Prolog による make の実装

笹山琴由　野口真理子　藤本尚子
鴨浩靖　新出尚之
kotoyu@ics.nara-wu.ac.jp
奈良女子大学　情報科学科

概要
make は、依存関係の定義をもとに、ターゲットの更新に必要となる動作を行うツールで、ソフトウェア開発に広く用いられている。make の動作は、依存関係のルールを記述したファイルに基づき、ターゲットを構築するためのルールを見つけ出し、必要なアクションを再帰的に起こすというものである。これは、論理プログラミング言語である Prolog の動作と類似している。そこで我々は、make の Prolog による再実装を行うプロジェクトを進行している。動作の検証を行う対象としては NetBSD のシステム全体の make を選び、ある程度大規模なソフトウェア開発に対してもこのプロジェクトが有効であることを示す。本発表では、我々のプロジェクトの紹介と、現在の進行状況について述べる。

Implementation of make by Prolog

Kotoyu Sasayama, Mariko Noguchi, Naoko Fujimoto, Hiroyasu Kamo and Naoyuki Nide
Nara Women's University, Department of Information and Computer Science

Make is a widely-used tool which executes the required operations to update the target based on the dependency definitions. Make reads a file which describes the dependency rules, then guess a rule to build the target, and recursively executes the necessary actions. This behavior is closely similar to Prolog, a logic programming language. In this view, we are running a project to implement a replacement of make using Prolog. As a target of verification of our product, we choose the building process of the whole system of NetBSD, so that we can show the effectivity of our project on software development of large scale. In this talk, we introduce our project and describe the progress status of the project.

１．はじめに

make は依存関係の定義をもとに、ターゲットの更新に必要となる動作を行うツールである。我々は、make の代替品の Prolog による実装を与える。そもそも make は依存関係がルールの形で書かれており、その依存関係が満たされているかどうかを検査し、満たされていなければコマンドを実行するという動作を再帰的に行っている。つまり知識がルールの形で書かれていて与えた質問が満たされているかどうかをそのルールによって検査し、その検査を再帰的に行う。これは Prolog が質問に答える過程と類似している。このことから、 make の動作を Prolog の後向き推論の機能を用いて置き換え可能なことは容易に推測できる。そしてそれらが実際にそれが可能であることを示した研究は既に存在するが[1][6][8][9][10][11]、Prolog で置き換えて実装してみたものはまだ存在しない。そこで我々は本研究でその実装を行い、現実のソフトウェア開発における make の置き換えに適用することで、その有用性を示す。現在の make の問題点としては、複数の make の実装があり、互換性に問題があること、また、C の#include による依存性の抽出ができないなど、依存関係を扱う能力に一部不十分な点があることなどが挙げられ、それらは我々のプロジェクトにより補えることが期待でき

Implementation of make by Prolog
Kotoyu Sasayama Mariko Noguchi, Naoko Fujimoto, Hiroyasu Kamo and Naoyuki Nide
Nara Women's University, Department of Information and Computer Science

る。今回の実証では、十分な規模を持つ適用対象として、OS (NetBSD)のシステム全体(カーネルおよび標準コマンド)のmake を選んだ。OS として NetBSD を選んだ理由は、NetBSD が移植性を重視して開発されているため、ソースの構成や Makefile などが見通し良く整理されて書かれており、今回の置き換え実験に都合が良かったからである。

我々の提案する新たな make は、既存のプロジェクトで既に用いられている make を置き換えようという目的のものではないが、新たに起こされるプロジェクトで我々の新たな make を採用することにより、従来の make を利用することによる不便さを回避できる利点はあるだろう。

本発表では、我々のプロジェクトの紹介と現在の進行状況について述べる。

2． Make の課題

make は、主にプログラム開発において広く用いられている重要なツールの 1 つである。プログラムを作成する過程におけるファイルの依存関係を、makefile と呼ばれるルール記述ファイルに記述しておき、目的とするプログラムなどのファイルを生成するときに、そのファイルを生成するためのアクションと、前提となるファイルがどれであるかを makefile から調べて、必要なアクションを起こす。

もし、前提となるファイルよりも目的のファイルが新しければ、そのことを認識して、目的のファイルは既に作成済みと判断し、不要な作成アクションは実行しない。また、前提となるファイルを作成するためにさらに必要なファイルがあれば、そのファイルを作成するためのアクションや前提ファイルを再帰的に調べて、必要なアクションを起こしてくれる。このため、目的のプログラムを作成するための多数のソースファイルのうち、一部だけを修正してコンパイルする場合、必要なものだけをコンパイルするといったことができ、開発過程の自動化と効率化に大きく貢献する。従って、これまでプログラムの開発プロジェクトにおいては make が必ずと言っていいほど用いられてきた。

しかし、ソフトウェア開発プロジェクトの大規模化につれ、make では対処が困難な問題も生じるようになってきた。

2.1. make の苦手な領域と make の拡張

ソフトウェア開発の過程で起こる問題において、make ではサポートしにくい領域として、例えば以下のようなものが挙げられる。

1．複数のディレクトリにまたがる構築作業
2．コンパイラオプションの違いによるオブジェクトファイルの管理
3．ヘッダファイル(.h)を取り込むことにより生じる間接的な依存関係の取り扱い

この問題の多くはコンパイラやその他の UNIX システム構成要素の制限により生ずるものである。しかし、巨大なソフトウェア開発プロジェクトの組織構成上の問題と言わざるを得ないものもある。そしてこれらは、単純な自動化ツールで簡単に解決できるものではない。このため、make を拡張するさまざまな試みや、make と併用して make の機能を補うツールの開発などが過去に多数行われてきた。nk、nmake、GNU make、imake、makedepend、shape がその例である。そして、プログラマや研究者の弛まぬ努力によって make は今も発展し続けている。

2.2. 非互換性

しかし、以上に述べたような事情が、新たな問題を生むことにもなった。多数の make の実装が生みだされた結果、複数の make の間で互換性がなく、ある make で利用できる機能が別の make では利用できない、ということが起こってきた。例えば、System V では使用できるが、4.3 BSD では使用できないものに、マクロの文字列置換、ファイル名とディレクトリ名の抜き出し、依存関係行上での括弧を使ったライブラリ用の文法、単一のサフィックスルール、include 命令、相対パスで指定されたコンポーネントの VPATH の探索、などがある。

このため、例えば、プログラムのソースに同梱する makefile が、特定の make にのみ存在する機能を用いている場合、他の種類の make を用いてはコンパイルできない、というようなことが起こる。これは、プログラムの開発と普及の上で障害となることの 1 つである。

３． Make と論理型言語

我々は make を論理プログラミングで再実装することを考えている。また、実装し直すにあたって、2 章に述べたような make のさまざまな問題点について考慮し、これらを解消できるような機能をもつ make の代替品を実現したいと考えている。我々が書き換えの手段に論理プログラミングを用いる理由は、make の働きが論理プログラミングの動作に非常によく似ているためである。本章ではそれについて述べる。

3.1. make の動作

make は自動コンパイルのための手段である。make は makefile というファイルに記述された依存関係を元にファイルの更新を行う。

make の行う処理は大きく以下の 3 つに分けられる。

1. ターゲットを構築するためのルールを探す。
2. ターゲットに依存されているファイルが更新されているかどうかを再帰的に調べる。
3. 記述されているアクションを実行してターゲットを作成する。

以下、簡単な例[3]を挙げて具体的に説明していく。

今、プログラムは 2 つの異なるバージョンを持ち、いつでも好きな方（plot_prompt か plot_win）を作ることが出来るとする。

```
plot_prompt : basic.o  prompt.o
    cc -o plot_prompt basic.o  prompt.o

plot_win : basic.o  window.o
    cc -o  plot_win  basic.o  window.o
```

上の例で plot_prompt か plot_win を初めて作るときには basic.c をコンパイルしなければならない。もし、prompt.c だけを更新して plot_prompt を再構築したい場合、make はそれぞれのファイルのタイムスタンプを調べ、prompt.c だけを再コンパイルしてからリンクする必要があると認識する。

実際$ make plot_prompt とすると、
```
cc -c prompt.c
cc -o plot_prompt basic.o prompt.o
```
のように順にコンパイルされていく。

3.2. 論理型言語

3.2.1. 論理型言語の動作

論理型言語の実行処理の推論過程は以下のような手順で行われる。

1. 与えられたゴールを解くための節を探す
2. 3.節の本体のゴールを再帰的に解く
   (本体に組み込み述語などで何らかのアクションが書いてあればそれを実行)

例えば 3.1 の例に対しては以下のような比較的自然な書き換えが考えられる。

```
make(plot_prompt):-
  make('basic.c'),make('prompt.o'),
  system('cc  -o  plot_prompt  basic.o
  prompt.o').
```

ここで system はシェルコマンドを実行する組み込み述語とする。

また、実際には make はタイムスタンプを参照して、アクションを起こすべきかどうかを決定する。

3.2.2. Prolog を選んだ理由

3.2.1 で既に述べたように、我々は論理プログラミングによる make の実装するにあたり、利用する言語として Prolog を選んだ。これは、Prolog が論理プログラミング言語としてはもっとも標準的な地位を占めていること、そのため最も処理系が普及していると考えられること、などの理由による。

3.3. make と論理型言語

以上に述べたことから、make の動作を論理型言語で置き換えることが自然に出来ると考えられる。、論理我々は型言語の処理系に十分な組み込み述語を加えたもので make を置き換えることを目指している。

更に、現在の複雑な make の規則を論理型言語の文法ですっきり書くことによって、ユーザーによる新たな更新にも対応しやすくなると考えられる。現在の新たな make の要求には make の機能拡張で対応しているため、初期の make と比べようもないほど複雑化してしまっているからである。

４． 実証

次に具体的な書き換え通して実際の Prolog への原理を示す。以下の例は実行ファイル aa を作るための Makefile である。ここでは Hello を出力するだけの単純な作りになっている.

```
/*Makefile*/
aa: a1.o a2.o
cc a1.o a2.o -o aa
a1.o: a1.c aa.h
cc -c a1.c
a2.o: a2.c aa.h
cc -c a2.
```

```
/*a1.c*/
#include "aa.h"
int main()
{
  a();
  return 0;
}
```

```
/**a2.c/
#include <stdio.h>
#include "aa.h"
void a()
{
  printf("Hello\n");
}
```

```
/*aa.h*/
void a();
```

```
/* -*-Prolog-*- */
/* ルールの記述 */
rule('aa', ['a1.o', 'a2.o'],
   ['cc -s -o aa a1.o a2.o']) :- !.
rule('a1.o', ['a1.c', 'aa.h'],
    ['cc -O2 -c a1.c']) :- !.
rule('a2.o', ['a2.c', 'aa.h'],
   ['cc -O2 -c a2.c']) :- !.
rule(_, [], []).

    /*引数に与えたターゲットを make する。その時ルールを検索し、依存するものを make しておいて、Target がそれらより古ければ make する。*/
make(Target) :-
  rule(Target, SourceList, CmdList),
  makelist(SourceList),
  make(Target, SourceList, CmdList).

    /*引数にターゲットのリストを与えてそれらを make する*/
makelist([]).
makelist([Target|TargetRest]) :-
     make(Target),makelist(TargetRest).

    /* ターゲットが存在しかつ依存リストより新しければ何もせず成功し、そうでなければルールに書かれたコマンドを実行する*/
make(Target, SourceList, _) :-
     access_file(Target, none),
     uptodate(Target, SourceList), !.
make(_, _, CmdList) :-  execmd(CmdList).

  /* ターゲットが依存リストより新しいかどうかを調べて、Target が Source より新しければ真となる */
uptodate(_, []).
uptodate(Target, [Source|SourceRest]) :-
     newer(Target, Source),
     uptodate(Target, SourceRest).

newer(Target, Source) :-
       time_file(Target, TargetMtime),
       time_file(Source, SourceMtime),
       TargetMtime >= SourceMtime.

  /* コマンドのリストを与えてそれを実行する*/
execmd([]).
execmd([Cmd|CmdRest]) :-
     write(Cmd), nl,shell(Cmd),execmd(CmdRest).
```

ただ、本来の make はルールに書かれていないと（例えば a1.c が存在しないとき）その時点で失敗するが、このプログラムでは停止してしまう。

５． NetBSD

5.1. NetBSD を選んだ理由

我々が構築中の Prolog による make の置き換えの検証対象として、我々は NetBSD のシステム全体の make という実例を選んだ。

NetBSD は、フリーな UNIX システムの 1 つである。特色としては、最も多数のアーキテクチャをサポートしていることが挙げられる。1990 年に 4.3BSD Reno から分かれた BSD Net/2 をもとに、そこからさらに分かれた 386BSD のパッチを統合して 1992 年に誕生した。そして USL 訴訟に対処して 4.4BSD Lite 1 を取り込み、完全なフリーの OS となって、以降、サポートされるアーキテクチャの種類を増やしながら現在に至っている。

本稿執筆時点での最新リリース版である NetBSD 2.1 では、17 種類の CPU、48 種類のアーキテクチャに対応しており、さらに、実験的に移植されているアークテクチャも数種類ある。我々が NetBSD のシステムの make を検証対象に選んだのは、フリーソフトウェアとしてソースコードとともに公開されている

ので、ソースを容易に入手できることと、多数のアーキテクチャをサポートし移植性を重視して開発されているために、ソースコード(make ファイルも含め)が読みやすく、動作が把握しやすいため、make の置き換えの動作検証に好都合であったためである。この NetBSD の優れた移植性は、MD（機種依存）部と MI（機種独立）部の徹底した分離によって実現されている。たとえば、CPU 依存のコードは、アーキテクチャ依存のコードから分離され、同一 CPU を使用するすべてのアーキテクチャで共用している。あるいは、イーサネットアダプタのドライバのコードから特定のコントローラを制御するコードが分離され、同種のコントローラであれば接続するバスが異なっても共用できるように設計されている。

5.2. NetBSD の実際の makefile の実装例

ここでは NetSD のシステムの一部である cat コマンドの makefile を例とし、その置き換えに必要な機能を考案する。

```
PROG= cat
.include <bsd.prog.mk>
```

これは cat の makefile である。このように NetBSD では、コマンドの大体の makefile は単に bsd.prog.mk を include しているだけである。bsd.prog.mk の内容の一部は次のようになっている。

```
.if defined(SHAREDSTRINGS)
  CLEANFILES+=strings
  .c.o:
     ${CC} -E ${CFLAGS} ${.IMPSRC} | xstr -c -
       @${CC} ${CFLAGS} -c x.c -o ${.TARGET}
       @rm -f x.c

  .cc.o .cpp.o .cxx.o .C.o:
     ${CXX} -E ${CXXFLAGS} ${.IMPSRC}| xstr -c
  -
       @mv -f x.c x.cc
       @${CXX} ${CXXFLAGS} -c x.cc -o ${.TARGET}
       @rm -f x.cc
.endif
```

.if と.endif の部分は条件分岐である。.c.o や.cc.o .cpp.o .cxx.o .C.o はサフィックスルールの定義である。各サフィックスルールに適応したサフィックスが見つかると依存関係行のルールに従った動作を行う。

例えば

```
${CXX} -E ${CXXFLAGS} ${.IMPSRC}| xstr -c -
```

の部分は ${.IMPSRC} を C++ のオプション -E ${CXXFLAGS}でコンパイルしたものを xstr の標準入力に渡すことを行っている。この部分の Prolog への書き換えは 3.1.2 と同じようにして行うことが出来る。この他に条件コンパイルや変数の値の更新などの機能が必要となる。これについては今後順次検討していく計画である。

６． 今後の課題とまとめ

本発表では、make を論理型言語である prolog で書き直すことを試みた。

現在は NetBSD の make の一部である cat の makefile の再構築を行っている。現段階の実装はまだ完成途中であるが、カーネルについては以下のような作業を行っていく予定である。

1. /usr/share/mk 以下にある makefile のインクルードファイルに対応する Prolog プログラムを書く。

2. カーネルのソースコードの各サブディレクトリにある makefile に対応する Prolog プログラムを書き、置き換える。

3. カーネルのコンパイル作業用ディレクトリを作成し、必要な C ヘッダファイルおよび Makefile をそのディレクトリの下に自動生成するコマンドである config コマンドに手を入れ、makefile のかわりに Prolog プログラムを出力するようにする。

4. config コマンドを除き、C で書かれたソースコードには手を加えない。

７． 関連研究

make の問題点については以前から指摘されており、make を拡張したツールがいろいろ発表されている。その例として、Java 言語による Ant[8]、Perl 言語による makepp[9]などが挙げられる。

しかし、これらはいずれも既存の make と同様、手続き型言語による make の再実装あるいは代替品である。これに対し、make の動作メカニズムが手続き型以外のプログラミングパラダイムと類似していることを指摘し、そのパラダイムのもとで make の自然な実現を図るという、我々と似たアプローチとしては、関数型 make[6]が挙げられる。これは、make を関数型言語により実装したものである。ターゲットの生成規則を、ファイルを返す関数とみなし、ターゲットの方が新しい場合に起こる生成の省略を関数の memoization[7]と見なすことによって make の機能を実現しようとしている。しかし同論文では、make で多用されている機能の 1 つである、サフィックスルールについては扱われていない。これに対して、Prolog による実現では、いくつか用意されているルールに対して順番にパターンマッチを行う。それによって適用可能なルールを適用するという動作を自然に書けるという特徴がある。

８． 参考文献

[1]Masami Hagiya : Prolog Shell ―Prolog with Modality IPSJ SIGNotes SYMbol manipulation Abstract No.026-002, 1983

[2]鴨浩靖:カーネルソースから考える,BSD magazine, ASCII MOOK No.18, 2003

[3]Andrew Oram,Steve Talbott 共著, 矢吹道郎 監訳, 菊池彰 訳: make 改訂版,O’REILLY,1997

[4]Leon Starling,Ehud Shapiro 著, 松田利夫訳 : Prolog の技芸, 共立出版, 1988

[5]The NetBSD Project, http://www.jp.netbsd.org/ja/

[6]:田中哲: 関数型言語 make, http://spa.jssst.or.jp/2003/program/papers/03007.pdf

[7]:Donald Michie: “Memo” functions and machine learning. *Nature*, Vol.218, pp.19-22, April 1968.

[8]The Apache Software Foundation: Apache Ant. http://ant.apache.org.

[9]Gary Holt: Makepp. http://makepp.sourceforge.net/.

[10]Peter Miller: Cook. http://www.canb.auug.org.au/~millerp/README.html

[11]Perforce Software. Jam. http://www.perforce.com/jam/jam.html