によって可能な範囲で、これらのレポートはどのオブジェクトがポインタに参照されているかを追跡します。ポインタまたは参照パラメータは、実際のパラメータ変数を追跡し、潜在的な変更を特定します。配列に設定されて配列のインデックスに使用されるポインタにより、配列が変更されます。これら

のレポートでは、ポインタを「クリーンに」使用することが想定されています。この場合ポインタは特定の変数に設定され、その変数上だけで機能し、隣接するメモリ領域に割り当てられている個々の変数に副次的にアクセスすることはありません。

関数ポインタの処理

呼び出しに関数ポインタを使用するソースコードを解析する場合、これらのレポートはプログラム全体について、そのポインタに対して可能なすべての割り当てを追跡します。その結果、関数ポインタによるすべての呼び出しが同様に追跡され、グローバル解析で特定された関数のいずれかが呼び出されます。これによって、レポートで実際の結果を超える範囲の解析が行われます。たとえば、あるタスクで関数ポインタ変数 fp が func1 に設定されて呼び出され、次に別のタスクで func2 に設定されて呼び出された場合、これらのレポートでは func1 と func2 の両方が両方のタスクで呼び出されたと見なされます。したがって、func1 または func2 のいずれかで設定されたグローバル変数は、両方のタスクで表示されます。この関数ポインタによる呼び出しは、関数ポインタが関わるすべてのレポートに適用されます。

Useless Assignments	一部の無用な代入が認識されない場合があります。
Uninitialized Vars Read	変数の読み込みが誇張されている場合があります。
Recursive Functions	実際のプログラムでは発生しない再帰がさらにレポートされる場合があります。
Shared Functions	実際のプログラムでは発生しない共用関数がさらにレポートされる場合があります。
Vars Set in Multi Tasks	無関係の変数または無関係の場所がレポートされる場合があります。
Reentrant Functions	無関係の関数が再入関数として記録される場合があります。
Funcs Not Used in Tasks	未使用の関数がレポートされる場合があります。
Mismatched Crit Reg	無関係の不一致がレポートされる場合があります。
Var Flow Between Tasks	特定のタスクにおける変数の使用が誇張される場合があります。
Out of Step Variables	無関係の out-of-step 変数がレポートされる場合があります。
Event Calls	無関係の待機、通知、クリアがレポートされる場合があります。
Events Transitions	無関係の待機、通知、クリアがレポートされる場合があります。

要約すると、Useless Assignments レポートを除き、これらの無関係とされるインスタンスは偽の肯定ですが、レポートでは真の否定が見落とされることはありません。

解析メッセージ

一部の Flow Check レポートに、次のような警告が含まれることがあります。

```
   Statement or partial statement in file.x line xx not reachable and
eliminated from analysis
```

これらの警告はデータフロー解析の問題を示すものではありません。到達不能となっているコードの行を、それが意図されていない場合を考慮して指摘するだけです。このメッセージは以下のような場合に現れます。

-- 静的に真または偽に評価される条件で、もう一方の分岐にコードがある場合

-- goto、break または continue などの、制御の移動の後に現れるコード

警告メッセージは、以下の例の注記した行に対して発行されます。

```
#define TRUE 1
#define TEST TRUE

void funcA () {
    int condition1, condition2;
    if (TEST) {
        funcB();
    } else {
        not_reached();  /* 到達不能であると指摘される */
    }
    while (condition1) {
        if (condition2)
            return;
        else continue;
        also_not_reached();  /* 到達不能であると指摘される  */
    }
    return;
}
```

メモリ要件

関数制御ロジック内および関数の境界を越えるデータフロー解析では、Flow Check Report により集中的に計算が行われます。中規模のプログラムでさえ、処理時間とメモリ消費が多くなる可能性があります。

メモリの必要量が、お使いのオペレーティングシステムの 2GB または 4GB の制限を超えてしまうこともあるでしょう。このようなメモリ要件に対処するため、Imagix 4D データフローエンジンには独自の仮想記憶システムが含まれています。データフロー解析に使用されるメモリは、ハードドライブにページングされます。詳しくは、本ユーザガイドの「プロジェクトのリソース」の項に記述されています。

Include Analysis レポート

Include Analysis レポートには、あるファイルと、直接または間接的にインクルードされたヘッダファイルとの依存関係が表示されます。特定のファイルで使用されている各シンボルについて、レポートでは、必要な他のシンボルの定義または宣言が判断され、ヘッダファイルのインクルードを通じてこれらの定義または宣言がどのように検出されたかが解析されます。

Include Analysis レポートを解析することで、#include 階層を最適化し、コンパイル時間を短縮し、コードの再利用が簡単になります。

Similar Functions レポート

Similar Functions レポートは名前および/または構造の面で類似している関数を検出します。この情報は、類似したコードの統合をめざしたリファクタリングの努力をサポートします。類似した関数の特定は、ある関数へのバグ修正がほかの関数にも適用されるかを調べる際にも有効です。

関数の類似性は、関数名、クロスリファレンスのファンアウト、およびメトリックスの集合に対する類似性スコアを結合した公式によって判定されます。任意の 2 つの関数について、名前の類似性は文字列内の大文字/小文字を区別せずに計算されます。ファンアウトの類似性は 2 つの関数によって呼び出される関数の重複に基づきます。メトリックスの類似性については、McCabe Cyclomatic Complexity、Halstead

Program Difficulty、および Number of Statements の 3 つのメトリックスが考慮されます。メトリックス類似性スコアには、最も類似性の低いメトリックスが反映されます。

これらの個々の類似性スコアを結合する公式は、ユーザによる指定が可能です。このレポートはデフォルトでは、90%の類似した名前と 90%の類似したファンアウトを持つか、90%の類似した名前と 90%の類似したメトリックス値を持つか、90%の類似したファンアウトと 90%の類似したメトリックス値を持つ関数をリストします。この公式は Similarity Formula ダイアログを通して変更することができます。

レポートは 2 つの方法で実行できますが、Reports メニューから呼び出された場合には、プロジェクト内のすべての関数について類似する関数がチェックされます。この場合、内容の少ない関数が考慮されないように、名前の長さ、クロスリファレンスのファンアウト、および McCabe Cyclomatic Complexity に対して最小値を定義できます。

あるいは、Analyze ダイアログから Similar Functions レポートを実行することにより、ある特定の関数に類似した関数を識別することができます。Analyze ダイアログが非ライブラリ関数にフォーカスしているときは常に、このダイアログでの選択が可能です。

Import Report 機能

Import Report 機能は 2 つの用途に利用することができます。第一にこの機能では、実際のレポートを再実行することなく、適切に保存された Flow Check レポートを、それらが作成されたセッションとは異なるセッションで再オープンして表示することが可能です。第二に、適切にフォーマットされた外部レポートを、Imagix 4D の完全なソースコード調査機能と統合させ、Imagix 4D にロードして表示することが可能です。

Flow Check Report の保存と再オープン

Flow Check レポートの多くは、基盤となるデータフロー解析が広範に及ぶため、生成するのに多くの時間を要することがあります。結果は ASCII ファイルに出力可能ですが、色づけされたリンクによって提供される知識やナビゲーションは失われます。

Import Report 機能では、Imagix 4D での後の異なるセッションのときに、レポートを再度生成することなくレポートを再オープンする方法が提供されます。初期レポートの保存時に特殊なレポート形式(.4dr)を使用することによって、すべてのリンクに関する情報がレポートの内容そのものと共に保存されます。その後プロジェクトが開かれる際に、異なるマシン上の異なるユーザによって開かれる場合であっても、結果となる.4dr ファイルがインポートされます。

外部レポートのインポート

Import Report メカニズムは、外部ツールで生成されたデータと Imagix 4D とを統合するための簡単な方法を提供します。特に、外部データによってフラグされた問題を調査する目的での、Imagix 4D のソフトウェア情報把握機能の使用を容易にします。データのインポート後、レポート内に表示された項目からそれに対応するソースコードへと直接移動して、そこから Imagix 4D の使用を開始することができます。

Import Report 機能では、データが特定のフォーマットでカンマ区切りの(.csv)ファイルに存在している必要があります。次の例について考えてみましょう。

```
Anything here is ignored.
File,Line,Col3 Descr,Col4 Descr,Col5 Descr
C:/full/path/to/file.c,123,something,something else,third thing
C:/full/path/to/anotherfile.c,456,again,xxx,anything thing
```

.csv ファイルのデータ部には少なくとも 3 つのカラムが含まれていなければなりません。4 つ以上のカラムが存在する場合、インポート処理ではファイルと行番号の情報以外にどのカラムをインポートすべきか

が問われます。ファイルのフルパス名は最初のカラムに記述され、Imagix 4D のプロジェクトに使用されているパス名に対応していなければなりません。また、行番号は 2 番目のカラムに記述されていなければなりません。データ行の前に、データカラムの名前を含むヘッダ行（最初は"File,Line,"）を入れる必要があります。.csv ファイル内でこのヘッダ行より前にある行はすべて無視されます。

その他のディスプレイ

Main パネルのディスプレイウィンドウを補完する、2 つのウィンドウがあります。Main パネルの左側には Project ウィンドウがあり、プロジェクトに関する概要と、ソフトウェアの特定の部分にアクセスするためのナビゲーションが表示されます。Main パネルの下にある Symbol パネルには、1 回に 1 つの特定のシンボルについて、総合的な情報が表示されます。

これらの固定のパネルのほか、Information オーバレイには、いずれかのディスプレイウィンドウに表示されているシンボルおよび関係に関する概要が示されます。

Imagix 4D の最大の特長の 1 つは、これらのウィンドウがすべて相互に連動しているところです。これらのディスプレイを組み合わせて使用することで、ソフトウェアをすばやく解析して理解を深めることができます。

Project パネル

Project パネルではプロジェクトに関する詳細なレベルの概要が表示され、ソフトウェアのさまざまな側面を活用するための基点になります。複数のパネルによって要約情報が提供され、コードの特定の部分へのインデックスになります。その他のタブには、プロジェクト全体のクエリメカニズムが含まれています。

File Index タブには、Imagix 4D データベースにロードされているすべてのファイルのリストが表示されます。C コードと C++コードの場合、このなかには指定された特定の.c、.cpp ファイル、及びコンパイルの必要があるすべての.h ファイルが含まれています。 Java コードの場合、.java ファイルとそれらがインポートする.class ファイルの両方が含まれています。その他の一連のタブは、File Index の右へとスライドアウトすることができ、1 回に 1 つの特定のファイルに対してメンバ、関係、メトリックス、およびソースチェックの概要が示されます。

類似した 2 つのタブに、物理的コンテナではなく論理的コンテナのインデックスが示されます。1 つめのタイトルは、C/C++プロジェクトの場合は Namespace Index 、Java プロジェクトのロードの際は Package Index です。2 番目の Class Index パネルには、Imagix 4D データベースに含まれるすべてのクラス、テンプレート（Java の場合はインターフェイス）、名前空間、構造体、共用体がリストされます。File Index と同様に、プロジェクトの内容に関する概要を把握し、特定のシンボルをすばやく参照するために役立ちます。また File Index と同様に、右側にあるスライドアウトされた一連のタブに、特定の名前空間またはクラスに関するメンバ、関係、メトリックスの詳細情報を表示することができます。

コンテナではないシンボルを見るかナビゲートするには、4 番目のタブである Symbol Index を使用できます。ユーザが指定するあらゆるシンボルについて、プロジェクト内のその型のシンボルをすべて一覧表示します。

Database Lookup タブにはクエリメカニズムが提供されており、それによってプロジェクト内のシンボルについてより焦点が絞られたリストを生成することができます。自分の評価基準に合致するシンボルを特定するには、名前、シンボル型、およびスコープやメトリックス値など他の属性を使用したフィルタリングが可能です。

プロジェクト全体のクエリメカニズムの 2 つめは、Grep Tool タブです。Grep Tool は、Unix の grep コマンドを用いて、ソースコードに含まれる特定の文字列の場所を突きとめることを可能にします。grep をシェルからではなく Imagix 4D 内部から実行することにより、文字列を含むソースファイルを迅速かつ容易に見つけることができます。

現在の Graph ウィンドウに表示されたすべてのシンボルをリスト表示することによって、Graph Symbols タブは複雑になる可能性のあるグラフに簡単なインデックスを提供します。シンボルがリストされる他のタブ

と同様に、シンボルを単純なアルファベット順のリストにするか、またはファイルシステム内の位置に従ってまとめるかを制御することができます。

そして最後に Project Summary パネルには、プロジェクト全体に適用されるソフトウェアメトリックスの値が表示されます。これらのメトリックスにより、プロジェクトの範囲をすばやく理解することができます。また Main パネルの Metrics タブを使用して、特定のファイル、名前空間、クラス、関数、変数のメトリックスを表示して比較することも可能です。

Symbol Context セレクター

Project パネルのいくつかのタブでは、シンボルのリストをファイルシステムの位置に従ってまとめることができます。

Imagix 4D データベースは個々のシンボルが含まれる場所を追跡します。たとえば、変数 A はクラス B で定義されていて、クラス B はファイル C で宣言されていて、ファイル C はディレクトリ D に物理的に置かれているといったことを追跡します。データベースのシンボルはたいていコンテナを持っていますが、場合によっては(ルートディレクトリのように)コンテナがないこともあります。また、データソースが(システムライブラリ関数のように)コンテナに関する情報を含まない場合もあります。

インデックスタブの一部は、ほかのシンボルを含むシンボルの階層構造をビューするメカニズムを提供します。このメカニズムは、特定のインデックスタブの左上隅に表示されるコンテナ制御ボタン、ディレクトリおよび名前空間のボタンを通して管理されます。たとえば、Graph ウィンドウで関数や変数を調べていれば、Graph Symbols タブを使用して、そのさまざまなシンボルがどのクラスやファイルに含まれるかを見ることができます。

Slide-Out Container Info タブ

コンテナインデックスタブの右上にあるアイコンボタンを使用すると、付加的な一連のタブを含むスライドアウト式の拡張パネルを有効にすることができ、それによりインデックスで選択した特定のコンテナ(ファイル、名前空間またはクラス)に関する情報を表示させることができます。

Members タブには現在のコンテナにあるすべてのメンバのリストがあります。メンバのリストを、特定の型およびスコープを持つシンボルに絞り込むことができます。状況に応じて、リストの拡張が可能です。例えば File Index の場合、実際にファイルに定義されているシンボルのほかに、ファイル内で宣言されているシンボルを含めることができます。またクラスの場合は、現在のクラスに定義されたクライアントメンバのほかに、基底クラスから継承したメンバを含めることができます。

Metrics タブには、カレントコンテナのソフトウェアメトリックスが表示されます。これらのメトリックスは、コンテナの大きさ、設計及び複雑度の概要を提供します。 設定した閾値の範囲を逸脱したメトリックスの値を容易に見つけることができます。

Relationships タブは、カレントコンテナとデータベース内の他のコンテナとの関係を表示します。 たとえばクラスを調べているときには、どのクラスがそのクラスを継承しているか、ほかのどのクラスがそのクラスの関数を呼び出す関数を持っているか、といったことを知りたくなることがよくあります。Relationships コンボボックスで関係を選択すれば、どのクラスレベルの関係を表示するかを制御することができます。

File Index については、現在調べられているファイルに対するソースチェック違反がすべて、Source Checks タブにリストされます。

Symbol パネル

Symbol パネルは 1 回につき 1 つの特定のシンボルに関する概要を提供します。パネルは一連のタブから成り、そこにそのシンボルの補足的側面に関するさまざまな詳細情報が示されます。これらのタブはパネルのアイコンバーと共に、ツール全体のナビゲーションセンターとしての役割も持ちます。

General Information タブには、シンボルの型や範囲など、シンボルの一般的な属性が表示されます。

シンボルの定義は Source Code タブに表示されます。このソースファイルからの断片は、File Editor に表示されているときと同様に色づけされており、クリック可能ですが、シンボルの定義のみが含まれ、ファイルからのその他の情報は含まれません。

File Location タブは、ファイルシステム内におけるシンボルの場所を示します。クラス、関数、マクロなどのプログラム要素であるシンボルは、シンボルが定義または宣言されたソースファイルにあります。

Imagix 4D では、ソースコードに関する一連のソフトウェアメトリックスが生成されます。Metrics タブには、カレントシンボルのソフトウェアメトリックスの値が表示されます。Main パネルで Metrics レポートを使用すると、複数のシンボルのメトリックスが一度に表示されます。

Cross Reference タブにあるグラフは、シンボルの直接の依存関係を示しています。Main パネルから Graph を使用すると、ソフトウェアのより詳細なグラフィック表示が見られます。

Usage タブには、シンボルのすべての使用箇所のソースコードが表示されます。Usage タブは、シンボルが実際に使用される場所に関するデータベースの知識を用いて、にせの文字列一致を排除し、シンボルの有効範囲を理解し、マクロ定義の背後に隠れたシンボルの使用を追跡します。 Usage タブは File Editor の Next Reference、Previous Reference の機能を補完します。 Reference の機能では、コンテキストに沿って、特定のシンボルが使用されているところをひとつひとつめぐって調べることができますが、Usage タブはすべての使用箇所を一度にディスプレイに表示します。

Members タブには、調べている特定のシンボルのメンバがリストされます。ファイル、名前空間、およびクラスについては、同じ情報が Project パネルのコンテナインデックスリストへのスライドアウト拡張にある Members タブからも利用可能です。Project パネルからアクセスすると、Members タブには、リスト表示するメンバをフィルタリングするためのコントロールが組み込まれます。

最近の表示履歴は History タブに提供されるため、最近調べたシンボルを再度呼び出し、それに戻ることができます。タブ自体のコントロールのほか、マウスの右クリックによるコンテキストセンシティブメニューも、表示履歴のフィルタリングに役立ちます。

Information オーバレイ

Information オーバレイは Imagix 4D のすべてのディスプレイウィンドウで利用可能です。Shift キーと Ctrl キーを押しながらマウスの左ボタンを押すと、押している間だけ、マウスポインタで選択されたシンボルや関係に関する情報がオーバレイで表示されます。

シンボルについては、その型、有効範囲、ファイル位置、それに多くの場合、関連するメトリックスも表示されます。ただし、これは変わることがあります。たとえば、Control Flow モードのグラフ、または Flow Chart ウィンドウでは、関連するソースコードの行も表示されるでしょう。また、File Editor では、オーバレイにファイルそのものに関する情報も表示されます。

関係が選択されている場合には、オーバレイは関連するふたつのシンボル、それらの関係の型、それに可能なら、その関係を生じるソースコードの行も表示します。Information オーバレイは、Graph ウィンドウに 3D で表示された関係には有効ではありません。

ドキュメントの作成

Imagix 4D は、複雑であるか、サードパーティ製であるか、あるいは旧式である C、C++、Java のソフトウェアの解析、ドキュメント作成、および改善に役立ちます。このうち、ドキュメント作成の面では、あとでほかの人に役立ててもらうためのドキュメントを作成するという、ソフトウェア開発者たちにとっては、最も気乗りがせず、報われることも少ない作業を処理します。

Document 機能は、変更の影響や問題の解析から、コードのレビュー、システム設計に関するドキュメントの作成に伴う面倒な作業を大幅にカットし、RTF や HTML などの多彩な形式でドキュメントを自動生成することにより、正確で最新の情報を提供します。

Print 機能は、特定のディスプレイウィンドウの内容を、対話型の操作を通して捕捉し、通信する作業を単純化します。

Document 機能

Imagix 4D のドキュメント生成機能によって、ツールのデータベースの情報から詳細なドキュメントを自動作成することができます。ドキュメントの範囲と形式を制御することによって、より多くのシナリオに使用可能なドキュメントを作成できます。その一般的な用途を以下に示します。

包括的な設計文書。自分で作成したコードまたは他人から受け継いだコードについて、ソフトウェアの特定の側面の参照と調査に最も良い方法は、通常は Imagix 4D をユーザインターフェイスを通して使用することです。しかし、この方法は予算の制約、技術的な制約、または静的な設計成果物の必要性などにより適切でない場合があります。そのような場合、プロジェクトを完全にカバーするドキュメントによって、組み込みのナビゲーションと一部の基本的な解析機能を含む総合的な設計情報が提供されます。

コードレビュー用ドキュメント。Imagix 4D により生成されたドキュメントを、体系的なコードレビューのためのプラットフォームとして使用することができます。ファイル、クラスまたはシンボルのレベルで、レビュー対象のコード単位について包括的な詳細と解析の情報を含むドキュメントを自動的に作成できます。コンテンツとレイアウトを制御することにより、ドキュメントを標準のレビュープロセスに容易に適合させることができます。

変更の影響または問題の解析用ドキュメント。Imagix 4D を使用してソフトウェアの制御フローや依存関係における問題を調査するとき、調査結果を伝達や参照の目的で記録したい場合があるでしょう。このような操作も、Imagix 4D のユーザインターフェイスの中で、ブックマーク機能を使用して実現できます。しかし、外部ドキュメントを入手したい場合には、ドキュメント機能を通して、包括的な記録を迅速かつ自動的に生成することができます。

ドキュメントの範囲

Document ダイアログでは、ソフトウェアのうちのドキュメント化したい部分、及び、そのドキュメントに含めたい情報の種類を指定します。Imagix 4D はそのデータベースのデータとソースファイルに存在するデータを組み合わせ、自動的にドキュメントを作成します。

ドキュメントの範囲は、一部、ドキュメント機能を呼び出した場所によって制御されます。Document ダイアログは、ユーザインターフェイスのいくつかの場所から開くことができます。呼び出された場所によって、異なる範囲が選択肢として与えられます。

プロジェクト全体を記録するには、メインのメニューバーから Document ダイアログを開きます。例えばコードレビュー用ドキュメントなど、特定のファイルまたはクラスに関するドキュメントを作成するには、ユーザインターフェイスの任意の場所でそのファイルまたはクラスを右クリックした後、表示されるコンテキストセンシティブメニューから Document...アクションを選択します。Graph ウィンドウで作業しているとき、ドキュメント化したい関数依存関係の呼び出しグラフが作成されていれば、その Graph ウィンドウのローカルメニューバーを通してダイアログを起動します。

ドキュメントのフォーマット

Document 機能は、異なる用途に対応するために ASCII、RTF、および HTML の 3 つの形式で出力を生成します。実際のところ、どのドキュメントも 3 つの形式のどれかで作成可能ですが、各形式には適した利用方法があります。多くの場合、ドキュメントの用途に応じて、選択すべき形式が決まります。

ASCII 書式では簡単で読み易いファイルが生成されます。ASCIIドキュメントの生成と後処理は、Imagix 4D から情報をエクスポートするための迅速な方法の 1 つです。本ツールは大規模なデータベースと強力な解析およびエクスポートの機能を備えているので、より高度なデータエクスポートやカスタムレポートを必要とする場合には、Imagix 社またはお客様の地域の代理店にご連絡ください。

RTF(リッチテキストフォーマット)では、ハードコピーのドキュメントを生成することができます。Imagix 4D は、ワードプロセッサにインポートすることができる(一連の)ファイルを生成します。大きいドキュメント、ワードプロセッサを使用すれば、タイプスタイルとページレイアウトを調整することができます。ワードプロセッサで目次を作成する場合は、項目を Section、Subsection の形式にします。しかし、RTF は通常、コードレビューや問題解析を目的とする小規模なドキュメントに使用すべき形式です。そのようなドキュメントには、多くの場合目次は不要です。ワードプロセッサを用いて、記述したい任意の注釈を迅速に追加した後に、作業中のタスクのために共有できる書類や電子文書を生成することができます。

HTML フォーマットでは、インデックスを含むオンラインドキュメントの作成が可能です。グラフ内に表示されるシンボルも含めて、すべてのシンボル参照がリンクされます。メトリックス表とクラス階層も含まれます。包括的な設計文書には、通常はこの形式を選択するべきです。

ドキュメントの内容とレイアウト

プロジェクトの範囲内で、ドキュメント出力の内容を幅広く調整することができます。含まれるシンボル型、及び個々の含まれるシンボルについて生成される情報を指定可能です。Imagix 4D の利用を開始するにあたって、これを制御する最も簡単な方法は Document Settings ダイアログを使用するものです。このダイアログには、タブで区切られた一連のチェックボックスがあり、どのメトリックスを含めるべきか、どのコメント形式を探すべきかなどを指定することが可能です。

特定のシンボルタイプに対して利用可能な情報の定義と、その情報の表示順序については、../imagix/user/user_doc.txt テンプレートによって制御されます。上級ユーザが、このファイルを見直して修正し、用途に合わせて出力を最適化したい場合もあるでしょう。それを行う手順はファイル自体に記述されています。変更を行う前に、元のファイルのコピーを必ず作成してください。

Document Settings ダイアログおよび user_doc.txt テンプレートの代わりに、DocGen シートを使用することもできます。DocGen シートを用いると、ダイアログおよびテンプレートの場合と同じ内容を指定したのち、それらの設定をセッション間で保存することができます。さらに DocGen シートは、ドキュメントとそのビルド処理に関するはるかに多くの側面の制御を可能にします。それらの側面には、ページ幅やグラフの大きさなどの外観の要素から、ドキュメント生成の分散などのビルド上の問題までが含まれます。

../imagix/user/doc_gen のディレクトリにある.dgn ファイルはすべて Document ダイアログを通して適用することができます。DocGen シートで行うことができるあらゆる設定に関する情報は、サンプル(../imagix/user/doc_gen/sample.dg_)におさめられています。

一連の DocGen シートを構成することによって、容易に適用可能なドキュメント形式の定義を、それぞれ特定のドキュメントタイプ生成のために最適化しながら作成することができます。

Print 機能

Print 機能は最も迅速かつ容易にハードコピー出力を生成する方式、または特定のウィンドウの内容をひとつのファイルに捕捉する方式を提供します。現在表示されたディスプレイウィンドウ、及びすべてのレポートはプリント出力することができます。

たとえば Graph ウィンドウや Flow Chart などのグラフィック表示をプリント出力することを選択すれば、Print ダイアログに表示される設定項目を選択し、選択しているページサイズには大きすぎるグラフィックスを縮小するかしないかを指定することができます。グラフィックス出力のその他の設定、とりわけサイズ及び解像度に関する設定は、Print ダイアログの範囲外ですが、デフォルト設定を変更することによって変更することができます。詳しくは../imagix/user/sample.4Ddefaults を参照してください。

非グラフィックディスプレイは、通常 ASCII ファイルとして出力されます。Metrics ウィンドウおよびいくつかのレポートについては、替わりにカンマ区切り(.csv)のフォーマットを使用してスプレッドシートにエクスポートされます。また、Windows システムのプリンタに直接出力する場合は、PCL が使用されます。Imagix 4D のテキストに対する、PCL のフォーマットの限界から、非グラフィック表示はファイルにプリント出力した上で、Wordpad またはその他のアプリケーションを用いてさらにフォーマットし、プリント出力したほうがよいでしょう。

グラフィックウィンドウからプリンタへ直接プリント出力する場合には、フォーマットは自動的に、Windows システムなら PCL、Unix システムなら PostScript に設定されます。ファイルへプリント出力する場合には、いくつかの選択肢のなかからフォーマットを選択することができます。

PNG	Portable Network Graphics(.png)フォーマットは、web ブラウザでのビュー、ワードプロセッサへのインポートなど、数多くのアプリケーションについて GIF の後継フォーマットです。PNG フォーマットは、GIF ファイルに伴う特許問題も回避します。PNG フォーマットではカラー出力またはグレースケール出力の選択が可能です。
PostScript (.ps)	Windows の一部、及び Unix の大半のプリンタは、PostScript フォーマットをサポートします。さらに、Adobe Acrobat、Microsoft Word など、数多くのアプリケーションも、PostScript(.ps)または Encapsulated PostScript(.eps)ファイルをインポートすることができます。PostScript フォーマットではカラー出力またはグレースケール出力の選択が可能です。
Visio .vdx	Visio Drawing XML (.vdx)フォーマットは、Imagix 4D グラフィック表示のさらなる編集を目的とした、Visio2003 以降のバージョンの Microsoft Visio へのインポートをサポートします。
Visio .csv	Comma Separated Values(.csv)フォーマットは、Imagix 4D グラフィック表示のさらなる編集を目的とした、Visio2003 より前のバージョンの Microsoft Visio へのインポートをサポートします。 Imagix 4D の.csv ファイルを Visio にインポートする手順は、次項に説明します。
Table	Table フォーマットは、Graph ウィンドウに表示されたシンボル及び関係情報を反映する ASCII テキストファイルの生成を可能にします。

グラフィック表示からプリント出力への色変換は、ファイル ../imagix/user/user_prt.tcl を通してカスタマイズすることができます。ハードコピーの読み取りを容易にするため、Graph ウィンドウはつねに 2D グラフとしてプリント出力します。

csv ファイルの Visio へのインポート

Visio 2003 より前のバージョンの Microsoft Visio を使用している場合、Imagix 4D のグラフィック表示は、Visio .csv フォーマットを選択してファイルへプリント出力することにより、さらなる編集のために Visio にインポートすることができます。

Visio のインストレーションプログラムを Imagix 4D で生成された .csv ファイルをインポートすることができるようにセットアップするには、まずステンシルファイル../imagix/user/doc_gen/imagix_shapes.vss をディレクトリ..\Visio\Solutions にコピーしなければなりません。これにより、Visio が.csv ファイルで使用されているさまざまな形状を見分けられるようになります。

imagix_shapes.vss ファイルのコピーが終われば、Imagix 4D のグラフを Visio にインポートすることができます。まず、Visio.csv フォーマットを選択した上で、グラフをファイルへプリント出力することによって Imagix 4D からエクスポートします。ファイル名には必ず、拡張子.csv を使用してください。

次に、その.csv ファイルを Visio にインポートします。Visio 側で.csv ファイルを開きます(Visio メニュー[ファイル][開く])。ファイルの種類フィールドがテキストファイル(*.txt、*.csv)に設定されていることを確認してください。ファイルを選択し、開くボタンをクリックすると、Visio ファイルコンバータダイアログが表示されます。セパレータの設定はデフォルト値(、" ;)のままにして、OK をクリックします。プログレスバーにインポートステータスが表示され、やがて Imagix 4D のグラフが Visio に表示されます。

システム管理上の注意点

ライセンスの管理

Imagix のライセンスマネージャをみなさんの環境のなかで動作するように設定する方法は 3 つあります。

ノードロックライセンス Windows 環境下で使用可能なノードロックインストレーションは Imagix 4D を特定のマシン上で操作することを可能にします。この方法にはノードロック型の Windows ライセンスファイルが必要です。

ファイルベースのフローティングライセンス Unix/Linux 環境下で使用可能なファイルベースのフローティングインストレーションはユーザにライセンスの共用を可能にします。ひとりのユーザが Imagix 4D のセッションを閉じると、空がひとつでき、別のユーザが Imagix 4D を開くことができます。ライセンスはファイルベースで管理されるので、ライセンスサーバがバックグラウンドで動作している必要はありません。この方法は Unix/Linux ライセンスファイルを必要とし、Imagix 4D の動作をひとつの Unix/Linux クライアントアプリケーションとしてサポートします。

サーバベースのフローティングライセンス Unix、Linux、及び Windows にまたがって使用可能なサーバベースのフローティングインストレーションはユーザにライセンスの共用を可能にします。ひとりのユーザが Imagix 4D のセッションを閉じると、空がひとつでき、別のユーザが Imagix 4D を開くことができます。ライセンスはサーバベースで管理されるので、Imagix 4D のセッションを開始できる状態になる前にライセンスサーバがバックグラウンドで動作している必要があります。ライセンスサーバを実行する場所に応じて、この方法は Unix/Linux のライセンスファイルまたはフローティングの Windows ライセンスファイルを必要とします。ライセンスサーバ自体が実行されるオペレーティングシステムにかかわりなく、Imagix 4D クライアントアプリケーションは Unix、Linux、及び Windows でサポートされています。

ライセンスファイルの内容

ライセンスファイルを調べれば、それがどのライセンスの方法をサポートし、どのワークステーションにつながっているかを確かめることができます。フローティングライセンスであれば、何人のユーザが同時並行してそれを実行することができるかも分かります。

こうした情報はすべてライセンスファイルに含まれています。このファイルの最初の 2 行は次のようなフォーマットになっています。

```
Installation company_name machine_id any
License 4D rel_num.0 [num_seats] [mac_addr] l\}vy...
```

company_name は、そのライセンスが発行されている会社を識別します。

machine_id は、そのライセンスが発行されている特定のマシンを識別します。ファイルベースのフローティングライセンスの場合、これは Imagix 4D のファイルサーバの役目を果たすマシンです。サーバベースのフローティングライセンスの場合、これはライセンスサーバが動作するマシンです。Unix では、machine_id はコマンド hostid（HP-UX では uname -i）を実行することによってわかるマシンのホスト ID です。Linux では、これは/sbin/ifconfig -a の実行によって検索される MAC アドレス（Hwaddr）です。Windows では、これはコントロールパネルのシステムアプレットのコンピュータ名タブに表示されるフルコンピュータ名です。

rel_num は、そのライセンスがサポートするリリース番号を示します。値が 4 であれば、そのライセンスが 4.X.X に該当するすべてのリリースの Imagix 4D に適用されることを示します。

num_seats は、そのライセンスがサポートする同時並行ユーザの数を示します。この整数値（Windows ライセンスでは最後に S が付きます）は、ノードロックライセンスでは表示されません。

Windows ライセンスだけに適用される *mac_addr* は、そのライセンスが発行されている特定のマシンを識別します。デフォルトでは、これは ipconfig /all の実行によって検索される、最初の Ethernet カードの MAC アドレス（物理アドレス）です。MAC アドレスがない場合は、稼動中の Windows のシリアル番号が代わりに使用されます。これは、コントロールパネルのシステムアプレットにある全般タブの使用者のセクションに表示される、英数字の文字列です（12345OEM67890123 など）。

ライセンスのインストール

詳細なライセンスインストールの手順書は、Imagix 4D の配布ソフトウェアに同梱されています。Imagix 社の Web サイトからダウンロードする場合、インストール手順書へのリンクは Imagix 4D ソフトウェア自体へのリンクと同じ Web ページ上にあります。配布される CD-ROM では、手順書は CD-ROM のルートディレクトリにあります。Unix/Linux の tar ファイルを展開するか、自己インストールの Windows 実行可能ファイルを実行すると、手順書のコピーが imagix のルートディレクトリに置かれます。以下に、ライセンスインストール処理の概要を示しますが、詳細についてはインストール手順書を参照してください。

Imagix 社またはその販売代理店からライセンスファイルを入手したら、それをみなさんのファイルシステムのパーマネントにコピーしてください。場所は Imagix 4D をインストールしようとしているマシンから可視のところを選んでください。

Imagix License Installer は Imagix 4D のメニュー[Help] > [License] > [Install]を通して起動します。現在有効なライセンスをインストールしていなければ、License Installer は Imagix 4D を起動しようとしたときに自動的に起動します。

ノードロックライセンス

通常、Windows 用に提供される評価ライセンスはノードロックライセンスです。ノードロックライセンスは管理者にとって比較的わかりやすい方法です。ライセンスが発行されたマシン上で、License Installer を起動します。License Installer のダイアログを通してインストールを進め、必要に応じて、コピーしたライセンスファイルの完全パス名を入力します。License Installer にライセンスのインストールが完了したことが表示されたら、Imagix 4D を再起動するだけで、このツールの使用を始められます。

ファイルベースのフローティングライセンス

ファイルベースのフローティングライセンスも管理者にとって比較的わかりやすい方法です。ライセンスが発行されている同一マシン上で、Unix のすべてのプラットフォームの中から適切な Imagix ソフトウェアをインストールします。そのマシンにログインしてください。これは、ログインした結果、操作しているモニタに Imagix License Installer の X ウィンドウが表示される状態にある限り、リモートログインで行えます。ディレクトリ../imagix/data に対して書き込みパーミッションを持っている必要があります。License Installer のダイアログを通してインストールを進め、必要に応じて、コピーしたライセンスファイルの完全パス名を入力します。

License Installer にライセンスのインストールが完了したことが表示されたら、Imagix 4D を再起動するだけで、このツールの使用を始められます。どの Unix マシン上でも、a)そのマシンが Imagix ソフトウェアのインストールされているファイル位置に対して読み取りも書き込みも可能で、b)インストールされた Imagix ソフトウェアをある種の Unix 上で動作させている場合、c)ライセンスの同時並行ユーザの空があるかぎり、Imagix 4D を使用することができます。

ファイルベースのフローティングライセンスでは、ユーザが正規の手続きを経て Imagix 4D を終了すると、同時並行ユーザの空ができます。segfault などが発生し、ユーザが正規の手続きを経ずに終了した場合には、空はできず、そのユーザはまだログインしているものと見なされます。空をつくるには、そのユーザにそのとき使用していたマシン上で Imagix 4D を起動し、正規の手続きを経て終了してもらいます。

サーバベースのフローティングライセンス

サーバベースのフローティングライセンスをインストールするのはやや複雑です。実際のライセンスをインストールする時点で、通信チャネルを指定する必要があります。また、Imagix ライセンスサーバを動作させているマシンが再起動するときにライセンスサーバが自動的に再起動されるようにシステムを設定する必要もあります。

Imagix ライセンスサーバは tcp/ip ソケットを介して実際の Imagix 4D クライアントアプリケーションと通信します。License Installer では、ライセンスサーバが動作するマシンの IP アドレス（またはそれに相当するマシン名）を指定する必要があります。また、通信用ポート、またはソケットも指定する必要があります。これは 1～9999 の整数で指定し、通常は 8000～9999 の範囲の値を使用することになっています。システム管理者に使用可能なポートを確認してください。

Imagix 4D のクライアントアプリケーションが Imagix ライセンスサーバとは別のコピーの Imagix ソフトウェアから実行される場合（すなわち、1 台またはそれ以上の Imagix ファイルサーバが Imagix ライセンスサーバとは別のマシンの場合）には、サーバベースでライセンスされているクライアントマシンの部分もインストールする必要があります。これも Imagix License Installer を通して行います。そのさいには、ライセンスサーバ用と同じ IP アドレス、及び同じポート、またはソケットを指定します。これは、ライセンスサーバがない個々のファイルサーバについてそれぞれ行う必要があります。

たとえば、Unix マシン 1 台（PaloAlto1）と Windows マシン 1 台（Redmond1）を Imagix 4D ファイルサーバとして動作させているとしましょう。この場合、第 2 の Unix マシン（PaloAlto2）のユーザが Imagix 4D を起動するときには、PaloAlto2 が PaloAlto1 から Imagix クライアントアプリケーションソフトウェアを読み込み、実行します。Windows 側でも同じです。Redmond2 が Redmond1 の Imagix ソフトウェアを読み込み、実行します。ライセンスサーバが PaloAlto1 だとすると、PaloAlto1 上で Imagix License Installer を実行し、server-based floating license - server machine としてのインストールを行う必要があります。また、Redmond1 上でも Imagix License Installer を実行し、server-based floating license - client machine としてのインストールを行う必要があります。

また、Windows マシンばかりでネットワークを構成しているが、ライセンスサーバを走らせるのは 1 台で、7 台のユーザマシンのそれぞれに Imagix 4D ソフトウェアをローカルにインストールしている場合も考えてみましょう。この場合には、Redmond1 に server-based floating license - server machine としてのインストールを行い、Redmond2～8 の各マシンには server-based floating license - client machine としてのインストールを行います。

ライセンスサーバを起動する

ノードロックライセンス、ファイルベースのフローティングライセンスでは、Imagix 4D はアプリケーションを起動するだけで使用できます。ただし、サーバベースのフローティングライセンスの場合は、Imagix 4D の空を確認する前に Imagix ライセンスサーバを稼動している必要があります。

Unix/Linux システムの場合、ライセンスサーバは、../imagix/bin/imagix-licsrv を起動することによって起動されます。また、Imagix ライセンスサーバを動作させているマシンが再起動するときにライセンスサーバが自動的に再起動されるようにシステムを設定する必要があります。Unix/Linux システムではこれは一般に、Imagix ライセンスサーバの呼び出しを初期化スクリプト（/etc/rc?）に追加することを意味します。Unix/Linux 版の製品には、../imagix/bin ディレクトリにサンプル rc スクリプトが含まれています。これは環境に合わせて修正することができます。

Windows 2000/XP/Vista/Win7 システムでは、インストールのプロセスで自動的にライセンスサーバがマシンのサービスのひとつとしてセットアップされますが、初めてライセンスをインストールするときには、Services アプレットを通して手動でライセンスサーバを起動する必要があります。これらの環境については、それぞれのシステムドキュメントでより詳しい情報を参照してください。

いくつかの設定オプションが用意されています。チェックアウト及びチェックインされたクライアントアプリケーションのログを生成するには、-licsrvlog FILE オプションを使用してライセンスサーバを起動します。FILE は、ライセンスのアクティビティが記録されるログファイルのフルパス名です。Windows では別の方法として、regedit を使用して文字列 LogInfo を HKEY_LOCAL_MACHINE\Software\Imagix\4D レジストリに追加することにより、ログファイルを指定することも可能です。次に LogInfo の値を FILE に設定します。

現在のライセンスの状態は、コマンドラインからの問い合わせにより取得することができます。Windows ライセンスサーバについて問い合わせるには、ライセンスサーバマシン上で `imagix-licsppt -licstatus' を実行します。Unix/Linux では、`imagix-licstatus' を起動します。

Imagix 4D クライアントアプリケーションを実行できるマシンを制限することもできます。通常は、ライセンスサーバにアクセスできるマシンは、空を利用できる限りクライアントアプリケーションからチェックアウトできます。ただし、ファイル../imagix/data/clients がある場合はファイルが読み込まれ、クライアントファイルにホスト名が表示されるマシンだけが、Imagix 4D クライアントアプリケーションからチェックアウトできます。このファイルでは、句読文字ではなくスペースでマシン名を区切ります。

フリーフローティングライセンス

サーバベースのフローティングライセンスのもとに Windows ラップトップ上で Imagix 4D を操作している場合には、ネットワークとの接続は切断しながら、そのラップトップ上での Imagix 4D の操作は続けたいと思うことがあるでしょう。Imagix License Server は、メニュー［Help］> ［License］を通して使用可能な Free-Float License 機能でそうした使用形態もサポートします。この機能を利用すれば、その Windows ラップトップへのライセンスを確認することもできます。ユーザとしての使用権限はそのラップトップによって 1 週間まで保持され、ライセンスサーバとの接続が切れているときにも Imagix 4D を起動することができます。あらためてネットワークに接続し、ライセンスが切れていた場合にも、上記のメカニズムを通してもう一度チェックインすればよいだけです。

ライセンスサーバを除去する

Windows 2000/XP/Vista/Win7 システムにおいて、Imagix ライセンスサーバを除去するための最初の手順は、ライセンスサーバマシン上の Services アプレットによりライセンスサーバを停止することです。次に、同じマシン上の..\imagix\bin ディレクトリから`imagix-licsppt -remove'を実行して、サービスのリストから Imagix ライセンスサーバを取り除きます。この手順が完了すれば、実際の Imagix ライセンスサーバとアプリケーション・ソフトウェアを当該マシンのインストール場所から除去することができます。

Unix/Linux システムでは、ライセンスサーバを停止して、初期化スクリプト(/etc/rc?)に加えた変更を元に戻す必要があります。

Imagix 4D のカスタマイズ

メニュー項目、オペレーティングシステム間でのシステムディレクトリの命名方法、最適化されたレポートに至るまで、Imagix 4D のいくつかの側面はカスタマイズが可能です。そのうち多くは、../imagix/user ディレクトリの下にあるサブディレクトリやファイルを変更することにより実現されます。ファイルとその変更方法に関する詳しい説明はファイル自体に含まれています。ここでは、実施可能ないくつかのカスタマイズについて簡単な説明を記述します。

ツールの動作と外観に関係する特定の設定は、Imagix 4D のダイアログで指定することができます。主なダイアログ設定および最近開かれた 50 個のプロジェクトのリストは、ユーザごとに次のセッションまで保存されます。Unix では、この情報はファイル~/.4Drc6 に格納され、Windows では、レジストリ HKEY_CURRENT_USER::Software::Imagix::4D 4Drc6 が使用されます。

その他の設定については、独自のデフォルト値を指定し、Imagix 4D 起動時にそれをロードすることができます。これは、4Ddefaults ファイルを通して行われます。4Ddefaults ファイルはサイト用のファイルとユーザ固有のファイルの両方を存在させることができます。そのような場合には、両方のファイルが読み取られ、個々のユーザ固有の設定がサイト用の設定に優先して使用されます。

ファイル../imagix/user/sample.4Ddefaults には、一部の共通デフォルトについての詳細とサンプル設定も含まれています。その他のデフォルトを変更する場合は、詳細についてテクニカルサポートにお問い合わせください。

デフォルト設定に加えて、Imagix 4D にはその他にもカスタマイズ可能な部分があります。ディレクトリ../imagix/user には、Imagix 4D の操作に関する特定の部分を制御する、いくつかのファイルがあります。

user_doc.txt	Imagix 4D の自動生成ドキュメントについて、デフォルトのコンテンツと組織をカスタマイズします。user_ed.tcl 非 Imagix 4D エディタを開きます。また、Imagix 4D の File Editor を設定管理システムと連携させます。
user_mnu.tcl	Imagix 4D のメニューをカスタマイズします。
user_prt.tcl	グラフィックウィンドウをプリント出力するときの色変換をカスタマイズします。
user_rpt.tcl	追加のレポートを生成し、それを［Reports］メニューに追加します。

これらのファイルには、最低限の使用説明が含まれています。ファイルを修正する場合には、tcl/tk 言語に関する基本的な知識と、ある種の思い切りが必要です。

環境への適合

Imagix 4D は集中的な計算よりも多くのメモリを使用します。このため、大規模なプロジェクトを扱っていて、ツールのパフォーマンスを向上させたくなったときには、プロセッサの処理速度を上げるより、メモリを追加するほうがよいでしょう。さもなければ、気がついたときにはワークステーションがメモリのスワッピングに多くの時間をとられているでしょう。

Imagix 4D を使っていると、2 バイトを割り当てようとしていてメモリが不足したというメッセージが表示されることがあります。これは通常、システムがスワップスペースを使い切ったことを示しています。この問題を解決するには、ディスクスペースから追加のスワップスペースを生成します。Imagix 4D が単一のプロジェクト内で動作しているときには、使用するスワップスペースがしだいに増加します。このメモリを解放し、パフォーマンスを改善するためには、作業中のプロジェクトを時々開き直したほうがよいでしょう。

同じメモリ不足のメッセージはヒープスペースが不足したときにも出ることがあります。スワップスペースを充分にとっているのに、まだメッセージが出るときには、使用可能なヒープスペースを確認してください。これは、デフォルトのヒープスペースの限度が 64MB の HP-UX システムでよく問題になります。大規模なプロジェクトを扱っているときには、少なくとも 500MB は使用可能にしたほうがよいでしょう（Solaris システムのデフォルト値は 2000MB です）。

問題のご報告

弊社は Imagix 4D が問題なく動作するよう最善を尽くします。それでも、これはきわめて複雑なツールであり、難しい問題が発生することがあります。このツールには、エラーログをとるメカニズムと、要求に応じてトレースファイルを生成するメカニズムが組み込まれています。

通常、Imagix 4D が動作しているときに内部でエラーが発生すると、カーソルが一時的にスプレー容器（虫除けスプレー）の形に変化し、現在の動作は停止し、エラーログが記録されます。その時点で、制御はユーザの手に戻ります。カーソルの変化に気づかなかった場合には、行った操作に対して何も変化していないことが唯一のエラー発生の合図となります。

Unix 環境では、Imagix 4D がエラーログを作成するために使用するデフォルトファイルは../imagix/user/.errorlog に置かれます。このファイルが書き込み可能でない場合には、代わりに ~/.4Derrorlog の場所が使用されます。ほとんどのバージョンの Windows ではファイル..\imagix\temp\error.log が使用されますが、Vista と Windows7 ではその場所が C:\Documents and Settings\(user name)\AppData\Imagix\error.log になります。ログには、単なるエラー発生時刻の記録のほかに、エラーの原因を排除する上で有用な情報も含まれています。みなさんにも、定期的にエラーログファイルの存在を確認し、それを support@imagix.com までお送りいただけると幸いです。

みなさんが Imagix 4D を使用していてぶつかった問題について弊社テクニカルサポートのスタッフにご協力いただいているときには、トレースログをお送りいただくことがあります。トレースログは、Options ダイアログ（メニュー[Tools] > [Options] > [Environment] > [Technical Support]）で生成されるようにすることができます。Unix では生成されたファイルは~/.4Dtrace_(hostname)_(timestamp)に書き込まれます。Windows ではファイルは trace_(username)_(hostname)_(timestamp).log という名前で error.log ファイルと同じディレクトリ（上記参照）に置かれます。Imagix 4D の再起動時、生成後 24 時間を超えるトレースファイルはすべて削除されます。トレース処理は Imagix 4D のパフォーマンスを低下させ、非常に大きいファイルが生成されることがあるため、問題が発生している場合にのみ使用することをお勧めします。クラッシュサポートモードでトレースを実行すると、さらにパフォーマンスが低下します。トレース処理で関連データだけを取得するように設定すれば、パフォーマンスとファイルサイズに関するこれらの影響を最小にすることができます。

付録

付録 A. Imagix 4D の起動

Imagix 4D は、..\imagix\bin\imagix.exe (Windows の場合) または ../imagix/bin/imagix (Unix/Linux の場合)を実行することにより起動されます。通常、Imagix 4D はオプションを付けずに起動されます。1 つ以上のオプションを付けて Imagix 4D を起動すると、その動作が変更されます。

-cmmd *filename* Imagix 4D をバッチモードで起動し、ファイル *filename* によって指定されたコマンドを実行します。

プロジェクトデータの再作成、またはレポートやドキュメントの作成など、重要で繰り返しの多いタスクについては、コマンドライン(バッチ)処理でツールを起動したいと考えるかもしれません。この機能に関しては、*filename* に記述するコマンドおよび書式も含め、付録 B で詳しく説明します。

-demo Imagix 4D を起動し、自動的にサンプルプロジェクトを開きます。

デモモードで起動すると、Imagix 4D は自動的にサンプルプロジェクトをロードします。ただし他のプロジェクトを開いたり、他のデータソースをロードすることはできません。デモモードでは一切ライセンスが使用されません。この動作は、Imagix 4D をデモライセンスキーを使用して起動するときも同様です。

-project *dirname* Imagix 4D を起動し、自動的に *dirname* というプロジェクトを開きます。

-project オプションを使用して起動すると、Imagix 4D は自動的に指定されたプロジェクトを開くほかは標準起動時と同様に動作します。Imagix 4D ではユーザインターフェイスを通して、他のプロジェクトを開いたり、データソースを追加したりすることができます。-project オプションは、マルチユーザ環境において、経験があまりないユーザが特定のプロジェクトを使用するようにする上で最も効果的です。

-regen *dirname* Imagix 4D を起動し、自動的に *dirname* というプロジェクトを開き、ただちにプロジェクトデータを再作成します。

-regen オプションを使用して Imagix 4D を起動すると -project オプションを使用したときと同様に動作しますが、異なる点は、プロジェクトがソースコードの現在のビューを示すようにプロジェクトデータがただちに再作成されることです。

付録 B. バッチモードコマンド

Imagix 4D はコマンドラインから動作させることができます。これは、バッチ操作として決まったスケジュールで実行したいプロジェクトの再生、ドキュメントの作成などの主要な、繰り返し行う作業には、非常に便利です。

Imagix 4D は、imagix -cmmd *cmmd_file_name* で呼び出すことにより、バッチプロセスとして実行することができます。このモードでは、Imagix 4D は起動し、コマンドファイル *cmmd_file_name* に指定されたコマンドを実行した上で、終了します。

より詳細なドキュメントが、../imagix/user/user_cmd.txt ファイルにあります。

コマンド

コマンドファイルの一部として使用したいと思われる手続きはいくつかあります。

cmmdOpenProject	プロジェクトを作業状態にする
cmmdUpdateProjectData	プロジェクトを更新する
cmmdRegenerateProjectData	プロジェクトを再生する
cmmdReport	レポートを出力する
cmmdReportExport	今後のインポート用にレポートを出力する
cmmdReportMultiple	複数のレポートを出力する
cmmdReportExportMultiple	今後のインポート用に複数のレポートを出力する
cmmdMetrics	メトリックスを出力する
cmmdCalcMetrics	計算を行うだけでメトリックスを出力しない
cmmdDocument	ドキュメントを出力する

cmmdDebug　デバッグトレースを作成する **cmmdOpenProject *project_name***

プロジェクト project_name を作業状態にする手続きです。project_name はすでに存在しているプロジェクトでなければなりません。これは Open Project 機能（メニュー［File］>［Open Project］）に相当します。

cmmdUpdateProjectData

現在作業状態になっているプロジェクトを更新する手続きです。これは Update Project Data 機能（メニュー［Project］>［Update Project Data］）に相当します。

cmmdRegenerateProjectData

現在作業状態になっているプロジェクトを再生する手続きです。これは Regenerate Project Data 機能（メニュー［Project］>［Regenerate Project Data］）に相当します。

cmmdReport *report_name* [*file_name*]

cmmdReportExport *report_name* [*file_name*]

レポート *report_name* をファイル *file_name* に出力する手続きです。cmmdReport の代わりに cmmdReportExport を使用すると、ファイルを後で Imagix 4D にインポートして戻し、参照とナビゲーションのためにシンボル名のすべてが完全にタグ付けされた状態で表示されるように、結果のファイルに特殊なデータ部分が組み込まれます。

report_name はメニュー［Reports］で表示されるレポートの名前からとります。スペースはアンダーバー（_）に変え、フォーマットはピリオドふたつ（..）で分ける必要があります。たとえば、メニュー［Reports］>

［File Summary］で表示されるレポートを出力する場合には、File_Summaryと指定し、メニュー［Reports］>［Source Checks］>［Expressions］>［Conversion Issues］>で表示されるレポートを出力する場合には、Source_Checks..Conversion_Issuesと指定します。

file_name はファイルの完全パス名です。file_nameを省略すると、cmmdReportは、前回 Imagix 4D がインタラクティブ（ノーマル）モードで起動されたときの Printダイアログに指定されたファイルまたはパイプにレポートを出力します。この場合、cmmdReportを実行する前に、プリント出力を制御する変数をコマンドファイルに設定したいと考えるかもしれません。これらの変数は Printダイアログの設定に対応します。これらの変数の設定により、前回のインタラクティブセッションでの設定が上書きされます。

変数	設定
vtgPrint(Comment)	*string*
vtgPrint(Output)	{ ToFile | PipeTo }
vtgPrint(PipeTo)	*string*
vtgPrint(FileName)	*file_name*

Windowsの場合、vtgPrint(出力)を PipeToに設定することは、Printダイアログで'To Default Printer'を選択するのに相当します。

レポートの多くには、いくつかのオプション設定が用意されています。デフォルトでは、cmmdReportは Imagix 4D が前回対話(通常)モードで実行されたときの設定を使用します。これらの設定は、../imagix/user/user_cmd.txt ファイルにリストされた変数を使用して、コマンドファイルで明示的に制御することができます。

cmmdReportMultiple [*directory_name* [*report_list* [*skip_list*]]]

cmmdReportExportMultiple [*directory_name* [*report_list* [*skip_list*]]]

これらの手続きは、個々の指定されたレポートがディレクトリ *directory_name* 内のファイルに出力されるようにします。cmmdReportMultipleの代わりに cmmdReportExportMultipleを使用すると、ファイルを後で Imagix 4D にインポートして戻し、参照とナビゲーションのためにシンボル名のすべてが完全にタグ付けされた状態で表示されるように、結果のファイルに特殊なデータ部分が組み込まれます。

出力ファイルの名前は自動的に付けられます。*directory_name* が省略されると、レポートはカレントプロジェクト(cmmdOpenProjectを通して開かれたプロジェクト)のあるディレクトリ内の、reportsサブディレクトリに出力されます。

デフォルトでは、cmmdReportMultiple は Reports メニューの下に現れるすべてのレポートを実行します。*report_list* および *skip_list* によって、実行するレポートを制限することができます。各レポートには、cmmdReport での指定と同様に、report_name の形式を使用してください。各リストは、{File_Summary Source_Checks..Conversion_Issues} のように波括弧で囲みます。skip_listで指定されたもの以外のすべてのレポートを作成したい場合は、report_list 引数に{}を使用するか allを指定します。

cmmdMetrics [*directory_name* [*symbol_type_list*]]

cmmdCalcMetrics [*symbol_type_list*]

cmmdMetricsは、カンマ区切り(csv)のフォーマットを使用して、シンボルメトリックスをファイルにインポートします。メトリックスを持つシンボルの型ごとに、別々のファイルが作成されます。

出力ファイルの名前は自動的に付けられます。*directory_name* が省略されると、レポートはカレントプロジェクト(cmmdOpenProjectを通して開かれたプロジェクト)のあるディレクトリ内の、metricsサブディレクトリに出力されます。

cmmdCalcMetrics は、計算を行うものの、シンボルのメトリックスをエクスポートしません。これは cmmdMetrics に代わるもので、Imagix 4D のプロジェクトサイズの設定が Very Large に設定されているときにのみ有用となり、計算されたメトリックスがセッション間でデータベースに格納されるようにします。(詳しくは、本ユーザガイドの「プロジェクトのリソース」の項を参照してください。)

デフォルトでは、これらの手続きは Reports メニューの下に現れるすべてのメトリックスカテゴリについてメトリックスをエクスポートします。*symbol_type_list* によって、作成するメトリックスファイルを制限することができます。symbol_type_list のデフォルト値は、6 つのシンボル型すべてを指定する{directory file namespace class function variable} です。

cmmdDocument [*directory_name*]

ディレクトリ *directory_name* にドキュメントを作成する手続きです。*directory_name* が省略されたときには、出力はカレントプロジェクト(cmmdOpenProject を通して開かれたプロジェクト)のある場所に基づいて、ディレクトリ(../project_name.4DD)に出力されます。

Document ダイアログの設定はセッション間で保存されます。デフォルトでは、この設定がドキュメント作成に使用されます。別の設定を使用したい場合には、DocGen シートで希望の設定を指定し、ドキュメント作成プロセスにそのシートを適用することができます。以下の変数は DocGen シート以外で制御されるドキュメント設定を制御します。Document ダイアログに示されているとおり、同時に使用できない Format と File の組み合わせがあります。

変数	設定
dcgOptions(Style)	{ UseOptionsSettings | ApplyDocGenSheet }
dcgOptions(DocGenSheet)	file_name_of_.dgn_file
dcgOptions(Format)	{ ASCII | RTF | HTML }
dcgOptions(File)	{ SingleFile | FileperSection | FileperSymbol }

cmmdDebug [*contents* [*crashsppt* [*tracefile*]]]

この手続きは、[Tools] > [Options] > [Behavior and Appearance] > [Technical Support] ダイアログを通して有効にされたデバッグトレースのサポートを重複させます。結果となるトレースファイルは、特に dbgf コマンドとともに使用される場合に、コマンドファイルの改善に役立つことがあります。

contents はダイアログからのコンテンツの選択肢のうち 1 つか、デバッグトレースを無効にするのに相当する Off にする必要があります。デフォルト値は Essentials です。*crashsppt* は 0 か 1 にする必要があります。デフォルト値は 1 です。*tracefile* は現在存在するディレクトリ内のファイルである必要があります。デフォルト値は、Technical Support ダイアログに表示された場所です。

例

以下に、注釈を付けたコマンドファイルの例を示します。このファイルには、実際の cmmdXXX コマンドと、その前後で変数設定を指定する"set"コマンドと"source"コマンドが含まれています。コマンドファイル内の"#"で始まる行は、コメントとして扱われます。サンプルファイル内のパスは Windows 用のものですが、パス名が適切に処理されるように、バックスラッシュ"\"ではなくフォワードスラッシュ"/"が使用されていることに注意してください。

```
# 最初のプロジェクトを開き、プロジェクトを再生成して、プロジェクトにソースコードの現在の状態が
# 反映されるようにします。次に、2 つのレポート設定をコマンドファイルから(残りの設定を最新の
# gui 設定から)使用することにより、2 つのレポートを生成します。レポート設定の完全なリスト
# は、../imagix/user/user_cmd.txt を参照してください。
cmmdOpenProject c:/test/projects/project_33.4D
```

```
cmmdRegenerateProjectData

set dfgRpt(Unused_Variables.global) 1
cmmdReport Variable_Flow_Checks..Unused_Variables c:/test/rpt/uv.txt
set dfgRpt(Recursive_Functions.order) alpha
cmmdReport Function_Flow_Checks..Recursive_Functions c:/t/r/rf.txt

# 2 番目のプロジェクトを開きます。1 つのコマンドファイルで複数のプロジェクトを開くことができます。
# 新しいプロジェクトを開くと、現在開かれているプロジェクトは閉じられます。

# プロジェクトパスにスペースが含まれている場合、引用符を使用しなければなりません。
cmmdOpenProject "c:/test/projects/space dir/project_600.4D"

# すべてのレポートを生成します。現在のコマンドファイルを拡張する、rpt_settings ファイルを
# 読み込みます(source を使用)。これは C/C++の"#include"文に相当します。
# rpt_settings ファイルには、複数のコマンドファイル間で共有される一連のレポート設定を指定
# するために、連続する"set"コマンドが含まれることがあります。スプレッドシートへのインポートを
# 容易にするため、レポートを.csv 形式で出力します。
# 出力パスにはスペースが含まれるため、引用符を使用する必要があります。
source c:/test/reports/rpt_settings
set vtgPrint(Format) csv
cmmdReportMultiple "c:/test/space dir/reports"

# デバッグ用に fprintf 形式の文を作成します。
cmmdDebug Essentials 1 c:/test/trace.log
dbgf About to start report export - format is $vtgPrint(Format)
cmmdDebug Off

# Imagix 4D への逆インポート([Report] > [Import])を可能にし、すべてのシンボルを
# タグ付きで表示させて GUI によるナビゲーションに対応させるために、特定のレポートを.4dr
# レポートとして生成します。レポートが容易に読み取れるように、ASCII 形式を使用します。
set vtgPrint(Format) ASCII
cmmdReportExport Source_Checks..Expressions "c:/test/space dir/expr.4dr"
cmmdReportExport Task_Flow_Checks..Reentrant_Functions "c:/t/s d/rf.4dr"

# HTML ドキュメントを作成します。../imagix/user/doc_gen ディレクトリから my_style.dgn
# というスタイルシートを適用して、ドキュメントの形式とコンテンツを制御します。出力パスにスペースが
# 含まれる場合、引用符を使用する必要があります。
set dcgOptions(Style)    ApplyDocGenSheet
set dcgOptions(DocGenSheet)  my_style
cmmdDocument "c:/test/space dir/demo.4DD"
```

2 番目のコマンドファイルのサンプルは、通常は夜間のコードビルド処理の一部として実行されると考えられます。翌日ユーザが操作するとき、Imagix 4D データベースが最新の状態であり、非常に速くロードされることに気付くでしょう。

```
# ビルドされるコードを表すプロジェクトを開きます。
cmmdOpenProject c:/test/projects/project_dev.4D
# ソースコードの現在のバージョンを解析します。これによって、プロジェクトが次回開かれたとき、
# Imagix 4D データベースが最新のソースコードを表し、迅速にロードされます。
cmmdRegenerateProjectData
# メトリックスのデータを計算します。Imagix 4D のプロジェクトサイズが Very Large に設定され
# ていれば、計算済みのメトリックスがディスクのデータベースに格納されるので、プロジェクトが次回
# 開かれたときに再計算する必要はありません。
cmmdCalcMetrics
```

Flow Check レポートのいくつかが定期的に使用されている場合には、以下に示すコマンドファイルを夜間ビルドで使用してレポートファイルを生成すれば、後のセッションの間に迅速にレポートファイルがインポートでき、ユーザが解析処理を待たずに済むでしょう。

```
#ビルドされるコードを表すプロジェクトを開きます。
cmmdOpenProject c:/test/projects/project_dev.4D
# ソースコードの現在のバージョンを解析します。これによって、プロジェクトが次回開かれたとき、
# Imagix 4D データベースが最新のソースコードを表し、迅速にロードされます。
cmmdRegenerateProjectData
# Imagix 4D への逆インポート([Report]>[Import])を可能にし、すべてのシンボルをタグ付き
# で表示させて GUI によるナビゲーションに対応させるために、特定のレポートを.4dr レポートとして
# 生成します。レポートが容易に読み取れるように、ASCII 形式を使用します。
set vtgPrint(Format) ASCII
cmmdReportExport Task_Flow_Checks..Reentrant_Functions "c:/t/s d/rf.4dr"
```

付録 C.パターンマッチングの形式

Grep Tool ウィンドウ、Filter メニューの Add ダイアログなど、Imagix 4D ではさまざまな機能がパターンマッチングを用いてユーザ指定のパターンに合致する文字列または名前を識別します。このような機能では、2 つのパターンマッチングフォームのうち 1 つが使用されます。glob スタイルのパターンマッチングは Unix シェルで使用されている規則をファイル名展開に適用します。正規表現パターンマッチングは Unix の regexp コマンドで使用されている規則を適用します。

glob スタイルパターンマッチング

glob スタイルは、ふたつの形式のうちでより簡単なほうですが、より制約があります。glob スタイルのパターンには、次のような特殊な文字を含めることができます。

?	任意の 1 文字とマッチします。
*	任意個（0 個も含む）の文字がつながった文字列とマッチします。
[*chars*]	[ ]内に指定された文字の集合（chars）のうちの任意の 1 文字とマッチします。文字の集合のなかに a-b の形式の文字列が含まれていれば、a から b までの（a、b も含む）いずれかの文字とマッチします
\x	文字 x とマッチします。

正規表現パターンマッチング

正規表現は、ふたつの形式のうちでより複雑なほうですが、より効果的です。この形式では、以下の規則を用いてパターンのマッチングを判断します。

正規表現とは、任意個（0 個も含む）のピースがつながったものをいい、最初のピースとマッチするもののあとに、2 番目のピースとマッチするもの、3 番目のピースとマッチするもの……とつながったパターンとマッチします。

ピースとは、アトムのことであり、場合によっては*、+、または?が後に付きます。*があとに付いている場合には、そのアトムとマッチするものの任意個（0 個も含む）の列とマッチします。+があとに付いている場合には、そのアトムとマッチするものの 1 個以上の列とマッチします。?が後に付いている場合には、そのアトムとマッチするもの、あるいは空文字列とマッチします。

アトムとは、丸括弧で囲まれた正規表現（その正規表現とマッチするものとマッチ）、レンジ（下記参照）、.（任意の 1 文字とマッチ）、^（入力文字列の最初の空文字列とマッチ）、$（入力文字列の最後の空文字列とマッチ）、＼とそのあとに続く 1 文字（その文字とマッチ）、あるいは他意のない 1 文字（その文字とマッチ）のことです。

レンジとは、[ ]で囲まれたひとつの文字列のことです。通常はその文字列に含まれる任意の 1 文字とマッチします。文字列が^で始まる場合には、残りの文字列に含まれない任意の 1 文字とマッチします。文字列のなかに-でつながった 2 文字がある場合には、ASCII コード上でそれらの一方の文字から他方の文字までのすべての文字とマッチします（たとえば、[0-9]はすべての 10 進数字とマッチします）。文字列にリテラル]を含めるには、それを最初の文字にします（あとに^を付けることは可能）。リテラル-を含めるには、それを最初または最後の文字にします。