課題研究報告書

ソフトウェア開発環境における構成要素の分類と競合回避手段に関する調査研究

指導教官　　鈴木　正人　　准教授

審査委員主査　鈴木　正人　　准教授
審査委員　　落水　浩一郎　特任教授
審査委員　　青木　利晃　　准教授

北陸先端科学技術大学院大学
情報科学研究科

0910701 KWEON HYUNG HAHN

提出年月：2013 年 2 月

課題研究報告書

ソフトウェア開発環境における構成要素の分類と競合回避手段に関する調査研究

北陸先端科学技術大学院大学
情報科学研究科

KWEON HYUNG HAHN

2013 年 3 月

目次

1. はじめに

1. 1. 背景

現代ソフトウェア開発において、既存のコマンドライン環境のほか、Eclipse, Microsoft Visual Studio などのソフトウェア統合開発環境(IDE)が、ソフトウェア開発に主に利用されているのが現状である。これらのソフトウェア統合開発環境は、プログラマーにより作成されたプログラム言語のソースコードを入力とし、実行上のパラメータの指定を伴うコンパイラー・リンカーなどのツール群の呼び出しを行い、その出力結果を統括的に管理できる環境である。

これらのツール群の動作を観察すると、対象とする言語および環境の特性により詳細は異なるものの、決められた手順を通じて入力であるソースコードが、最終的に実行可能コードに変換され、実行される過程が行われるのがわかる。変換に関わるほとんどの過程において、成果物の特性を決定するための様々な情報の追加指定が必要である。これらの情報は、ターゲットとする環境に依存してパラメータとして変換過程に影響を与える。これらのパラメータ指定を変更することで、同一の入力に対しても異なる結果が得られる（図 1）。

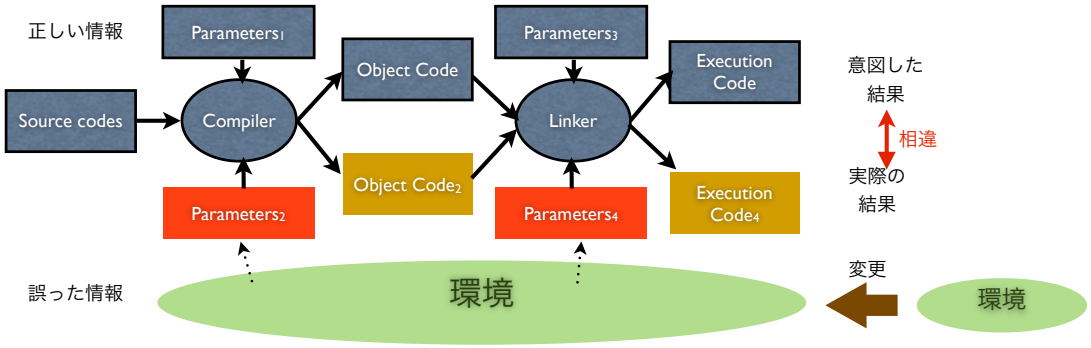


図 1 ソフトウェア開発環境上の内部アクションと環境に基づく情報との関係

問題はこれらの情報は、対象環境の至る所に存在し、その情報に基づいて具体的パラメータが作成されることである。

具体的には、ソフトウェア開発においては、開発環境にかかわるツール自身の変更だけでなく、関連するライブラリおよびフレームワークのバージョ

ンアップによる機能の呼び出し方法(インターフェース)や内容などの変更、または機能の用件定義による特定のバージョンへの機能制限など、環境の変更が頻繁に発生し、その変化は開発の様々な段階に関わるパラメータに影響を与える。パラメータは単一または複数のファイルの形態で開発ツール群および実行に関わる別環境に影響を与える（図 2）。

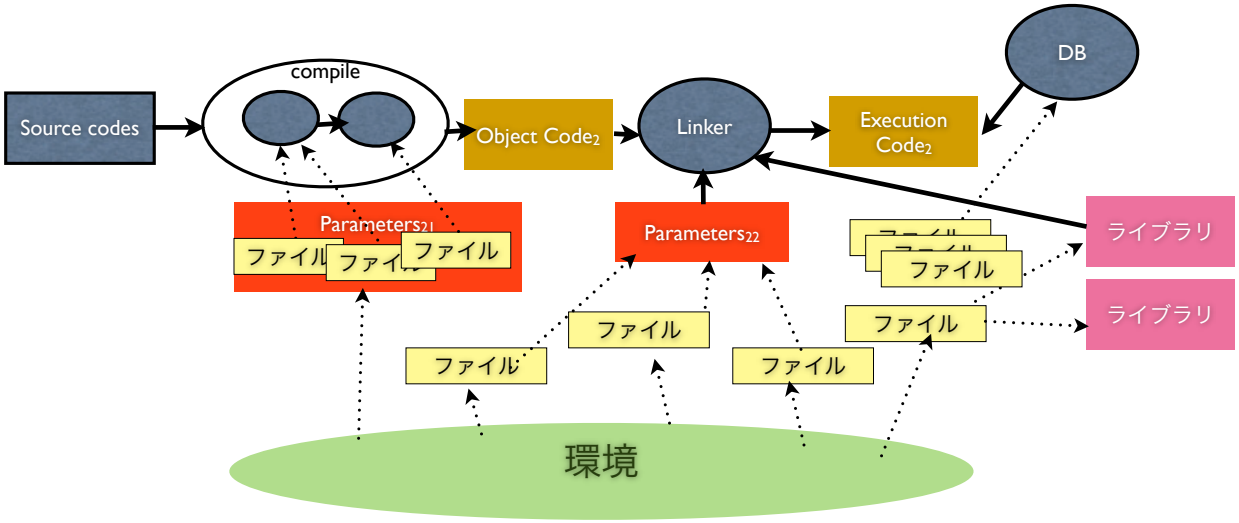


図 2　開発の様々な場面に影響を与えるパラメータファイル

この点を考慮し整合性がある設定を与えないと、成果物であるプログラム意図したとおり動作しない。このような不具合は、開発者がソフトウェア開発に対する設定情報の意味を正しく理解していない場合にはその発見と修正が困難である。さらに、様々な要因により環境が部分的に変化することで、実行結果の一貫性が失われる可能性が発生する。

1.2. 目的

上記のように開発段階に関わる付加情報の不整合が要因で、成果物が本来のプログラマーの意図と実際の動作に違いが発生することを、本研究では**競合(conflict)**と呼ぶこととする。

この競合が発生する原因を分析することで、競合が起こりそうかを予め検出するための方法の提案が可能になり、その回避方法を検討することができる。本研究では、ソースコード以外にプログラムの挙動の決定に影響をあたえる情報（**挙動決定情報**）を定義しその抽出および競合の検出方法を調査・分類することで、環境が変化した場合における競合の回避を目的とする。

対象プログラムの構成および変換ツール群の設定情報に基づき、言語要素

と環境要素および要素間関係から挙動決定情報を抽出する。具体的な言語および環境の事例に対し挙動決定情報を分類し、発生する可能性のある競合の種類を特定し、検出および回避のためのチェックリストを与える。

1.3. 論文の構成

本論文の構成として、まず 1 章は研究の背景および目的を述べ、2 章ではソフトウェア統合開発環境を中心とした現状の開発環境を整理し、問題としての競合を取り上げる。3 章と 4 章では開発環境で発生する各過程を構成する要素として言語要素と環境要素、および挙動決定情報を定義し、これらに基づいて競合を定義する。5 章では現状使用される主な開発環境においてケーススタディを行い、最後に研究内容のまとめと今後の課題を述べる。

2. 現状の開発環境と問題

本章では、プログラム言語を実行するための開発環境について考察し、開発環境上で発生する競合について定義する。

2.1.開発形態による環境の分類

本節では開発の形態からみて開発環境を分類する。開発の手続きの記述が単線的に与えられ、逐次的に変換されるタイプと、ユーザの自由度が高い変換を支援するタイプがある。

2.1.1. コマンドライン環境

開発の段階から図 3 のように逐次的に変換されたる環境として、コマンドラインによるインターフェースを使用し、ユーザのコマンド入力によりプログラムのコンパイル、リンク、実行などの手順を指定する方式が挙げられる。プログラムのコンパイルのための手順の指定はプロセスファイル（GNU make の Makefile や Ant[1]の build.xml など）により記述される。このプロセスファイルは、成果物の特性を獲得するために、または成果物が動作する対象環境の特性に適合させるために用意された、いくつかの選択肢の中から開発者が適切なものを選択するより自動生成されることもある。伝統的に GNU build system がこの目的で広く使用されてきたツールの１つである。

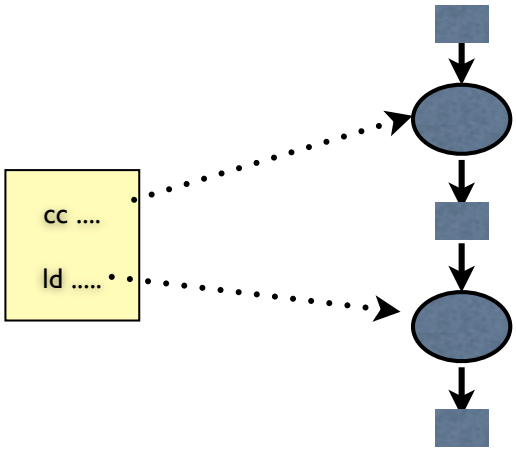


図 3　コマンドライン環境

2.1.2. 統合開発環境

複数のツールから構成され、一般にはGUIを使用してユーザがツールの起動を自由に指示できる方式である。対象になるプログラムのソースコードやライブラリおよび作成手順を表すプロセスファイルなどは**プロジェクト**という単位で管理される。プロジェクトは対象ソースコードおよびプロセスファイルだけでなく、デバッグ・リリーズなどの特性を実現するためのコンパイル関連オプションも保持している。

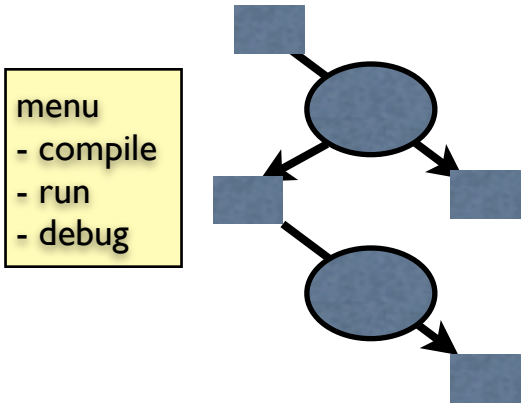


図 4 自由度が高い開発環境

以下にいくつかの例を紹介する。

(1) Eclipse

Eclipseは、IBMにより開発されたオープンソース系ソフトウェア統合開発環境の１つである。Java 言語を筆頭に、C/C++, Ruby などいくつかの言語の開発を支援する。

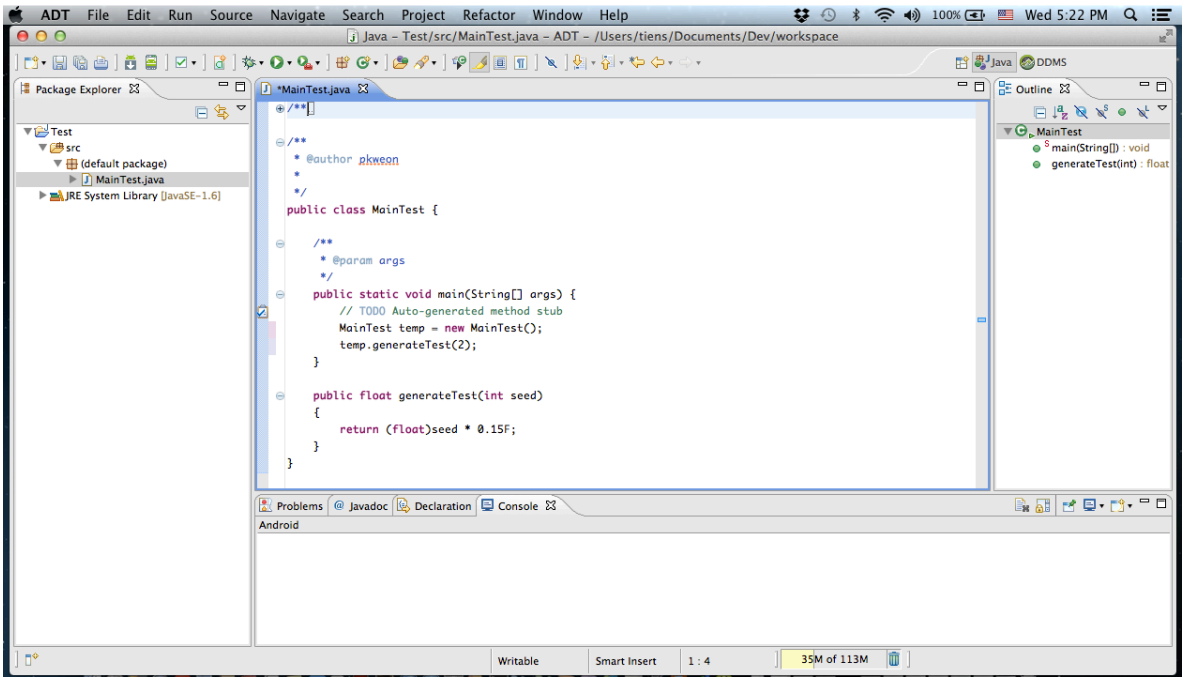


図 5　Eclipse 画面の例（Android Development Toolkit から）

Eclipse の特徴としてプラグインという仕組みにより、様々な機能のツールを統一的なインターフェースで統合することが可能であることが挙げられる。このプラグインによる拡張により、Eclipse は標準 Java アプリケーションの他、様々なウェブアプリケーションフレームワーク上のアプリケーション、さらに最近はモーバイル環境である Android など、多様な環境で動作するアプリケーション作成のためのツールとして位置づけられている。

(2) Visual Studio

Visual Studio[2]は Microsoft 社によって開発され統合開発環境である。標準的な C/C++言語および Visual Basic 言語の他、C#, Visual Basic.NET, F#, Visual C++, Ruby 言語の一種である IronRuby などいくつかの言語の統合開発環境を提供する。

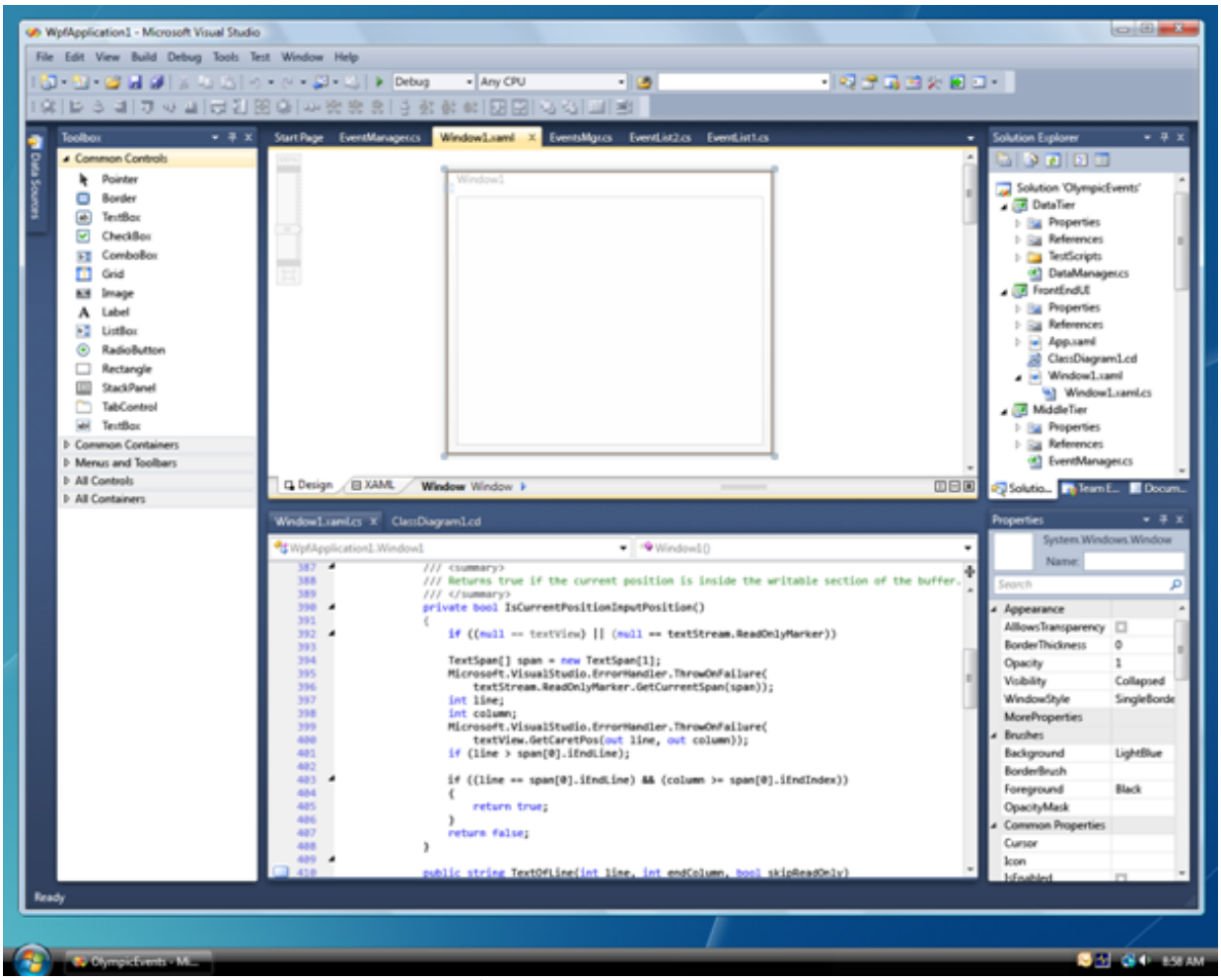


図 6 Visual Studio

従来の Win32 形式のプログラムのほかに、Microsoft の次世代言語実行環境である.NET Framework に対応している。ASP.NET を利用することでウェブアプリケーションの作成が可能であり、IDE 上での画面 UI のデザイン、デバッグなどの作業もが可能である。なお、対象プログラムのコンパイルの際、構成管理による複数の構成をサポートし、用途に応じてデバッグ情報を含む実行用ファイル作成や、実行時のリリース用に最適化された実行ファイル、特定ターゲットに対応したバイナリの作成などを支援する。

2.2. 実行形態の観点からみたアプリケーションの分類

競合の発生を分類するために、まず対象となるアプリケーションの実行形態による分類を行う。

(1)スタンドアローンアプリケーション

独立した単一のアプリケーションが環境上で動作する。事務処理系のプログラムのように入力データを連続的に加工して出力データを得るもの(ストリーム変換型)と、非同期的な入力と出力を扱うインタラクティブ型がある。

(2)多重環境連携型アプリケーション

単一プログラムの環境ではなく複数の環境が連携して動作する。通常はクライアントとサーバの双方でアプリケーションが実行される。例えば最近はウェブフレームワークを使用したアプリケーションがこの分類に属し、入出力はウェブの基本的な通信規約であるhttpに基づいてHTMLを仲介して実行される。

2.3. 競合が発生する例

以下に競合の発生可能性があるケースについて述べる。

事例1:ライブラリの別バージョンをリンクした場合、正しく動作しない。

特定の外部ライブラリ上に想定の出力を行う公開関数があり、呼出し元はその関数の呼び出し結果で生成されたファイルを参照する場合を想定しよう。開発中の都合で同じ名称のライブラリが更新された場合、そのライブラリ上の関数の呼び出し結果が想定と異なる可能性がある。

以上の内容を図で表示した例を図7に示す。

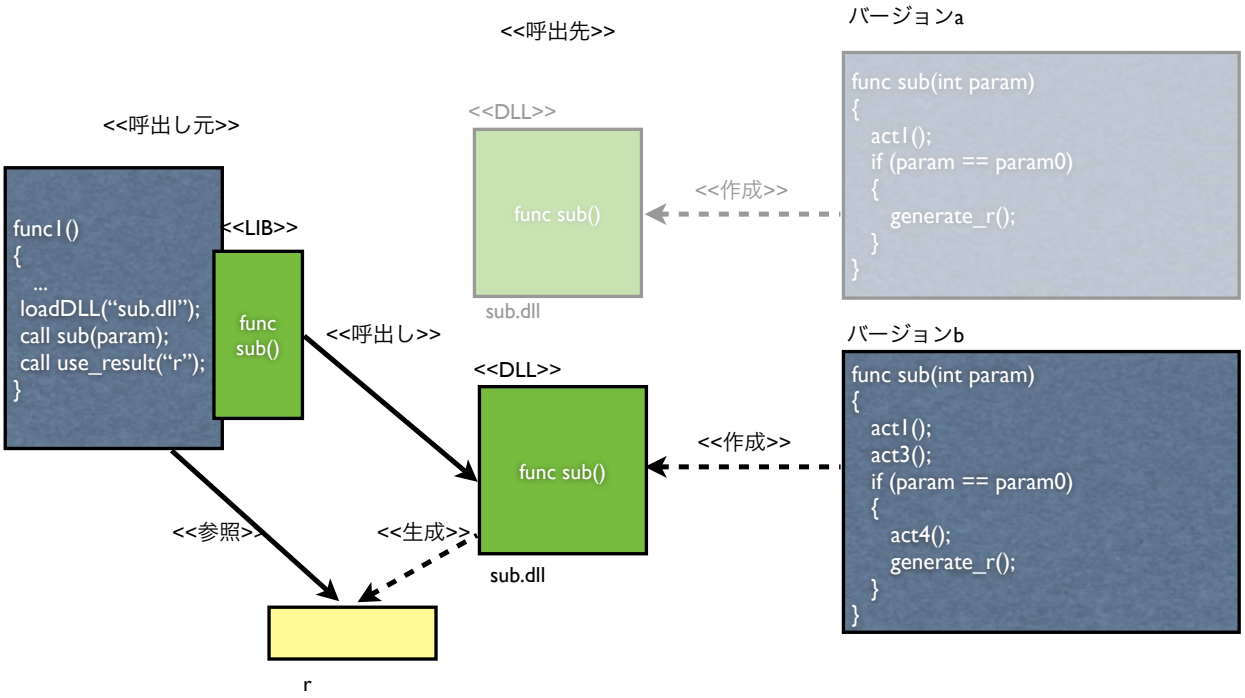


図 7　ライブラリの別バージョンをリンクした場合

この場合は(a)対象ライブラリが異なったり、(b)関数名の内部処理に変更があったり、(c)対象ライブラリ作成もしくは呼出し元の開発作業中のライブラリのリンクの際、コンパイルオプションの指定変更により実際の動きがことなったりするケースを含め、開発段階および実行時の競合の可能性が潜んでいる例である。

事例２：設定ファイルの場所の変更が環境に反映されていない。

ウェブアプリケーションの開発において、用途別で提供されるコンポーネントで構成されるフレームワークを利用する場合が多い。この際、フレームワークを利用するために、ソースコードや実行可能コードの他に、フレームワークの動きを指定する設定内容を保持するファイルが存在する。このような設定ファイルは、フレームワーク自体のバージョンアップ、機能変更などにより、もしくは要求の追加変更により置き場所が変わる場合がある。このような変更があったのにも関わらず反映されていない場合、プログラムの動作が意図したものとことなる可能性がある。

通常のプログラムの場合は設定情報の不整合はエラーメッセージなどに

より検出される場合が多いが、環境の不整合が原因である場合は、メッセージが表示されない場合が普通であり、この場合検出が遅れる結果につながる。

3. 競合を決定する要因：言語要素と環境要素

競合を検出するためには、まず環境の現在指定されている設定のもとでプログラム野動作に必要な情報を特定しなければならない。本章ではプログラムの動作を決定するのはソースコード自身とライブラリなどの外部情報の２種類があり、前者の構成要素を言語要素、後者の構成要素を環境要素と定義する。

3.1. 言語要素

本節ではプログラムを構成する概念をその役割に従って分類したものを言語要素(language elements)として定義する。

プログラムの最初の入力になるソースコードに注目すると、ソースコードの中には、関数の定義および参照が含まれる。C++のようなOOPの場合はクラス定義もあり、言語の種類によって定数マクロなどを含む場合がある。

C言語の例として図8の左側の図のようなコードがあって、コンパイルされることを想定する。

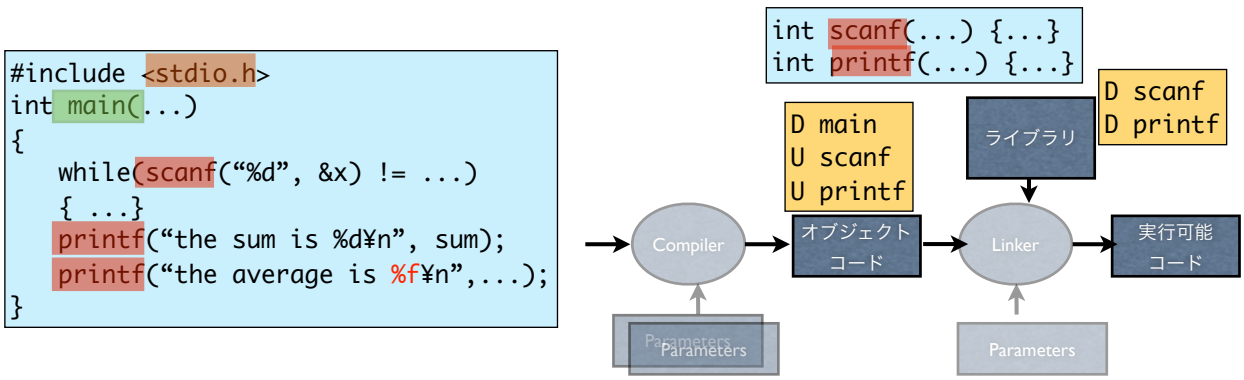


図 8　言語要素

このソースにはmain関数の定義とscanf/printfの参照がある。このソースコードが次々と開発の段階を経て形を変えながら実行可能コードまでたどり着く。

オブジェクトコードでは関数の参照と定義が混在して、ふくまれる参照を解決するため別のところのその関数の定義をもつファイル（ヘッダ・ライブラリ）とのマッピングが行われます。この過程がリンクであり、一方関数の参照に対する関数の定義を持つファイルとしてヘッダファイルと、実装を持つライブラリがある。

以上に挙げられるプログラミング言語で作成したプログラムのソースコード（変数・関数・クラス定義)、オブジェクトコード、実行可能コード、ライブラリなどを言語要素として挙げる。これらの用語はプログラム開発の現場で一般的に使われるが、本研究では下記の通り定義する。

(1) ソースコード

１つもしくは複数の変数、関数およびプロシージャ、クラスの定義および実装の記述を含むファイル。ソースコード中に含まれ、公開されている定義はすべて言語要素として扱う。

(2) オブジェクトコード

ソースコードがコンパイラにより変換された結果ファイル。

(3) 実行可能コード

複数のオブジェクトコードとライブラリが結合した結果。

(4)ライブラリ

事前に作成され、ソースコードと結合することで実行可能コードを得るためのファイル。実行可能ファイル作成時に結合される静的リンクライブラリと、実行可能ファイルが実行される際にスタブを通じて呼び出される動的リンクライブラリがある。

(5) リソースファイル

ソースコードで作成されるプログラムが内部で参照する非実行可能データファイル。特定のコンテキストで特定の意味を持つことがある。例えば Visual Studio で使われる Win32 プログラムプロジェクトに含まれる.rc 拡張子のリソースファイルは Resource.h ヘッダファイルと緊密に関係を持っ

て、画面の構成を指定する役割を持つ。Android 環境のリソースファイルと中に layout フォルダ配下の.xml ファイルも画面構成の役割を果たす。リソースファイルはファイル形式として専用拡張子の他、実行ファイル、ライブラリの形をしていることもある。

(6) その他
ヘッダファイル、定数マクロなど

3.1.1. 言語要素の抽出方法

対象のプログラム開発環境から、言語要素を選択するには、変数および関数の定義が競合と関連する言語要素を抽出する方法を表 1 にまとめる。

表 1　主な言語要素の抽出方法

言語要素	抽出方法
ソースコード	ユーザが作成したすべてのファイルを含める。
変数	ユーザ定義変数で外部に公開される変数を含める（大域変数、公開されている変数など)。
関数･プロシージャ	ユーザ定義変数で外部に公開される関数を含める。
定数･マクロ定義	C/C++言語の#define 文、#ifdef、#ifndef 文
クラス定義	クラスはユーザ定義変数と関数の集合であり、公開されるメンバ変数・関数が 1 次対象になる。
オブジェクトコード	要素として選択したソースコードと関連するすべてのファイルが対象になる。
実行可能コード	ソースコード･オブジェクトコードと関連するすべての実行コードを含める。
ライブラリ	実行可能コードと関わる静的および動的リンクライブラリを対象に含める。
リソースファイル	ユーザが作成したソースコードで参照するリソースファイルを含める。

3. 2. 環境要素

ソフトウェアが開発される開発環境でコンパイル、リンクなどの開発作業中に生成されるファイル形式を観察すると、元の入力になるファイルが、特定のパラメータと共に呼び出されるツールにより解析され、別のフォーマットを持つ出力ファイルとして生成される過程が発見される。例えば、C 言語で作成された**ソースコード群**があり、この中で特定の機能を実現するために必要とされる**ソースコード**のみを選択することがある。この作業はソースコードを管理するツール（構成管理ツール）により実現される。選択されたソースコードはそれぞれコンパイラによりコンパイルされ、**オブジェクトコード**がその結果として生成される。オブジェクトコードは、必要とするライブラリと共にリンカーにより結合され**実行可能コード**が生成される。ストリーム処理型の場合、実行可能コードは**入力データ**を**出力データ**に加工する働きを持つ。インタラクティブ型の場合は、入力のイベントと生成された時系列を入力データ、出力イベントと生成された時系列を出力データと見なすことで、原理的には同様の構成が適用できる。図 9 はこの過程を表現したものである。

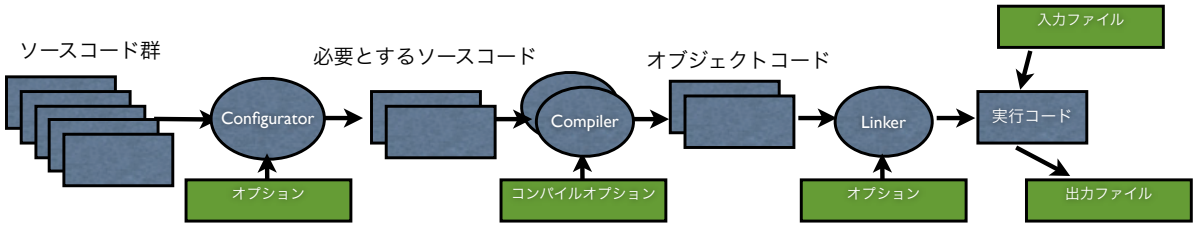


図 9　一般的な実行可能コード生成までの過程

以上のように開発環境上の各過程で必要とするコードやツールオプションなどの要素を**環境要素**（environmental elements）と呼ぶこととする。具体的には、過程上で呼び出されるツールがあり、ツールへの入力となるファイルとツールへの指令情報であるオプション、これらの結合により生成される出力結果の関係が連鎖的に発生するが、これらに参加する要素を環境要素として取り上げることができる(図 10)。

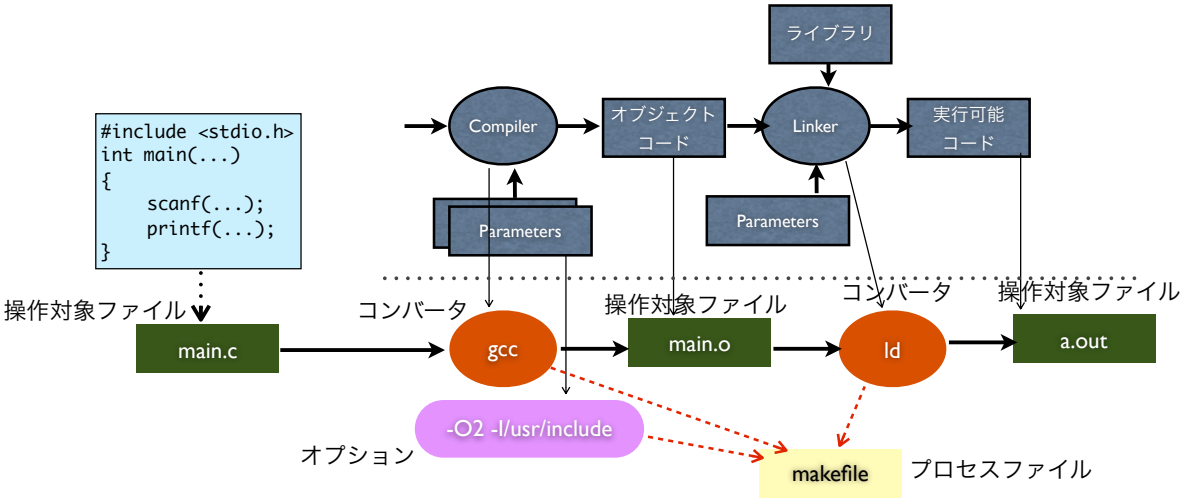


図 10 環境要素

本章では、環境要素をプログラムが生成される時と、生成されたプログラムが起動する時に分けて考察することとする。

3.3. 作成時に必要な環境要素

プログラムの作成時に必要な環境要素には以下のような種類がある。

(1) 操作対象ファイル

プログラムを構成するファイル（ソースコードなど)および各過程で生成されるファイルを操作対象ファイル(target file)と呼ぶ。C 言語ではソースファイル(*.c)、ヘッダファイル(*.h)、オブジェクトファイル(*.o、*.obj)などが相当する。Java 言語ではクラス定義コード(*.java)およびバイトコード(*.class)、ウェブプログラミングで使われる*.jsp ファイルなどが相当する。コンパイルによるオブジェクトコードを生成しないで直接ランタイム環境から実行される言語環境もあるが、この場合対象ソースコードのみ操作対象ファイルに含まれることとする。一方、ウェブフレームワークの種類ごとに様々な拡張子のファイルが存在する。

操作対象ファイルの例を表２にまとめる。

表 2 操作対象ファイルの例

言語・環境	捜査対象ファイル	説明
C/C++言語・スタンドアローン環境	.c, .cpp	関数定義の実行ファイル
	.h, .hpp	ヘッダファイル

	.o(.obj)	オブジェクトファイル
.NET 基盤言語	.cs	C#クラス群の定義ファイル
	.vb	Visual Basic クラス定義
Java 共通	.java	クラスの定義ファイル
	.class	クラスファイル
（アーカイブ）	.jar	アーカイブファイル
Java ウェブフレームワーク環境	.jsp	JavaServerPages
	.xml	JSF Facelet, 設定ファイルなどで使用される
	.war	Web アーカイブファイル

XML 文書を定義する.xml ファイルの場合、環境設定ファイルだけでなく UI の定義で実現されている場合もため、その役割によりどちらかを判別する必要がある。

(2) ライブラリファイル

事前に作成されたファイルで、操作対象ファイルと結合することで実行可能コードを得るためのファイルを**ライブラリファイル(library file)**と呼ぶ。C 言語ではほとんどのプログラムが、入出力や OS サービスを利用するために標準ライブラリを使用する。標準ライブラリ以外にも DB 接続やウェブ処理などの特定の機能を実現したライブラリを適宜に使用する場合もある。

Java 言語の場合、操作対象ファイルもライブラリも同一形式のアーカイブ(*.jar)で実現されているため、その役割によりどちらかを判別する必要がある。表 3 にライブラリファイルの例を示す。

表 3　ライブラリファイルの例

言語・環境	ライブラリファイル	説明
C/C++/Fortran/…	.lib, .o	リンク時に参照される。
ライブラリアーカイブ	.a	静的ライブラリ
	.dll, .so	動的ライブラリ
	.jar	アーカイブファイル

(3) コンバータとオプション

操作対象ファイルを別形式に変換する役割を持つツールおよびアクションを**コンバータ**(converter)と呼ぶ。コンパイラはソースコードからオブジェクトコードに、リンカーは複数のオブジェクトコードとライブラリを実行可能コードに変換するコンバータの例である。実行可能コードを特定場所に配置したり、機能別クラスコードをパッケージングしたり、ソースコードが必要な情報のみを抽出してドキュメントファイルを生成したりする処理もコンバータとして解釈することができる。表 4 にコンバータの例を示す。

表 4　コンバータの例

コンバータ	コンバータ実体	説明
コンパイラ	cc, gcc, g++, cl.exe	C/C++
	javac(.exe)	Java
	csc.exe	C#
	vbc.exe	Visual Basic
リンカー	ld, link.exe	
コンパイラドライバー	(cc), msbuild.exe, make, ant	
デプロイツール	install, (cp, mv)	Shell command
パッケージングツール	ar, jar(.exe)	
ドキュメントビルダー	javadoc(.exe)	Java
構成管理ツール	(git)	

コンバータの実行にはオプションの指定が必要とする。暗黙的に処理されるオプションと必要に応じてユーザが指定するオプションなどがある。なお、コンバータには同じ種類でもバージョン毎、メーカー毎に異なるツールが存在し、作業ごと分別して使用されたり、併用されたりする。

本研究では、同一コンバータについて異なるオプション、もしくは異なるバージョンを使う際、それぞれを別ものとして扱うこととする。(オプションがつく同じコンバータを別ものとして扱うこととする)。後述する構成管理情報もコンバータのオプションとして記述する。

(4) プロセスファイル

コンバータの呼び出し手順および条件などを記述したファイルを**プロセスファイル**(process file)と呼ぶ。プロセスファイル中では手順および条件は専用の言語により記述され、更新が必要な操作対象ファイルのみを作成するコンバータが呼び出される。総合ビルドツールである make が使用する Makefile や、ant が使用する build.xml ファイルはプロセスファイルの例である。

(5) 構成管理情報

プログラムの目的、ターゲットになるプロジェクト、変更された特定のバージョンなどを管理する構成管理(software configuration management)[3]により、プログラムを作成するソースから対象ソースを選択する基準となる情報を構成管理情報として扱い、環境要素の１つとして含める。

構成管理用ツールとして、リポジトリなどの機能を持ち、ソフトウェアの変更履歴を管理するバージョン管理システムから、ソフトウェアの開発から運用までの段階に渡り、ソフトウェアを含む資産の変更管理を目的としたツールまであるが、本研究では、ソースコード群から特定の目的のソースコード群を選択する機能の面を対象とする。このようなツールはコンパイラなどの開発ツールの外部ツールとして用意される場合と(例: git)、IDE に含まれる場合がある(Visual Studio) 。

(6) 名称実体対応記述

C 言語のプログラムをコンパイルするためのコンバータにはいくつか種類があるが、その役割は共通しており「コンパイラ」と呼ばれる。プロセスファイル中ではコンバータを呼び出す際にはコンパイラとのみ記述し、そのツールと実体との対応は別に定義される。

本研究ではこの役割と実体との対応関係の記述を**名称実体対応記述**(mapping description)と定義し、以下の２種類を考える。

- コンバータ名称実体対応記述(mapping description for converter)

コンバータの役割と実体の対応を記述したものである。

例えばプロセスファイル conf47 に“cc = gcc47”という記述があった場合には、cc(C 言語専用コンパイラ)が呼ばれた際には PATH 上の登録済みの場所に存在する gcc47 という実ファイルで示されるコンバータ実体を起動

するものと定義する。(図 11 を参照) 以降、便宜のため conf47(“cc=gcc47”) と表示する。

図 11　名称実体対応記述の例

- ライブラリ名称実体対応記述(mapping description for library)
　ライブラリの役割と実体の対応を記述したものである。
　例えばプロセスファイル conf47 に“stdlib = stdlib47”という記述があった場合には、stdlib(すべてのプログラムで使用する標準ライブラリ)が使用される際には、stdlib47 というファイルを対応させるものと定義する。

3.2.1. 事例

　以下に C 言語コンパイラの異なるバージョンの例として、C コンパイラの異なるバージョンとして gcc4.7 と gcc4.8 を仮定し、C 言語の任意のソースファイル（２つの関数定義、１つのメイン処理）からプログラムを実行して結果を得るまでに使用する構成要素を示す。

　まず対象ソースコードの例をリスト 1,2,3 に示す。

```
/* x.c */
#include <x.h>
float f_x(float sum, float param)
{
        float new_sum = sum + param;
        printf(“%f <- %f”, new_sum, param);
        return new_sum;
}

/* x.h */
#pragma once
#include <stdio.h>
```

リスト 1　ソースコードの例(C 言語の関数定義)

```
/* y.c */
void f_y(int param, float param2)
{
        // ….
}

/* y.h */
#pragma once
#include <stdio.h>
void f_y(int, float );
```

リスト 2　ソースコードの例(C 言語の関数定義)

```
/* main.c */
#include “x.h”
#include “y.h”
#include <math.h>
#define PI 3.141592
int main(int argc, char *argv[])
{
        float a = 0f;
        int r = 0;
        while(scanf(“%d”, &r) != EOF)
        {
                a = f_x(PI, pow(r, 2));
                f_y(r);
                // 結果の出力
        }
}
```

リスト 3　ソースコードの例（C 言語のメイン関数定義）

プログラムの実行に使う入力ファイルの例をリスト 4 に示す。

```
1
2
3
4
…
```

リスト 4 入力ファイルの例

プログラムの予想される出力結果の例をリスト 5 に示す。(a 変数をそのまま出力する場合の結果)

```
3.14159265358979
12.5663706143592
28.2743338823081
50.2654824574367
…
```

リスト 5　予想される出力結果の例

以上のソースコード群をビルドするための Makefile の例をリスト 6 に示す。

```
run:   a.out < input > output
a.out: x.o y.o main.o
       $(LD) x.o y.o. main.o
…
```

リスト 6 Makefile の例

以上の例から得られる環境要素を列挙する。

(1) 共通

gcc4.7 と gcc4.8 に共通する要素を表 5 に示す。

表 5　共通環境要素

環境要素	対象
捜査対象ファイル	x.c, x.h, y.c, y.h, main.c, input(入力ファイル)
ライブラリファイル	stdlib.a
プロセスファイル	Makefile

(2) gcc4.7 の場合

gcc 4.7 を使用する際追加で使用する環境要素を表 6 に示す。

表 6　gcc4.7 の場合の環境要素の例

環境要素	対象
コンバータ名称実体対応記述	conf47 (“cc=cc47, ld=ld47)
コンバータ本体	/usr/local/gcc47/bin/gcc47 /usr/local/gcc47/bin/ld47
ライブラリ名称実体対応記述	lib47(“stdlib = stdlib47”)
ライブラリ本体	/usr/local/gcc47/lib/libstdlib47.so

(3) gcc4.8 の場合

gcc 4.8 を使用する際追加で使用する構成要素を表 7 に示す。

表 7　gcc 4.8 の場合の環境要素の例

環境要素	対象
コンバータ名称実体対応ファイル	conf48 （“cc=cc48, ld=ld48)
コンバータ本体	/usr/local/gcc48/bin/gcc48 /usr/local/gcc48/bin/ld48
ライブラリ名称実体対応ファイル	lib47(“stdlib = stdlib48”)
ライブラリ本体	/usr/local/gcc48/lib/libstdlib48.so

以上の内容を図に表した結果を図 12 に示す。

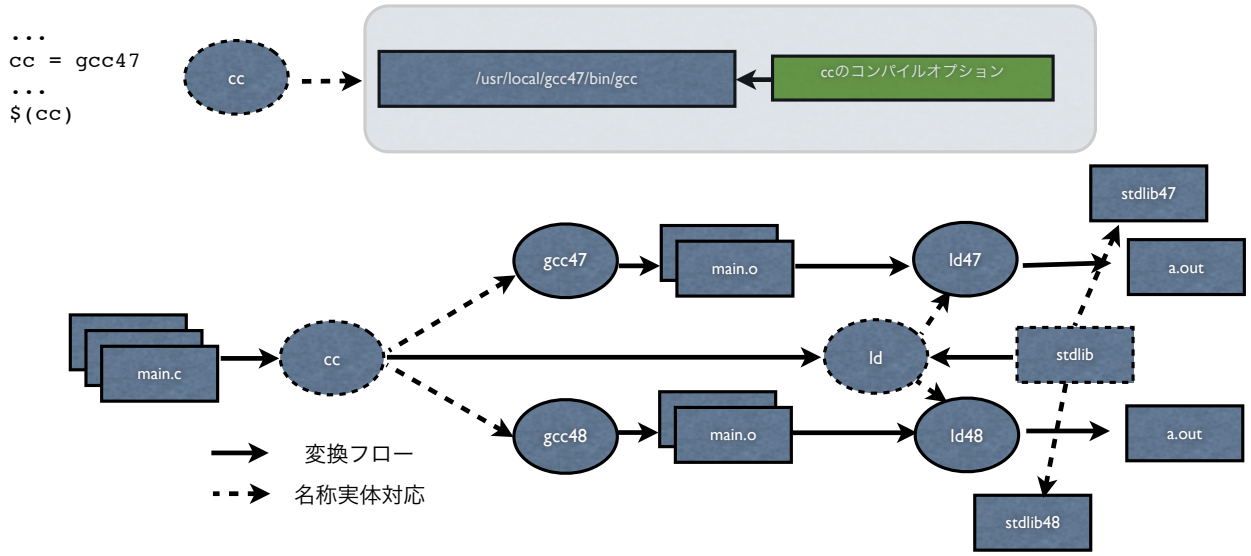


図 12　名称実体対応が含まれる 2 つの環境

3.3. 実行時に必要か環境要素

プログラムの実行時に必要な要素として以下のものを定義する。

(1) ランタイム

ランタイムとは、すべてのプログラムにおいて実行される際に必要とされる処理を行うための処理系機能の一部である。具体的にはC言語ではメモリの初期化（大域変数のクリア)、スタックの初期化、main()関数を呼び出す際に引数の準備と main()の戻り値を OS に渡す処理を行う。C＋＋では例外処理のための初期化などが加わるが基本的には同じである。

(2) 機能別アプリケーションフレームワークの構成要素

アプリケーションフレームワークは、機能に応じて提供される動的ライブラリ群の集合体である。

アプリケーションフレームワークの例として、ウェブアプリケーションフレームワークはコマンド実行による実行されるスタンドアローンプログラムと異なり、ウェブアプリケーションなどの実行系では、実行の際に必要とする仕組みが多い。

例えば、Apache Tomcat を使用したウェブアプリケーションでは、Tomcatは以下の処理を行う。

- 入力と出力のデータ定義と変数を用意する。ウェブアプリケーションでは入力は Request, 出力は Response という型で定義され、実体はフレームワーク内部に確保される。
- サーバ側で保持する必要がある情報（ユーザ ID/パスワードなど）を保持するための仕組みであるセッションを実現する。
- 画面の作成と遷移の制御を行う。特にエラーが発生した場合に、必要に応じて画面やデータベースの制御を行う。

フレームワークを利用するアプリケーションが正しく動作するためには、フレームワークで規定されているコンポーネント類の設定情報が正しく(設定情報が正しく設定）設定され、配備されることを前提とする。
例えば DB アプリケーションの場合は、入力検査（validation)、DB コネクションおよび認証情報、表示の仕方（例: apache velocity template)

validator がフレームワーク依存、これらの情報が正しくない場合、意図通り動かなくなる。

ウェブアプリケーションを起動するにあたっては、アプリケーションの場所およびポート番号などの指定、アプリケーション動作を設定する専用の xml ファイルがあり、これらが未指定されたり、情報が正しくなかったりの場合は同じくアプリケーションが正しく動かなくなる。

図 13 に実行時に必要な環境要素を示す。

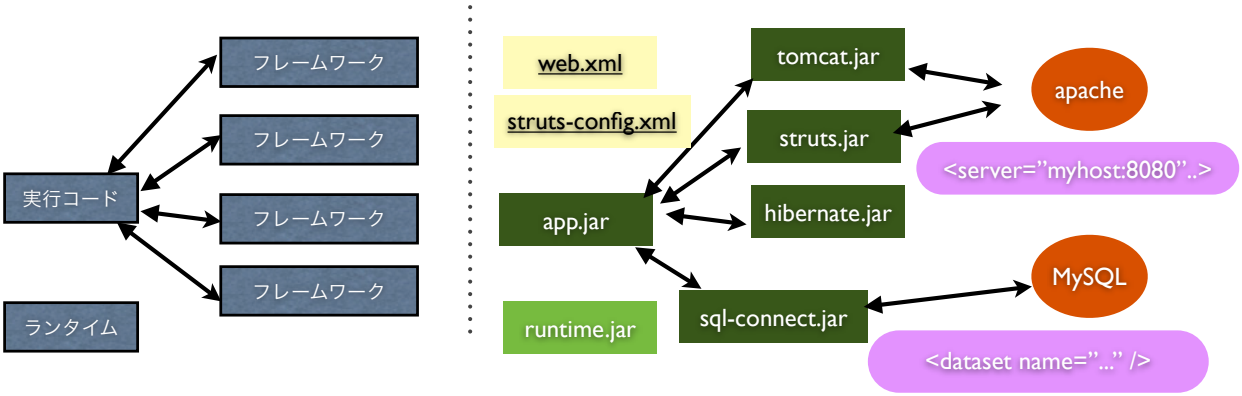


図 13 実行時に必要な環境要素

3.3.1. 実例

以下に、Apache Tomcat 7.0/MySQL 6.5 を使用して、簡単な検索を行うウェブアプリケーションを作成実行する際に必要とされる構成要素を示す。検索用ウェブページと結果ページを別々で設ける。

検索用ウェブページの例(input.jsp)をリスト 7 に示す。

```
<jsp:include page="input.jsp" />
<%
       String title = request.getParameter("title");
       String keyword = request.getParameter("keyword");
       if (title == null) user = "";
       if (keyword == null) pass = "";
%>

<form method="post" action="output.jsp">
<table>
<tr><td>TITLE</td><td><input       name="title"       value="<%=
title %>"></td></tr>
<tr><td>KEYWORD</td><td><input     name="keyword"     value="<%=
keyword %>"></td></tr>
<tr><td></td><td><input type="submit" value=”search"/></td></tr>
</table>
</form>
```

リスト 7　検索用ウェブページの例(input.jsp)

検索結果表示ウェブページの例(output.jsp)をリスト 8 に示す。

```
<jsp:include page="output.jsp" />
<%
       String title = request.getParameter("title");
       String keyword = request.getParameter("keyword");
       if (title == null) title = "";
       if (keyword == null) keyword = "";
       String url="jdbc:mysql://localhost/resvdb";
       String dbuser="guest";
       String dbpass="guest";
       String sql = "";
       Class.forName("com.mysql.jdbc.Driver").newInstance();
       Connection con =
       DriverManager.getConnection(url,dbuser,dbpass);
       con.setReadOnly(true);
       Statement st = con.createStatement();
       if (title.length() > 0 && keyword.length() > 0) {
          sql = "select * from info_table where title like ‘%” +
title + “’ and keyword like ‘%” + keyword + ”%’”);
          ResultSet rs = st.executeQuery(sql);
          while(rs.next()){
          %>
  <tr>
    <td><%=rs.getString("title")%></td>
    <td><%=rs.getString("author")%></td>
    <td><%=rs.getString("published")%></td>
    <td
align="right"><%=objFmt.format(rs.getLong("price"))%></td>
    <td><%=rs.getDate("publishDate")%></td>
  </tr>
          <%
          }
       } else { out.println("<p>Invalid title or keyword</p>");
       }
%></body></html>
```

リスト 8　検索結果表示ウェブページ例(output.jsp)

Tomcat 基盤のウェブアプリケーションの動作を制御するための展開記述用ファイルとして web.xml ファイルを要する。web.xml ファイルの例をリスト 9 に示す。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
    <description>
      DB 検索サンプル
    </description>
    <display-name> DB 検索サンプル</display-name>
    <jsp-config>
        <taglib>
            <taglib-uri>
              http://jakarta.apache.org/tomcat/debug-taglib
            </taglib-uri>
            <taglib-location>
              /WEB-INF/jsp/debug-taglib.tld
            </taglib-location>
         </taglib>
         <taglib>
            <taglib-uri>
              http://jakarta.apache.org/tomcat/examples-taglib
            </taglib-uri>
            <taglib-location>
              /WEB-INF/jsp/example-taglib.tld
            </taglib-location>
         </taglib>
         <taglib>
            <taglib-uri>
              http://jakarta.apache.org/tomcat/jsp2-example-taglib
            </taglib-uri>
            <taglib-location>
              /WEB-INF/jsp2/jsp2-example-taglib.tld
            </taglib-location>
         </taglib>
    </jsp-config>
</web-app>
```

リスト 9　web.xml ファイルの例

上記のリストではDB接続のための認証情報をソースの内部に直接記入したが、JDBC 経由で DB 接続するために JNDI(Java Naming and Directory Interface)を利用する場合、Tomcatサーバ構成設定に使われる server.xml ファイルにJNDIリソースの設定を、web.xmlにJNDIリソースを参照する設定、ソースコード内部でJNDIリソースを呼び出すコードを記載しJDBC接続を行う。

上記の例に対応する環境要素を以下に述べる。

(1) 共通：

操作対象ファイル

input.jsp, output.jsp(入出力画面定義ファイル)

web.xml(ウェブアプリケーションの構成)

server.xml(サーバ構成、外部オブジェクト定義ファイルの例) - DB 接続情報などを含む。

プロセスファイル

Tomcat のアプリケーション作成用手順を記述するプロセスファイル build.xml には表8のようなコンバータの処理を適切な順序に呼び出すための情報が記述されている。

表 8 Tomcat の build.xml 内のコンバータ処理

処理	説明
コンパイル	.jsp ファイルを.java に変換し、さらに.java ファイルから.class ファイルを作成する。
アーカイブ	.class ファイルと各定義ファイル（manifest）を結合して.jar ファイルを作成する。
デプロイ	.jar ファイルを適切な場所に配置することで展開と初期化を行う。
クリーン	作成途中生成されたすべてのファイルを削除する。

ライブラリ

java.lang.＊ ：Javaで基本的に使用されるクラス群の定義を含む。

javax.servlet.＊ ： 処理本体で必要とするデータ構造などが定義されて

いる。

java.sql.* ：データベース操作に必要なデータ構造などが定義されている。build.xml の例をリスト 10 に示す。

```
<?xml version="1.0" ?>
<project name=“test_search” default="deploy" basedir=".">
  <property name="build.dir"  value="." />
  <property name="jar.name"  value="my-examples.war" />
  <property name="target.dir" value="/Tomcat 7.0/webapps"/>
  <target name="compile">
       <javac
    srcdir="${build.dir}/src" destdir="${build.dir}/WEB-INF/classes" >
              <include name="**/*.java" />
       </javac>
  </target>
  <target name="jsp">
     <touch><fileset dir="${build.dir}/jsp" /></touch>
  </target>
  <target name="jar" depends="compile,jsp">
       <jar jarfile="${build.dir}/${jar.name}"
         basedir="${build.dir}" includes="jsp/**,WEB-INF/**" >
       </jar>
  </target>
  <target name="deploy" depends="jar">
    <copy file="${jar.name}" todir="${target.dir}" />
  </target>
  <target name="clean">
    <delete>
       <fileset
         dir="${build.dir}/WEB-INF/classes" includes="**/*.class" />
       <fileset dir="${build.dir}" includes="${jar.name}" />
    </delete>
  </target>
</project>
```

リスト 10　build.xml の例

(2) Tomcat 7.x の場合に使用する構成要素：

コンバータ名称実体対応記述

build.xml 上のコンバータアクションは役割が定義され、実体は定義されないため、この例では省略する。

ライブラリ名称実体対応記述

tomcat7(“tomcat-*.jar = tomcat7.0-*.jar”)

(3) DB として MySQL 6.5 を使う場合使用する構成要素：

ライブラリ名称実体対応記述

mysql6.5(“mysql-connecter.jar =
mysql-connecter-java-6.5*.*-bin.jar”)

(4) フレームワークの実行時に使用する構成要素：

フレームワークを利用するアプリケーションが正しく動作するためには、フレームワークで規定されているコンポーネント類が正しく設定され配備されることを前提とする。例えば DB アプリケーションの場合は、入力検査（validation)、DB コネクションおよび認証情報、表示の仕方（ex: apache velocity template)がフレームワーク依存であり、これらの情報が正しくない場合、アプリケーションは意図通り動作しない可能性がある。

3.4.環境要素の抽出方法

本節では、プログラムの開発環境および実行環境から競合の原因となりうる環境要素を抽出する手順を下記に示す。

3.4.1.プログラム作成時に必要な環境要素

(1)操作対象ファイル

以下の基準の合致するファイルを操作対象ファイルに含める。

- ユーザが作成したソースコードおよび入力ファイル。
- ソースコードからコンバータの(生成元、生成先)関連があるすべてのファイル。

(2)コンバータ

以下の対象をコンバータとして含める。

- コンパイラツールの実体名
- リンカーの実体名
- それ以外開発段階から呼び出されるツールの実体名

以上の項目を取得するにあたって、処理系が規定するツール名を指定してプロセスファイル記述からの検索する方法をとることができる。

(3)ライブラリ

以下の基準に合致するファイルをライブラリファイルに含める。

- ソースコードに明記された import/include 文に対応するライブラリ(例：java.io.File, #include <stdio.h>, #include <sstream>)
- 暗黙的に参照されるライブラリ(例：java.lang.*)
- 上記のライブラリを含むアーカイブファイル(例：libjava.jnilib, library.jar, LIBSUB.dll)

(4)プロセスファイル

以下の基準に合致するファイルをプロセスファイルとする。

- ツールにより規定されているファイル名：
  例）GNUmakefile, makefile, Makefile (make)
  例）build.xml (ant)
- プロセスファイルの生成に関わる選択肢もしくはシステム関連情報

(5) 名称実体対応記述

名称実体対応記述の方法はプラットフォームに依存する。

例）プロセスファイル内に埋め込まれている場合

対応関係として（例：a=b）の形の記述を対象とする。

例）コマンドラインオプションとして指定される場合

コマンドの直前もしくは直後に現れる記述を対象とする。

例）外部で指定する場合

プロセスファイルの動作に影響を与える外部指定（例：環境変数）がある場合はそれらを対象に含める。

3.4.2.実行時に必要な要素

(1) ランタイム

プログラムが実行される環境の情報をランタイムの対象として決められる。Operating System のバージョン、ネイティブアプリケーション実行環境(IA64/Win32, Linux のディストリビューションおよびバージョン、カーネルのバージョンなど)、専用ランタイムの要求されるバージョン(Java 1.5 以上、.NET Framework 3.5 以上など)

(2)アプリケーションフレームワーク

アプリケーションが実行するに依存するすべてのフレームワーク内のファイルを対象とする。フレームワーク別に、名称が固定されている場合と、ユーザの指定によりファイル名が変わる場合その場所について名称実体対応記述を改めて定義できる。

4. 挙動決定情報の定義

本章では、2章および3章で定義した言語要素および環境要素に基づいて、開発環境で発生し得る競合を検出するための手段として挙動決定情報を定義し、競合が発生する際の挙動決定情報の特徴および発生条件との関係について述べる。

4.1. 挙動決定情報の定義

ソフトウェアの開発および実行環境上の要素の中で、意図したプログラムの動作のために関わる言語および環境要素だけを特定する。するとこれらの要素の間には関連がある。

本研究ではプログラムの動作を決定するために必要となる言語要素・環境要素およびその間の関連を**挙動決定情報** (Behavior dominant information) と定義する。

まず操作対象のファイルとしてのソースコードの中に存在する言語要素の間の関連などを明らかにするため、グラフの表現を借りて抽象化を行う。

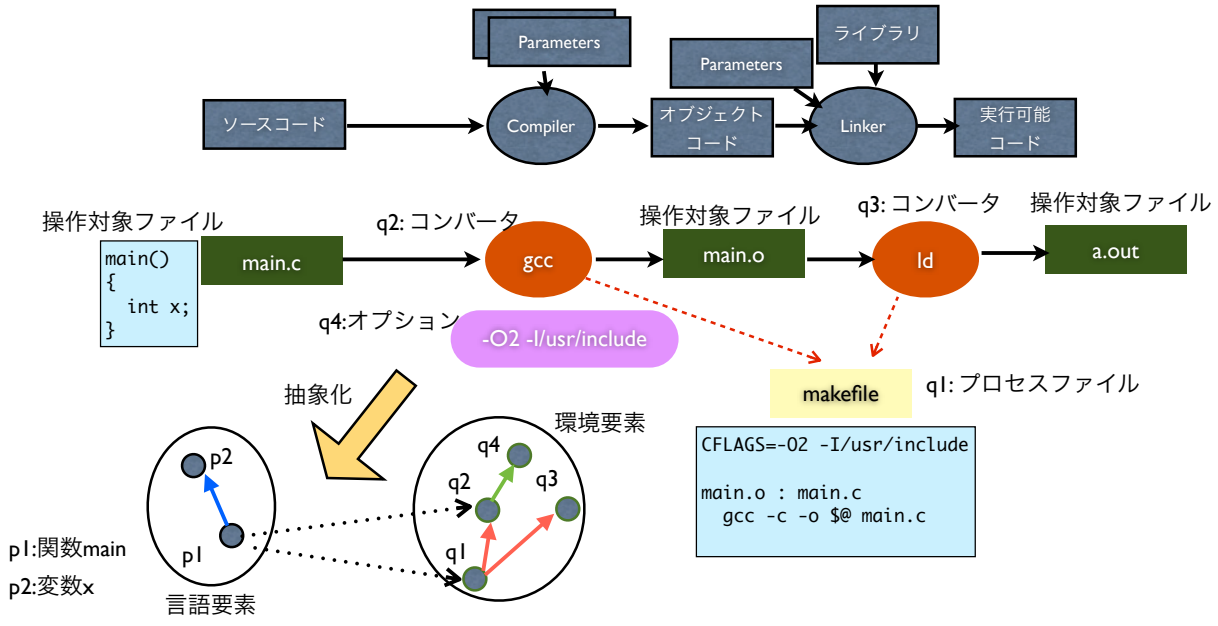


図 14 挙動決定情報の抽象化

図14のmain.cにあるmain関数の中で変数xを参照することがあれば、このようにグラフの２つノードとエッジで表現できる。次は環境要素に注目すると、開発過程の挙動を決めるのはプロセスファイルである。プロセスファイルに開発手順が含まれていて、その中でコンバータおよびオプションが出る。これらの関連をグラフで表記することができる。なお、main()はmain.cに含まれるので、言語要素から環境要素に関連が発生する。

関連の種別は分けて考えられるため、グラフのエッジの色を別々で記述することができる。

以上の内容を数式で表現すると、下記のとおりになる。

$P = \{p_1, p_2, \ldots, p_m\}$　　：言語要素
$Q = \{q_1, q_2, \ldots, q_n\}$　　：環境要素

$Sp = \{(p_i,\ p_j) \mid p_i,\ p_j \in P\}$：要素間関係(言語)
$Sq = \{(q_i,\ q_j) \mid q_i,\ q_j \in Q\}$：要素間関係(環境)

$R = \{(p,\ q) \mid p \in P,\ q \in Q\}$ ： 要素間関係(言語→環境)

$Rs = \{\ R_r \mid\ R_r \subset 2^R\ \}$

B = $(P,\ Q,\ Sp,\ Sq,\ Rs)$ ：　挙動決定情報

挙動決定情報をグラフで抽象化した例を図 15 に表示する。

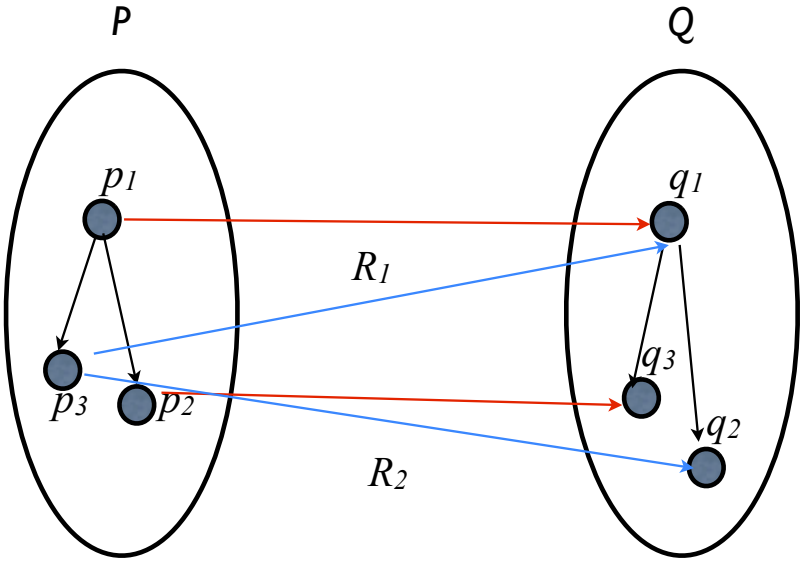


図 15 グラフに表記した挙動決定情報

挙動決定情報に含まれる要素間関係は以下の通りである。

言語要素
　ユーザによる定数・マクロ定義
　ユーザによる変数・クラス定義
　ユーザによる関数・メソッド定義
環境要素
　操作対象ファイル
　プロセスファイル
　構成定義情報
　コンバータ
　　各コンバータに対するオプション
　ライブラリ
　名称実体対応記述
要素間関係
　言語要素と環境要素との関係
　要素の包含関係：要素の内部に別要素を含む場合
　要素の依存関係：要素から別要素を参照する場合
　言語要素の部分集合

4.2. 事例

4.2.1. 単一ソースコードの場合

事例１

例えば C 言語の単一のソースコードで記述されたスタンドアローンアプリケーションを考える（リスト 11）。

```
/*  sum.c */
#include <stdio.h>
int main(int argc, char **argv)
{
        int x;
        int sum, avg;
        while(scanf(“%d”, &x) != EOF)
        {
                sum += x;
                ++cnt;
        }
        printf(“the sum is %d¥n”, sum);
        printf(“the average is %f¥n”, sum/(float)cnt);
}
```

リスト 11　単一コードのアプリケーションの例

リスト 11 のソースは単独でコンパイルされ、その結果は「sum」という名称の実行ファイルとして作成されることとする。上記のソースを同一のソースコードファイルでも利用するライブラリが異なれば実行ファイルの振る舞いも異なる場合がある。例えば通常の printf()を含むライブラリと、組み込みシステムなどで用いられる、%f をサポートしていないバージョンの printf()を含むライブラリがあると仮定する。プログラム sum の入力はテキストファイルで用意し、実行結果をテキストファイルにリダイレクトすることとする。

上記のプログラムの要素は下記の通りである。

言語要素

ソースファイル(sum.c) 内に定義され、エクスポートされる変数および関数

この例ではエクスポートされる関数は main() のみで、変数はない。main()の内部では scanf(), printf()関数を呼び出している。以上の結果より言語要素として scanf(), printf()を対象とする。

環境要素

1)構成情報ファイル

ソースコードが１つしかないので不要。

2)プロセスファイル

プロセスファイルとして Makefile を記述する(リスト 12)。この例では Makefile 内にコンパイルオプションとして最適化レベル(-O0/-O2)を指定する。

```
#Makefile
OPTLVL = 2       # ここを変更すると挙動が変わる可能性あり
 run:   sum
      ./sum < input > output
 sum: sum.o
      gcc sum.o -o sum
 x.o : x.c
      gcc -c sum.c -O$(OPTLVL)
```

リスト 12　Makefile の例

3)ライブラリ

例えば標準ライブラリとメモリなどの制限がある環境で使用される簡易版（実数入出力機能の省略版）の２種類を考える。簡易版では%f も整数として出力されるので実行結果が異なる。

4)入力ファイル

ファイル名を input とする。

```
1 2 4 8
```

5)出力ファイル：

ファイル名を output とする。

標準ライブラリの場合

```
the sum is 15
the average is 3.75000
```

簡易版ライブラリの場合

```
the sum is 15
the average is 3
```

6)名称実体対応記述

標準ライブラリの機能として同一であるが、デバッグ専用の機能として各関数の呼び出しの入口と出口で情報をトレースファイルに記録するライブラリを考える。トレース機能を持たないものをリリース用ライブラリと、持つものをデバッグ用ライブラリと呼ぶ。デバッグ用を使用した場合には以下のようなトレースファイルが作成される。

```
20120112 14:40:11: main called...
20120112 14.40:12: scanf called...
```

7)要素間関係

この例では言語要素としては scanf()/printf()、構成要素としては各ライブラリのみを対象とし、要素間関係を定義することができる。

4.2.2. 構成管理を必要とする場合

１つのプログラムを複数の異なった構成に基づいて作成して実行する場合には、言語要素と環境要素の数が増える。構成管理ツールをコンバータの一種として解釈し、構成定義情報は構成管理ツールのオプションとして挙動決定情報に含めることとする。簡単な例として以下のようなソースファイル群と構成定義を考える。

前提条件

ソースファイル群は機能ごとに分割された複数のファイルを含む。
１つの機能はファイル内に記述された複数の関数により実現されている。
ファイル内には複数の機能が定義されている場合もある。
包含関係／依存関係は関数ごとに定義される。

ここでは機能を３つ（F1, F2, F3)に、各機能を構成する関数を複数用意することとする。

F1 には f11(), f12(), f13()
F2 には f21(), f22()
F3 には f31(), f32() が含まれる。

２種類の実行可能ファイルを作成する作成過程があると考える。過程ごとに必要となる構成定義情報は以下の通りとする。

構成１で必要な関数　f11(), f31()
/ main()から f11(), f31()への呼び出しがある。
構成２で必要な関数　f21(), f31()
/ main()から f21(), f31()への呼び出しがある。

事例 2.1.

すべての機能を実現しているすべての関数に包含関係および依存関係が１つもない場合は、必要機能（言語要素の部分集合）は構成情報定義に記述されたものと同じ。すなわち構成１では[f11(), f31()]、構成２では[f21(), f31()]である。

事例 2.2.

機能 1 に包含関係が 1 つある場合（f11() →[f12(), f13()]）、構成 1 の必要関数は f11(), f12(), f13(), f31()に変更される。構成 2 は変更されない。

事例 2.3.

機能 2 と機能 3 の間に依存関係が 1 つある場合（f21()-> f32()）、構成 2 の必要関数は f21(), f31(), f32()に変更される。構成 1 は変更されない。

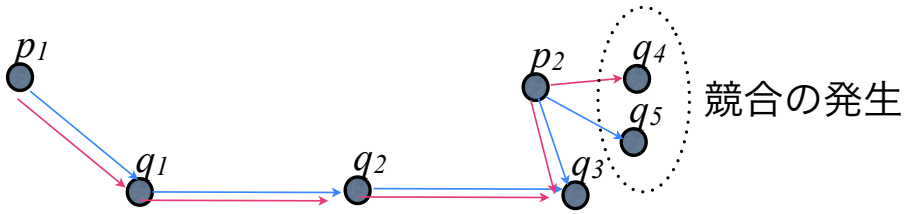
このように構成定義情報に含まれる包含関係および依存関係は、必要ソース群の決定および作成された実行可能コードの振る舞いに影響を与えるため、挙動決定情報として扱うこととする。

4.3. 競合

本節では、言語要素と環境要素、挙動決定情報を用いて競合を再定義する。

競合(conflict)とは、新しい環境の挙動決定情報の集合を現在の環境上に適用した場合に、一貫性が失われることと定義する。これは複数の環境要素を必要とする実行環境（ウェブアプリケーションなど）を部分的に更新した場合に発生しやすい。

競合が発生した状態を表すために、言語要素、環境要素および要素間関係をグラフで記述する。ただし 2 つ以上のグラフが共通するノードを持つ場合、それらは同一の実体でなければならない。これが満たされない場合は競合が発生する(図 16)。



青：意図したもの

赤：実際

図 16 競合発生の例（赤(p2,q4)，青(p2,q5)が競合）

4.3.1. ユーザの意図との不一致

挙動決定情報の一貫性を維持したまま環境の更新を行うことは困難である。不適切な更新により更新の前後で挙動決定情報の間に競合が発生する場合には、アプリケーションの挙動がプログラマーの意図したものと一致しない現象が発生する。例えば 4.2.1 節の事例 1 においてプロセスファイルがリスト 13 のように誤って記述されたとする。

（誤った記述の例）

```
GCC = gcc48
LIB = lib47   # コンパイラとライブラリ間の不整合
run:   sum
     ./sum < input > output
sum: sum.o
     $(GCC) sum.o -o sum -L$(LIB)
sum.o : sum.c
     $(GCC) -c sum.c
```

リスト 13　誤った記述の例

この例では挙動決定情報のうちライブラリを指定する記述(LIB)が lib48 であるべきところが lib47 と誤っているため、挙動がユーザの意図と異なる可能性がある。この例では原因が 1 箇所のため発見と修正も容易であるが、

一般的には複数の挙動決定情報の間で不整合の発見は困難であり、挙動決定情報の一貫性を検査する技術が必要とされる。

複数の挙動決定情報が関連する例として、ユーザ作成ライブラリを含むスタンドアローンアプリケーションを考える。例えば、ファイルを操作するユーティリティの場合、プラットフォームに依存する関数(ファイル名の長さ、使用可能文字の制限などが異なる) をまとめてライブラリ化しておき、ユーティリティ本体からは統一的なインターフェースにより呼び出しを可能とするような構成をとるのが一般的である。このユーティリティを新しい環境上で動作させることを想定する。

名称実体対応記述（外部からの設定）

```
export LIBDIR = /usr/lib/fs2/
```

リスト 14 名称実体対応記述の例

挙動決定情報が正しい場合は、プロセスファイルはリスト 15 の左側のようになるが、この２つの挙動決定情報が右側のように誤っている場合を考える。

```
utility: main.o func.o userlib.a
    gcc -o utility main.o func.o userlib.a
userlib.a : libfunc1.o libfunc2.o
    ar cu userlib.a libfunc1.o libfunc2.o
```

```
#名称実体対応記述
export LIBDIR = /usr/lib/fs2
```

```
utility: main.o func.o userlib.a
    gcc -o utility main.o func.o userlib.a
userlib.a :   # 依存するファイルの記述漏れ
    ar cu userlib.a libfunc1.o libfunc2.o
```

```
#名称実体対応記述
export LIBDIR = /usr/lib/fs1/
```

リスト 15　挙動決定情報の不一致の例

右記の場合、libfunc1.o または libfunc2.o が更新されたとしても

userlib.a は正しく更新されないので、プログラマーの意図と異なる動作をする。さらに、対象となる userlib.a の名称実体対応記述に関して競合が発生しているため、プロセスファイルをすべて正しく修正する必要がある。そのためには、userlib.a に依存するすべての挙動決定情報を抽出するなどの手段が必要となる。

4.3.2. 実行時の競合：ウェブアプリケーションの挙動の誤り

ウェブアプリケーションの挙動決定情報にはフレームワークの固有の情報が含まれる場合が多い。例えば Apache Tomcat では web.xml というファイルに、外部オブジェクト定義や、DB 接続情報などが記述されているため、挙動決定情報に含める必要がある。

事例 1: DB 接続情報の記述方法は Tomcat 5 と Tomcat6 で異なることが判明している。Tomcat 5 での DB 接続情報の記述の例をリスト 16 に示す。

```
<Resource name="jdbc/mysql" auth="Container"
        type="javax.sql.DataSource"/>
<ResourceParams name="jdbc/mysql”>
  <parameter>
    <name>factory</name>
<value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
  </parameter>
  <parameter>
    <name>driverClassName</name>
    <value>org.mysql.Driver</value>
  </parameter>
  <parameter>
    <name>url</name>
    <value>jdbc:mysql://127.0.0.1:3306/testdb</value>
  </parameter>
  ...
</ResourceParams>
```

リスト 16 Tomcat 5 での DB 接続情報の記述の例

一方、Tomcat 6 での DB 接続情報の記述の例をリスト 17 に示す。Tomcat5 とは<Resource>部の記述が異なるため、古い接続情報のままだと DB へ正しく接続できない。

```
<Context    docBase="jspAndServlet2"    path="/jspAndServlet2"
reloadable="true"
source="org.eclipse.jst.jee.server:jspAndServlet2">
       <Resource name="jdbc/mysql" auth="Container"
                type="javax.sql.DataSource"
      driverClassName="org.mysql.Driver"
                url="jdbc:mysql://127.0.0.1:3306/testdb"
            username="user"
             password="pass"
            maxActive="20" maxIdle="10" maxWait="-1"/>
  </Context>
```

リスト 17　Tomcat 6 での DB 接続情報の記述の例

事例 2: DB のパフォーマンスに影響する内部構成定義ファイルは MySQL のバージョン 5.1 とバージョン 5.5 で異なることが判明している (MyISAM →InnoDB に変更)。バージョン 5.1 から 5.5 にアップグレードした際、設定ファイルが古いままだと InnoDB が正しく動作しないため、パフォーマンスの低下を起こすケースもある。また、新バージョンで提供されている性能改善の効果を全く得ることができない問題が発生する。これも競合と考えることができる。

Tomcat と MySQL に関連する挙動決定情報は以下の通りである。以下の情報を正しく記述することで競合の回避が可能である。

- Tomcat(version 6)の DB 接続情報が記述されている設定ファイル(web.xml) の<Resource>欄(リスト 18)。

```
    name="jdbc/mysql”
    auth="Container"
    type="javax.sql.DataSource"
    driverClassName="org.mysql.Driver"
    url="jdbc:mysql://127.0.0.1:3306/testdb"
    username=“user”
    password=“pass”
    maxActive="20"
    maxIdle="10"
    maxWait="-1"
```

リスト 18 Tomcat 6 の DB 接続情報の設定の例

- MySQL の innoDB 関連設定。正しく設定することでパフォーマンスの面で改善できる(リスト 19)。[4]

例)

```
    innodb_buffer_pool_size=4G
    innodb_log_file_size= 1024M
    innodb_flush_log_at_trx_commit=2
    innodb_doublewrite=1
    innodb_flush_method=O_DIRECT
    innodb_thread_concurrency=0
    innodb_max_dirty_pages_pct=80
    innodb_file_format=barracuda
    innodb_file_per_table
    max_connections=50
    table_cache=1024
```

リスト 19　MySQL の innoDB の設定の例

5. ケーススタディ

本章では、実際いくつかの例を考察して、4章で定義した挙動決定情報に基づく、競合の可能性を探る方法について述べる。

図 17　凡例

5.1. C/C++言語のアプリケーション

本節ではC/C++のアプリケーションについて、挙動決定情報を決定し、想定される競合の種類と検出および回避方法について述べる。

5.1.1. 言語要素

C/C++言語では関数を公開するためにはヘッダファイルを利用する。ヘッダファイルに宣言された関数および大域変数を言語要素として含める。

C++の場合クラス定義に対して同様にヘッダファイルを作成する。ただし、publicで定義されるメンバ変数および関数のみを言語要素として含める。

5.1.2. 環境要素

(1) 一般的なcc/gcc系の場合

一般的なUNIX/Linux系のOS環境上でcc(Cコンパイラ)もしくはGNU Compiler Collectionのgcc/g++を使用してC/C++アプリケーションをビルド・実行する手順は図18のように表記できる。

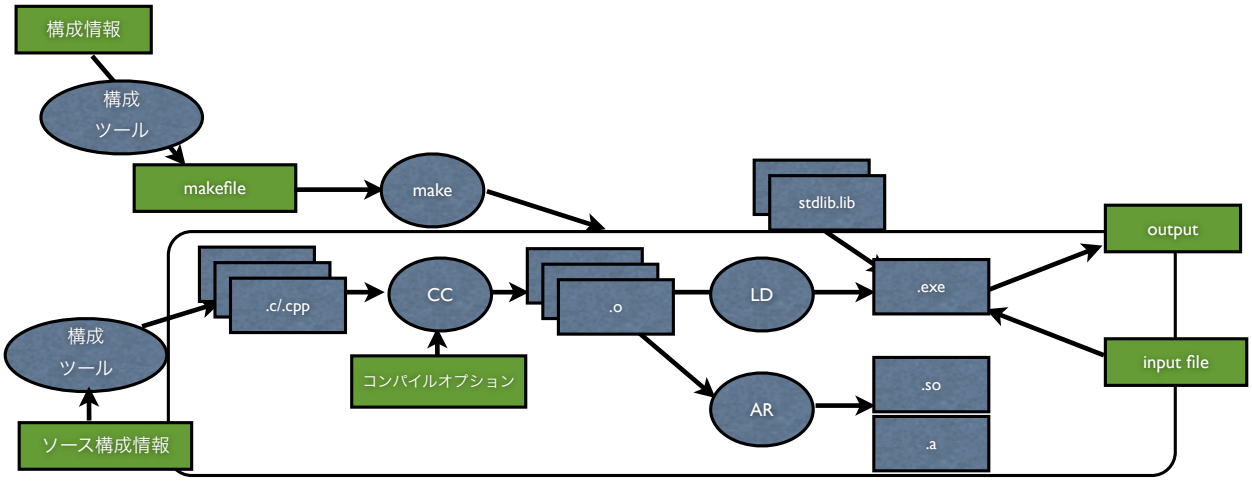


図 18　cc/gcc 系コンパイラツールによるプログラムの作成手順

上図で現れる環境要素を下記にまとめる。

a)ソース構成情報（source configuration info）

全体ソース群から対象のソースを絞り込むための構成情報。ソース構成管理ツールを用いた場合、ツールの種別により設定情報が異なる。

b)構成情報(configuration info)

大規模のソースコード群をコンパイルする際、Makefileは、対象の環境に依存しない作成手順などの記述と、コンパイル時の具体的なオプションおよびライブラリの参照有無、特定のツールの存在有無など対象の環境ごとに異なる定義または制約の記述の２つから構成される。環境ごとに異なるオプションの有無などの情報は環境要素として有効である。図19はMakefileを作成するためのconfigureスクリプトの実行例である。

```
$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
checking for ranlib... ranlib
（略）
checking where the gettext function comes from... libc
configure: creating ./config.status
config.status: creating Makefile
config.status: creating contrib/Makefile
config.status: creating doc/Makefile
config.status: creating gnulib/lib/Makefile
config.status: creating man/Makefile
config.status: creating po/Makefile.in
config.status: creating src/Makefile
config.status: creating tests/Makefile       Makefile の作成
config.status: creating config.h
config.status: executing depfiles commands    依存関係の作成
config.status: executing po-directories commands
config.status: creating po/POTFILES
config.status: creating po/Makefile
¢
```

図 19 configure スクリプトの実行の例

c) プロセスファイル

プロセスファイルであるMakefileには作成手順として呼び出されるコンバータの種類およびオプション情報がふくまれる。これらの情報を環境要素

として扱うことができる。さらにプロセスファイル内部の記述にb)で抽出した構成定義情報との依存関係がある場合、あわせて環境要素として扱う。

d) コンバータ

makeによりソースのビルド作業が開始する際、その中でコンパイル(CC)・リンク(LD)・アーカイブ(AR)などのコンバータ機能が呼び出される。

C/C++コンパイラおよびリンカーのツールは環境要素に含める。コンバータのコマンド名を直接指定せずに抽象的な名前によって指定する場合もあり、この場合は名称実体対応記述を抽出し環境要素に加える。

e) ライブラリ

プロセスファイルによる明示的なライブラリ指定に加えて、 標準ライブラリ(libc)など暗黙的に追加されるライブラリが存在する。ライブラリファイルのバージョンおよび位置情報を環境要素に加える。

f) 操作対象ファイル

コンパイル対象に含まれるソースコードおよび入力ファイルおよび予想される出力結果を環境要素に加える。

(2) Eclipseを使用する場合

Eclipse上でC/C++プログラムを作成する際には、プロジェクトを単位として作成プロセスを管理することが可能である。これらのファイルの内部記述から必要なものを環境要素に加える。

.project

Eclipseプロジェクトファイル。対象ファイルのフォルダ体系、ツールパスなどの情報を持つ。

.cproject

C++専用Eclipse CDTプラグイン用ファイル。コンバータの名称実体対応記述と、プロジェクトごとの作成プロセスの情報を持つ。

以上のファイルの中身から必要な情報を環境要素に加える。

(3) Windows/Visual Studioを使用する場合

図20にWindows上でVisual Studioで作成したC/C++アプリケーションをビルドし実行する手順を示す。

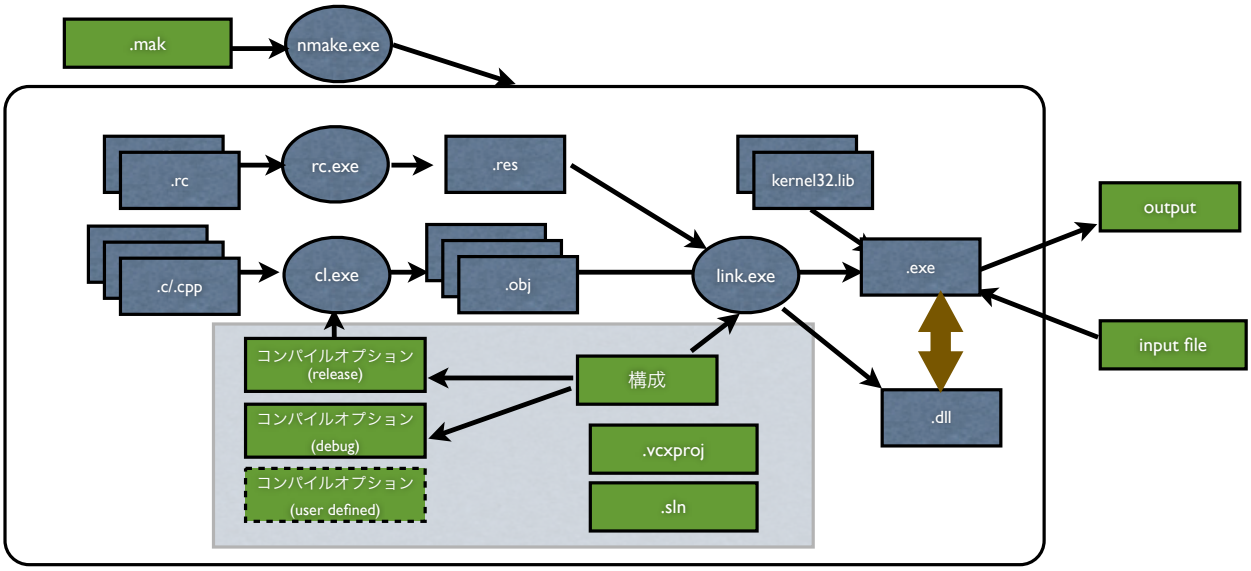


図 20　Windows 上で Visual Studio 使用によるスタンドアローンプログラム作成の流れ

構成情報として含まれるファイル群を下記に記述する。

a) プロジェクト・ソリューションファイル

Visual Studioはビルド対象のソース群およびライブラリ参照先、データベースアクセスのための情報、フォルダおよびファイルの一覧を保つため、ソリューションファイル(*.sln)とプロジェクトファイル(.vcxproj)を持つ。Visual StudioにてIDE上の対象プロジェクトに対するすべての設定変更はこれらのファイルに記録される。

なお、msbuildツールによるソースのビルドの場合、コンパイル時に適用するデフォルト情報が.NET Frameworkごと異なるため、その情報を適切な場所にある特定のファイルも環境要素として扱うべきである。例えば.NET Framework 2.0のランタイムの場合、該当フレームワークのインストール場所（基本的には「C:¥WINDOWS¥Microsoft.NET¥Framework¥v2.0.50727」）にある「MSBuild」フォルダ内の「Microsoft.Build.Commontypes.xsd」というファイルを参照する必要がある。

b）.makファイル

GNU Makeのプロセスファイルである Makefileと同様。.makファイルをプロジェクトで用いる場合、環境要素として扱う。

c）.rcファイル

Win32環境で利用されるリソース指定ファイル。リソースコンパイルにより.resファイルが作成される。

d）コンバータ

通常のコンパイラ(cl.exe)およびリンカー(link.exe)の他にリソースコンパイラ(rc.exe)がある。

5.1.3. 挙動決定情報

表9に挙動決定情報の候補とすべき対象要素の一覧をまとめる。

表 9　C/C++ スタンドアローンプログラムの挙動決定情報の候補

カテゴリ	要素	挙動決定情報に包含有無	対象
言語要素	公開変数	○	ヘッダファイル
	公開関数	○	ヘッダファイル
	定数・マクロ	○	ヘッダファイル・構成定義情報
環境要素	操作対象ファイル	○	ソースコード、入力ファイル
	コンバータの実体	○	パスの直接指定またはシステム環境変数上のパス指定 構成管理項目の選択により実体が変更される。(オプション変更含み)
	コンバータ名称実体対応記述	○	Makefile, *.mak, build.xml, .project, .cproject
	構成管理項目の選択	○	Debug/Release/ユーザ定義
	ライブラリの実体	○	ライブラリパスの直接指定またはシステム環境変数上のパス指定
	ライブラリ名称実体対応記述	○	Makefile, *.mak, build.xml, .project, .cproject 対象フレームワークのデフォルトプロパティ定義ファイル
	構成定義情報	○	Makefile, *.mak, build.xml, *.vcproj .project, .cproject
要素間関係	公開関数→ライブラリ	○	
	定数→ライブラリ	△	#defineは含まれない
	マクロ変数→構成定義情報	○	
	要素の包含・依存関係	○	構成定義情報によりオプションなどが決定。

5.1.4. 競合の種類

1）作成・実行時：原因＝対象ライブラリのバージョンが異なる場合

３章および４章では、ライブラリの指定の誤りにより、意図したのと別のライブラリを参照することで謝った結果を得られる可能性について紹介した。

2）作成時：原因＝コンパイラーバージョンが異なる場合

同一コンパイラでもコンパイラのバージョンが異なる場合、もしくはオプション指定が異なる場合、異なる結果を得る場合がある。

例えば、ANSI CのC89仕様とC99仕様に内部コンパイル時のいくつかの挙動に差異があることが知られている。[5] C89仕様で作成され動作確認済みのソースコードをC99仕様に準拠したコンパイラでコンパイルを行う際、明示的に後方互換性のオプションを指定しないと意図とは異なる結果を得る可能性がある。

5.1.5. 競合の検出・回避方法

前節で挙げた競合の例に対して検出および回避する方法として下記に述べる。

1）作成・実行時：原因＝対象ライブラリのバージョンが異なる場合

挙動決定情報をグラフで表示した場合、比較対象の操作対象ファイルに当たるノードが一致しなかった場合である。

実際想定できるケースとしては、ライブラリの名称とバージョンだけでなく、ターゲットプラットフォームが異なったり、ユーザ定義ライブラリの場合、作成日、ファイルサイズなどの付加情報の照合結果が不一致した場合も考えられる。

そのため、このような競合を正確に検出するためには、対象ライブラリのバージョンたけでなく、想定された対象プラットフォーム、依存する別ライ

ブラリの情報など、ソフトウェア動作に影響を与える情報を挙動決定情報に追加しておく必要がある。

さらに、正しい使用方法に基づいたライブラリ名称とその属性の組み合わせを事前に検出しておく必要がある。この組み合わせの取得対象として下記の例が考えられる。

- ライブラリの名称と位置を記述した名称実体対応記述
- プロセスファイル上のライブラリ実体の記述
- ライブラリのバージョンおよびファイルサイズ、ターゲットプラットフォームなどの情報
- ライブラリで公開された関数および変数の一覧
- 構成定義情報に記述されているライブラリ実体の記述
- 対象ライブラリが依存する別のライブラリ

収集した情報に基づいて競合が発生する組み合わせが存在する場合は、ライブラリと公開関数変数の対応が原因で、かつ変更可能な場合はライブラリの構成を変更することで、競合の回避が可能になる。

2）作成時：原因＝コンパイラーバージョンが異なる場合

挙動決定情報をグラフで表示した場合、比較対象のコンバータのノードが一致しない場合である。

実際はコンバータの名前および位置情報だけでなく、実行時に与えたオプションも挙動決定情報として取り入れなければならない。挙動決定情報としては下記のことが挙げられる。

- コンバータの種類
- コンバータのバージョン
- コンバータに与えたオプション

収集した情報に基づいて競合が発生することがわかった場合、正しいと確証がとれている挙動決定情報の組み合わせに合致しない特定のオプションを使用しないことで、競合の回避が可能になる。

5.2. 構成管理を使用する場合

本節では構成管理機能が加わりそれによって対象ソースコードが選別されたり、実行可能コードの作成方法が変わったり、実行に必要な環境要素が変わったりする場合について、挙動決定情報を決定し、想定できる競合の種類と検出および回避方法について述べる。

5.2.1. 言語要素

構成管理によって公開される関数および大域変数、クラス定義などが言語要素として含める。

5.2.2. 環境要素

構成管理機能によって変わる対象に関わる情報を環境要素として含める。

a) 対象ソースコード群の変更

5.2.1.節で述べたソース構成定義情報がこの範疇に相当する。全体ソース群から対象のソースを絞り込むための構成定義情報は環境要素として含める。

たとえばgitツールの場合、.gitディレクトリの内部に全管理対象ソースから対象になるソース群に関する情報が登録されている。gitツールのcloneオプションなどで必要なソースの特定のバージョンが取得できる。この際、特定のバージョンを指定する場合には指定のための情報を環境要素に追加する。

b) 実行可能コードの作成方法の変更

IDEで管理されるソースコードのプロジェクトは、ビルド構成管理という機能によって実行可能コード生成方法が変わる場合がある。[6]例えばVisual

Studioで作成されたプロジェクトは、デフォルトとしてデバッグ用バイナリ生成モードとリリース用バイナリ生成モードの２つの構成項目を支援する。この項目はユーザによって追加定義が可能であり、選択した構成項目によりコンパイルオプションなどに変更が生じる。従って、対象にする構成項目およびそれにより変更可能性があるオプション、定数定義などを環境要素に追加する。

c）アプリケーションに適用する構成定義項目、定数・マクロの定義

構成管理により管理される複数の構成項目から１つを適用することにより、コンパイル時に適用する定数、マクロなどの変化により異なるコンパイル結果を得る場合もある。定数およびマクロにおいては、言語要素としてソースコードの内部で記述される場合もあるが、環境要素として記述する場合もある。

5.2.3. 挙動決定情報

表10に構成管理を使用する場合の挙動決定情報の候補をまとめる。

表 10　構成管理を使用する場合の挙動決定情報の候補

カテゴリ	要素	挙動決定情報に包含有無	対象
言語要素	定数・マクロ	−	ヘッダファイル・ソースコード・構成定義情報
環境要素	構成管理項目の選択	○	Debug/Release/ユーザ定義
	構成定義情報	○	*.vcprojのプロジェクト構成部分の記述 gitコマンドのオプション
	マクロ変数→構成定義情報	○	

5.2.4. 競合の種類

1）作成時・実行時：複数の構成が混在する場合（デバッグ／リリーズ）

構成項目の変更により、コンパイル時に適用されるオプション群に変更が発生し、結果の実行可能ファイルの動きが変わる場合がある。

例えばVisual StudioでC/C++で作成されたソースコードのプロジェクトでは、デバッグ構成項目を適用してコンパイルを行った場合、デバッグメモリアロケーターが使用され、すべてのメモリ割り当ては、ガードバイトで囲まれる。これにより、メモリの上書きが検出される。これに対してリリース構成項目を適用した場合、このようにならない。さらにリリースとデバッグではヒープメモリーのレイアウトが異なるため、デバッグビルドではメモリの上書きによって問題が生じない場合でも、リリースビルドでは致命的な問題が生じる場合があると知られている。従って異なる構成が混在するオブジェクトコードを静的にまたは動的にリンクした場合、プログラムの動作中に例外によるクラッシュが発生する可能性がある。

2) 作成時／実行時：原因＝使用する文字コード選択の誤り

ソースコード側で扱う文字のエンコードに合致する関数を使用しない場合、実行結果に誤りが発生する可能性がある。

WindowsのWin32環境でC/C++ソースコード作成時、文字列を保持するため標準（マルチバイト用）のchar, ユニコード用のwchar_t, 両方を支援するためのマクロとしてTCHARのような複数のタイプを使用可能である。なお、文字列を扱う一部の関数はマクロとして定義され、Visual Studio上で図21のようにプログラムで使用する文字セットの設定により、マルチバイト用とユニコード用の専用関数の実体への定義が変更になる。例えば、画面上にメッセージボックスを表示するMessageBox()関数は、文字コード設定によりマルチバイト専用のMessageBoxA()またはユニコード専用のMessageBoxW()にマッピングされる。

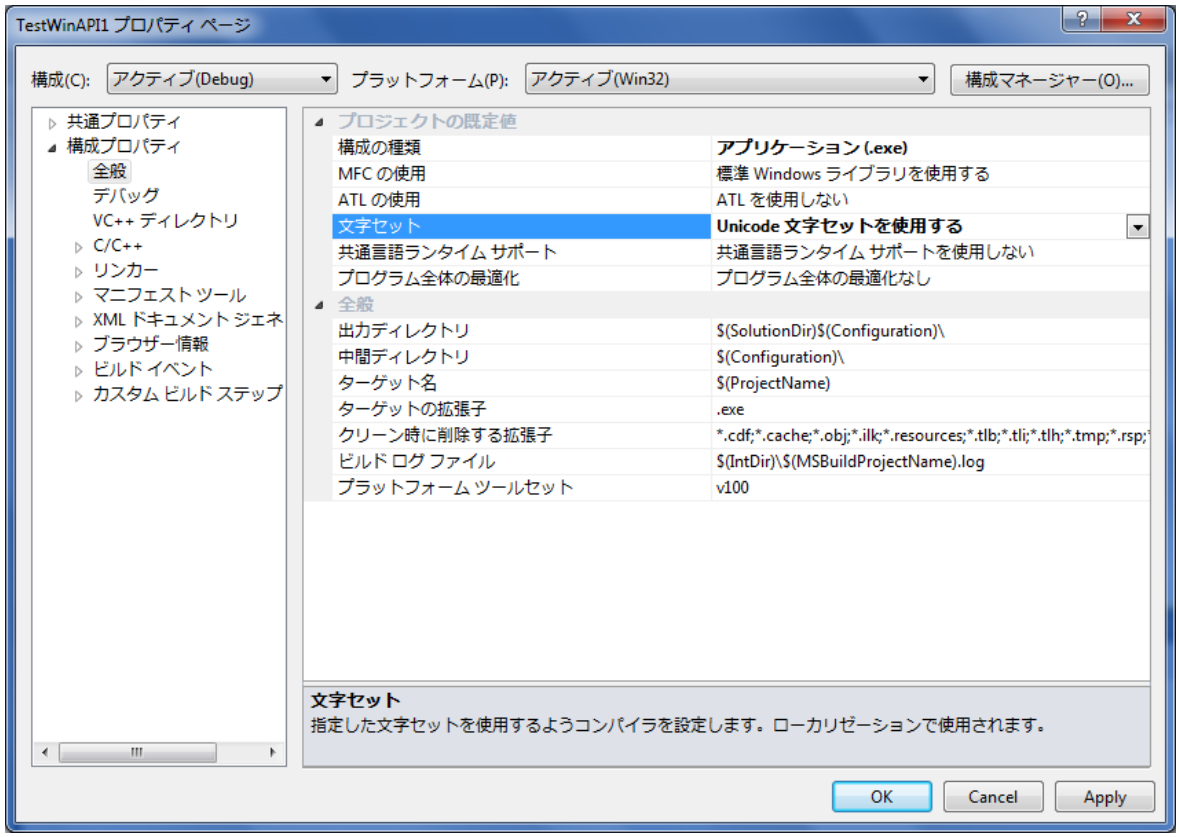


図 21　Visual Studio のプロジェクト上の文字セットの設定

このような特性により、ユーザが作成するプログラムで参照するライブラリの内部で文字列を扱う関数の動作が、環境設定（文字コード）により変わる可能性がある。

5.2.5.　競合の検出・回避方法

1）作成時・実行時：複数の構成が混在する場合（デバッグ／リリーズ）

このような競合を検出するには、対象プロジェクトによって生成された実行可能ファイルおよびライブラリ群に対して、適用された構成項目の種別を挙動決定情報として含める。

回避方法としては、比較対象のソースコード群に適用する構成項目が一致するかを確認することが望ましい。

2）作成時／実行時：原因＝使用する文字コード選択の誤り

対象ソースコードおよび関連するライブラリの作成時の構成定義情報から文字コードが一致するかを確認することにより回避できる。

一般的には、比較対象のソースコード群に適用する構成項目はもちろん、構成項目により選択されるすべての構成定義情報に対してすべて一致するかを確認することになるが、実際の挙動に影響を及ぼさない情報を取り除くのが望ましい。

5.3. 実行時の競合：ウェブアプリケーションフレームワーク

フレームワークを用いて作成されたウェブアプリケーションは、ソースコードのコンパイルだけでなく、実行に関わるフレームワークの挙動に関して様々な設定情報を要する場合が多い。

5.3.1. Java + ウェブアプリケーションフレームワーク

Javaで作成されるアプリケーションは、図22のような過程で実行までたどり着くことが一般的である。

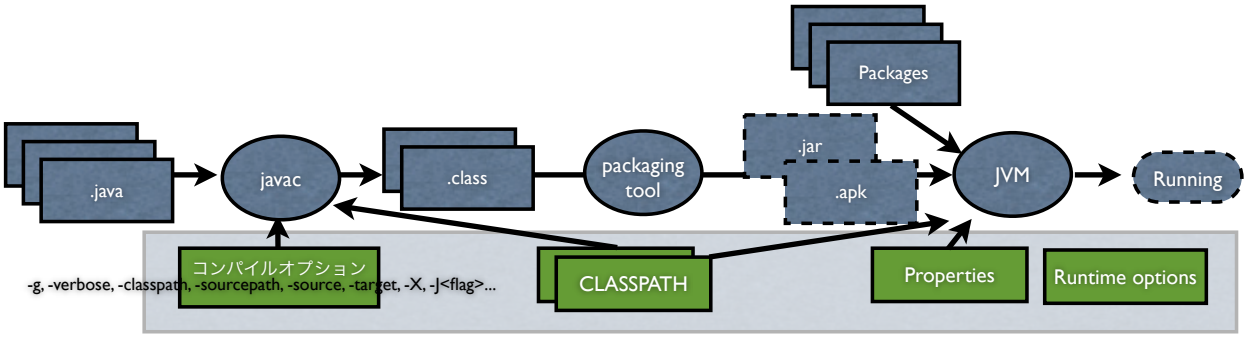


図 22　Java アプリケーションの作成・実行の流れ

この際、表11のような要因が挙動決定情報の候補として挙げられる。

表 11　Javaプログラムの挙動決定情報の候補

カテゴリ	要素	挙動決定情報に包含有無	対象
言語要素	クラスの定義	−	ソースコード
環境要素	CLASSPATH	○	
	実行時のオプション	○	-Xm512, …
	プロパティ	△	
	外部ライブラリの実体	○	Java標準ライブラリ以外
	設定ファイル − ファイル個体 − 名称実体対応記述	○	フレームワークにより異なる

a)コンパイル・実行時のオプション

実行時のオプションにはJVMの起動時のメモリ領域の制御を行うアプションなどの指定ができる。

b)作成時および実行時に参照されるCLASSPATH

クラスの名称に対して定義の実装を探すため参照するパス。パスが正しく指定されていない場合は指定されたクラスに対してまったく別の実装を参照し振る舞いが異なる可能性がある。

c)プロパティ

java.util.Propertiesで保持可能なプロパティ情報。一部プロパティはアプリケーションの振る舞いに変更を与える可能性がある。

Javaアプリケーションとのウェブアプリケーションフレームワークの組み合わせの例としてTomcatとStruts[7]のバージョン2のフレームワークの場合（以下Struts2と略す)、Struts2のアプリケーションは機能別で処理系のアクションを定義する形で作成される。個別のアクションに対してhttp://(サービスのURI)/(機能).actionの形でアクセスでき、アクセスによ

るRequestに対して該当するアクションを探すActionMapperクラスから図23のような流れで処理が実行される。

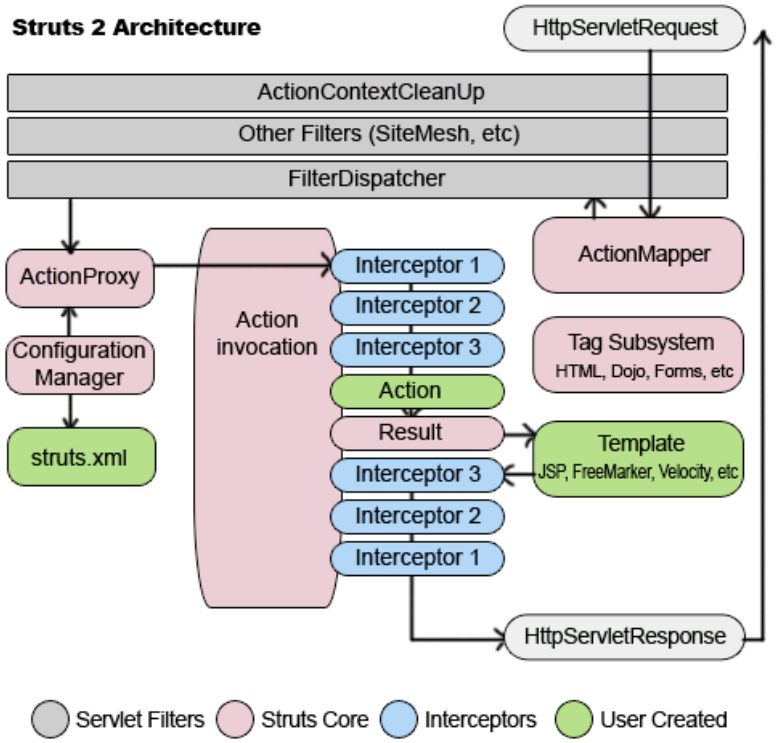


図 23　Struts2 の処理の流れの概要

Struts基盤のアプリケーションに含まれる環境要素として以下のものがある。

warファイル

アプリケーションクラスのアーカイブ。

タグライブラリファイル

.tldファイル。

web.xml

ウェブアプリケーションの設定を記述する配備記述子ファイル。

Tomcatのバージョンによりweb.xmlファイルの位置が異なることが知られている。

struts.xml

Requestが届いたときの動作を規定する設定ファイル。

(アクション)-validation.xml

アクション単位のウェブ検証を設定ファイル。

WEB-INF/lib

.jarのライブラリの配置場所。

WEB-INF/classes

Struts以外のクラスファイル、JSPファイルの配置場所。

この中でweb.xmlとstruts.xml，検証設定用XMLファイルはアプリケーションの動作に関わる設定情報を含むため、実行時に使用するStrutsライブラリのバージョンと一致していない場合、意図した通りに動作しない場合がある。例えばのStruts1.1と1.2には現在のstruts.xmlは旧名称のstruts-config.xmlであり、またforward要素に<contextRelative>という属性があるが、Struts1.3では廃止されている。Struts1.3のDTDを指定している場合、<contextRelative>を記述するとXMLファイルとしては不正になる。

Struts1.2のDTDを指定すれば<contextRelative>をstruts-config.xml上に記述してもエラーにならないが、ライブラリが1.3だと<contextRelative>の処理は削除されているので記述した内容が挙動に反映されない。

これらの設定ファイルを挙動決定情報に追加する必要がある。具体的には、以下の対象が収集の対象になる。

ファイルの位置

ファイルの内部タグ構造

設定値からの名称実体対応記述

フレームワークの別の例として、DBアクセスのためMyBatisフレームワークを利用する場合、リスト20のようなDB接続情報を持つ設定ファイルと、リ

スト21のようなクエリをマッピングするXMLファイルが挙動決定情報の対象になる。

```
<properties resource="org/mybatis/example/config.properties">
  <property name="username" value="dev_user"/>
  <property name="password" value="F2Fa3!33TYyg"/>
</properties>
```

リスト 20 MyBatis の設定ファイルの例

```
<insert id="insertAuthor" parameterType="domain.blog.Author">
  insert into Author (id,username,password,email,bio)
  values (#{id},#{username},#{password},#{email},#{bio})
</insert>

<update id="updateAuthor" parameterType="domain.blog.Author">
  update Author set
    username = #{username},
    password = #{password},
    email = #{email},
    bio = #{bio}
  where id = #{id}
</update>

<delete id="deleteAuthor" parameterType="int">
  delete from Author where id = #{id}
</delete>
```

リスト 21　MyBatis のクエリマッピングファイルの例

ウェブアプリケーションにはクライアント側の振る舞いを実装、制御するJavaScriptのフレームワークを用いる場合、スクリプトのウェブページ上の呼び出しの順番によりウェブページの実行が正しく行われない場合がある。これは複数のフレームワークが同じ名前の特定のキーワードもしくは関数の名称をそれぞれ定義して、同時利用により競合が発生するためである。コードの宣言の順番により同じコードでも別の振る舞いを与える特徴は、コン

パイラでなくソースコードで直接実行されるタイプの言語環境では発生可能性があり、競合の１つとして挙げることができる。この競合を回避するには、挙動決定情報の順序関係を依存関係として見なすか、別の挙動決定情報として扱い、特定の言語要素もしくは環境要素に対する依存関係を別途管理する方法がある。

5.3.2. ASP.NET/C#

ASP.NETの基盤技術である.NET Frameworkは、Common Language Runtime(CLR)基盤の実行環境と共に、Javaに類似したクラスライブラリ群を提供する。この上に位置するASP.NETはウェブのRequest/Responseの処理を内部で保持するページ単位のクラス群、HTMLで画面上に表示される主なUIコンポーネントをカプセル化したクラス群、ユーザのアクションによるイベントをWindowsのイベント処理と類似した形で処理するためのイベント処理系クラス群を提供する。

Visual StudioでASP.NET系ウェブアプリケーションを作成する場合、下記のファイルを挙動決定情報に含めることが可能である。

表 12 ASP.NET の挙動決定情報の候補

カテゴリ	要素	挙動決定情報に包含有無	対象
言語要素	クラスの定義	−	C#/Visual Basicソースコード
環境要素	コンバータ(.NET Frameworkのツール)	○	
	ライブラリの実体	○	アセンブリのバージョン
	設定ファイル - ファイル個体 - 名称実体対応記述	○	Global.asax Web.config 複数の関連設定ファイル

コンバータのバージョン

.NET Frameworkはバージョンによってクラス群が追加された形で発展してきており、バージョンの変化にコンバータ自体が変更されたことはない。

Global.asax

ウェブアプリケーションレベルのRequest/Responseなどのイベントの基本処理用定義ファイル。イベント対応処理コードが含まれている。旧バージョンのASP技術ではGlobal.asaファイルが同じ役割を持っていた。

Web.config

ウェブアプリケーションの共通設定情報を保持するファイル。DB接続情報などの情報を含む。IIS上に配置されるすべてのウェブアプリケーションに対してこのファイルが存在するため、配置先および中身の変更による競合の可能性がある。すべてのルートのWeb.configファイルは%windir%¥Microsoft.NET¥Framework¥framework_version¥CONFIGにある。

Machine.config

サーバレベルのIIS設定情報

%windir%¥Microsoft.NET¥Framework¥framework_version¥CONFIGにある。

ApplicationHost.config

サーバレベルのIIS設定情報

%windir%¥system32¥inetsrv¥configにある。

これにより複数の設定ファイルの同時使用による設定の抜けなどの競合が発生する可能性がある。

同一情報に対して複数の参照が存在する競合の例として、SP.NETはフォルダ体系としていくつかの既定の名称を持つフォルダが存在する。これらのフ

ォルダにはそれぞれの役割を持つため、正しい場所に所定のファイルを配置しない場合、意図通りの動作結果を得られない場合がある。似ている役割のフォルダが複数存在し、場所選定に混乱を生じることもある(Bin/, App_Code/, アセンブリのキャッシュなど)。このようか競合を回避するには、役割に関連するすべての可能な参照先を挙動決定情報として抽出し管理するなどの選択肢がある。この情報は対象環境の特性に依存するほかその情報量が増える可能性があるため、意味を持つ挙動決定情報だけを絞り込む作業が必要である。

5.4. 事例と挙動決定情報と競合の検出

前節までに示した調査結果に基づき、挙動決定情報とそれを利用した競合の検出および回避方法をまとめる。

本研究では、言語要素として公開変数、公開関数および定数およびマクロを挙げた。言語要素については作成アプリケーションの実行形態に依存せず、これらのすべてを挙動決定情報に含める。

環境要素としては操作対象ファイル、プロセスファイル、コンバータ、各コンバータに対するオプション、ライブラリ、名称実体対応記述などを挙げた。

以上の挙動決定情報の中でスタンドアローンのアプリケーションの場合は、プロセスファイルにほとんどの挙動決定情報が含まれているため、あらかじめ決められた規則に従ってその内容を解析することで必要な情報を抽出することが可能である。

例として Makefile には「変数＝値」というパターンに一致する行が複数含まれることが多いが、この行は名称実体対応記述あるいはコンバータのオプション定義として抽出可能である。

環境によっては複数のプロセスファイルを必要とする場合もあり、これらの中では他のプロセスファイルを逐次呼び出す場合と、階層的に構成されているプロセスファイルを呼び出す場合がある。この場合の挙動決定情報はプ

ロセスファイル毎独立に定義されるか、あるいはプロセスファイルの種類により情報が分散される。

前者の例としてはMakefileの中で別のMakefileによるmake呼び出しが挙げられる。後者の例としてはEclipseの.project/.cprojectおよびVisual Studioのソリューションファイルとプロジェクトファイルが挙げられる。

その他アプリケーション作成に関わるコンバータおよびライブラリのバージョンなどの属性の食い違いなどの要因により発生し得る競合も存在する。

一方、実行時の別環境との連携を行うアプリケーションの場合、それぞれの別環境を正しく使用するための設定値を保持する定義ファイル(外部環境構成定義ファイル)を必要とする場合が多い。このときの挙動決定情報としては、ファイルから検出できる構成定義情報の記述形式、および設定ファイルの配置先(操作対象ファイル)などが重要である。

例として、ウェブアプリケーションフレームワークのTomcat, Struts, DB接続を実現するための専用フレームワーク(myBatis)、その他 Hibernate, Spring、ASP.NETなどが挙げられる。

要素間関係としては言語要素と環境要素との関係、要素間の包含関係および依存関係を考慮する。ライブラリの参照パスを環境変数・オプションで指定できる環境(JavaのCLASSPATH)、もしくはライブラリもしくはフレームワークの参照の順番により意味の変動が起こりやすい環境(JavaScript系のウェブフレームワーク、ソースコードから直接実行可能な言語環境)において要素間関係を挙動決定情報に含める。

以上をまとめたものを表13に示す。この結果に基づいて競合が発生するかどうかを判定するためのチェックリストを作成することができる。この中で特定のアプリケーションドメインに関わる有効な挙動決定情報だけ絞り込むことも可能である。

表13の縦軸には4章で定義した動決定情報の一覧を、横軸には本章でケーススタディを行った環境を記述する。四角の中の◎と○は挙動決定情報が競合の要因との関係が強いことを(◎＞○)、△は限定的な関係が存在することを示す。-は挙動決定情報と挙動の要因が見つからないことを示す。

表 13　挙動決定情報のまとめ

	挙動決定情報	言語および環境				
		C/C++	Java	.NET/C#/VB	JS	Tomcat/Struts2
言語要素	公開変数	○	○	△	○	−
	公開関数	○	○	△	○	−
	定数・マクロ	△	−	−	−	−
環境要素	操作対象ファイル	−	○	△	−	−
	コンバータ	○	○	△	−	−
	コンバータ名称実体対応記述	○ (makefile, 環境変数)	○ (CLASSPATH)	○ (system env. variable)	−	−
	構成管理項目	○	○	△	−	−
	ライブラリ	○	○(CLASSPATH)	○ (システムパス)	−	○ （システムパス）
	ライブラリ名称実体対応記述	○	○	△	−	○
	構成定義情報	◎	◎	◎	△	◎
	設定ファイル	−	.property	app.config web.config	−	◎
	オプション	○	○	△	−	−
要素間関係	公開関数→ライブラリ	○	○	△	−	−
	定数→ライブラリ	○	△	△	−	−
	マクロ変数→構成定義情報	○	○	○	−	−
	要素の包含・依存関係	△（マクロによる関数の選別）	−	依存関係	−	依存関係 順序関係（※）

（※）順序関係は依存関係の１つとして扱うことができる。

6. おわりに

6.1. まとめ

本研究では、言語要素と環境要素に基づいてプログラムの挙動に影響を与える因子を挙動決定情報として定義し、実際いくつかのケースに対して、挙動決定情報の適用の範囲、および競合の発生および回避の方法に関するケーススタディを行った。

ケーススタディの結果で、挙動決定情報は対象の環境の種類により発生する頻度が異なることがわかった。言語要素と環境要素の関係のずれには明細実体対応記述が増えるが、外部の実行環境を使用する環境ではその環境を正しく動作させるための構成設定情報の重要性が向上する。

6.2. 今後の課題

今後の課題としては、挙動決定情報が複数の関連を持つファイル群に股がって発生する場合についての考察、挙動決定情報の決定方法の環境による依存性についての更なるケーススタディ、挙動決定情報の抽出方法の改良、環境の構築を行う際使用できる挙動決定情報に基づいたチェックリストもしくは環境の自動生成ツールの実装などが挙げられる。

謝辞

本研究の遂行にあたり, 多数の方々からご指導ならびにご協力を頂きました。この場を借りて感謝の意を示します。

まず、本テーマの課題研究を実施するにあたり、主指導教員である鈴木准教授および落水特任教授、青木准教授には、厳しくも暖かいご指導を賜りました。論文指導たけてなく、研究生活全般についてもサホートして頂きました。副テーマをご指導いただいた敷田教授にも心から感謝を申し上げます。学校生活に関してサポートしてくださった教育支援課教務係の皆さんにも感謝致します。

最後に大学院生活を送るにあたり、金銭面・生活面・精神面で支えて続けてくれた家族に心から感謝を申し上げます。

参考文献

[1] Apache Ant, http://ant.apache.org/

[2] Microsoft Visual Studio, http://www.microsoft.com/ja-jp/dev

[3] 今村智, 意外と知らない構成管理の正体 第1回 ファイルバージョンの管理だけで十分ですか？@IT 情報マネージメント、2004/07/27
http://www.atmarkit.co.jp/farc/rensai/config01/config01.html

[4] http://dev.mysql.com/doc/refman/5.0/en/innodb-configuration.html
http://www.facebook.com/note.php?note_id=434698460932)

[5] http://www.velocityreviews.com/forums/t287495-p2-iso-c89-and-iso-c99.html

[6]方法 : デバッグ構成とリリース構成を設定する。MSDN,
http://msdn.microsoft.com/ja-jp/library/vstudio/wx0123s5.aspx

[7] Apache Struts, http://struts.apache.org/2.x/index.html