2022 年度 卒業論文
Prolog による make の実装　—並列実行の実現—

奈良女子大学 生活環境学部 情報衣環境学科 生活情報通信科学コース 4 回生
新出研究室 19480257 樫原円香

2023 年 2 月

概要

我々は、論理型プログラミング言語の一種である Prolog によって、make と同等のビルドツールを実装する研究を行っている。make と Prolog はどちらも後ろ向き検索を行う点で類似しており、make の動作を自然に Prolog で記述できると考えられる。そこで、Prolog で make を実装しようと、2005 年に Mipl プロジェクトが本学で立ち上がった。Mipl は現在も研究が継続されている。

本研究では、make の-j オプションに相当する機能を Mipl に実装した。

目次

1　はじめに

1983 年、萩谷による「Prolog Shell」[1] で、Prolog による make の実装は理論上可能であると示された。make はプログラムの構築に利用されるビルドツールであり、Makefile と呼ばれるファイルに依存関係等のルールを記述し、それに基づいてコンパイル等を行う。また、Prolog は推論機能を持つ論理型プログラミング言語である。make と Prolog はどちらも後ろ向き検索を行う点で類似しており、make の動作を自然に Prolog で記述できると考えられる。しかし、萩谷の研究以降、そのような実装例がほとんどなかった。

そこで、Prolog で make を実装しようと、2005 年に「Mipl（Make in Prolog）」プロジェクトが奈良女子大学理学部情報科学科（現在の生活環境学部情報衣環境学科生活情報通信科学コース）で立ち上がった。Mipl はデータが失われプロジェクトが中断された時もあったが、2016 年に再び始動し、2023 年の現在、研究が継続されている。[2, 3, 4]

また、Mipl の実装は、Prolog の処理系で広く利用されている SWI-Prolog[5] で行っている。

2　make -j に相当する機能の実装

2.1　make -j オプション

make には「-j」オプションというものがあり、-j の後にジョブ数を指定することで、最大でジョブ数の分だけコマンドを並列で実行し、-j オプションがない場合よりコンパイルの時間を短縮できる。ところが、今までの Mipl には make の-j オプションに相当する機能が存在しておらず、コマンドの並列実行やコンパイルの時間短縮ができない状態だった。

そこで、本研究では make の-j オプションに相当する機能を Mipl に実装した。-j オプションに相当する機能を実装するにあたり、Mipl が「make を Prolog で自然に実装できることを示す」という趣旨のプロジェクトであることを考慮し、make での実装を模倣するのではなく、Prolog での自然な記述で実装する方針を採った。

2.2　make_j 述語

-j オプションに相当する機能は、make_j という述語として実装した。make_j 述語は、make_j(number, process, target). という形をしており、第 1 引数 number に指定したジョブ数で、第 2 引数 process に指定したターゲットの処理を、第 3 引数 target に指定したターゲットに対して並列に実行する。第 2 引数に指定するターゲットの処理はターゲットを作成するコマンドを実行する execmd、ターゲットを作成するコマンドを表示する printcmd、ターゲットをタッチする touchfile のいずれかである。printcmd は make の-n オプション、touchfile は make の-t オプションに対応し、execmd はどのオプションも指定しない場合に対応している。

make_j の大まかな構造は次の通りである。

1. ターゲットを make するとき、依存するものが何かを Prolog のルールに従って調べる
2. それらを make するスレッドを、SWI-Prolog のスレッド機能を使用し、Prolog が起こす
3. 各スレッドが再帰的に make を行い、成功したかどうかを Mipl のデータベースに記述する

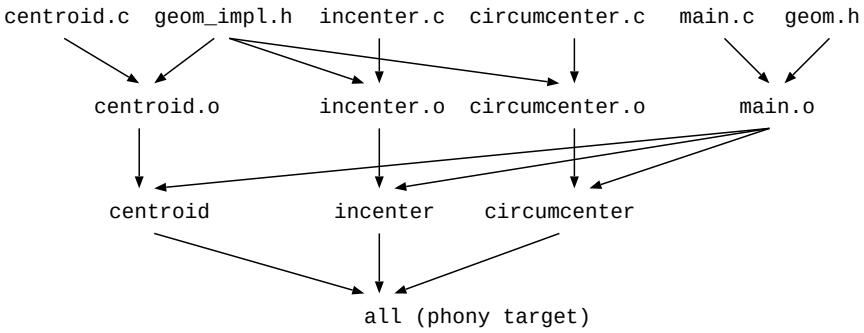

図 1　ファイルの依存関係の例

4. 依存するものの make が全て成功していたら、指定したターゲットの処理を、ターゲットに対して行う
5. 2 を行う中で、ジョブ数の制限を行う onejob 述語によって、指定したジョブ数以内でターゲットの処理を並列に実行する

4 で、依存するものを make するスレッドの終了を待ち合わせるには、SWI-Prolog の thread_wait 述語を用いる。この述語は SWI-Prolog のバージョン 8.4 で追加されたものであるため、Mipl で make_j 述語を利用するには、このバージョン以降の SWI-Prolog を使用する必要がある。また、この述語は busy wait はしないため、これを使用することで性能に悪影響は及ぼさない。

3 では、各スレッドはあらかじめ dynamic 宣言された述語を assert することで、make の成否を Prolog のデータベースに記録する。SWI-Prolog では、dynamic 宣言された述語を各スレッドから変更しても一貫性が保たれることは保証されているため、このような 使い方をすることに支障はない。

2.3　make_j の実行例

make_j 述語の実行例を、既に Mipl に導入されている他の述語と比較しながら、ターゲットの処理ごとに見ていく。また、全てのターゲットの処理において、図 1 のようなファイルの依存関係を例とする。ただし、all は実際のファイルに対応しない phony ターゲットである。

2.3.1　ターゲットの処理が execmd の場合

オブジェクトファイルと実行ファイルが存在しないときに、ターゲット all に対して execmd の処理を行い、オブジェクトファイルと実行ファイルを作成するコマンドを実行する場合の実行例を以下で説明する。Mipl にはターゲットを作成するコマンドを並列化せずに実行する make_1 述語があり、それと make_j 述語とで比較を行う。

図 2 は make_1(all) を実行した結果であり、図 3 は make_1(all) を実行した時のファイルの依存関係である。また、図 4 はジョブ数を 2 に指定した make_j(2, execmd, all) を実行した結果であり、図 5 は make_j(2, execmd, all) を実行した時のファイルの依存関係である。

図 2・図 3 と図 4・図 5 を比較すると、make_1(all) では 1 つのオブジェクトファイルを作成した直後にそのオブジェクトファイルを必要とする実行ファイルを作成するということを繰り返し行っているが、make_j(2, execmd, all) では先にオブジェクトファイルを 2 つ作成し、その後全てのオブジェクトファイルを作成し終わってから実行ファイルの作成を行っている。

```
?- make_1(all).
cc -c centroid.c -o centroid.o (1)
cc -c main.c -o main.o (2)
cc centroid.o main.o -lm -o centroid (3)
cc -c incenter.c -o incenter.o (4)
cc incenter.o main.o -lm -o incenter (5)
cc -c circumcenter.c -o circumcenter.o (6)
cc circumcenter.o main.o -lm -o circumcenter (7)
true.
```

図 2　make_1(all) の実行結果

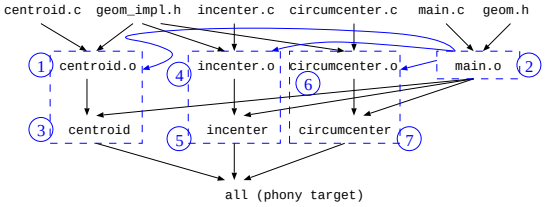

図 3　make_1(all) を実行した時のファイルの依存関係

```
?- make_j(2,execmd,all).
cc -c centroid.c -o centroid.o
cc -c incenter.c -o incenter.o (1)
cc -c main.c -o main.o
cc -c circumcenter.c -o circumcenter.o (2)
cc incenter.o main.o -lm -o incenter
cc centroid.o main.o -lm -o centroid (3)
cc circumcenter.o main.o -lm -o circumcenter (4)
true.
```

図 4　make_j(2, execmd, all) の実行結果

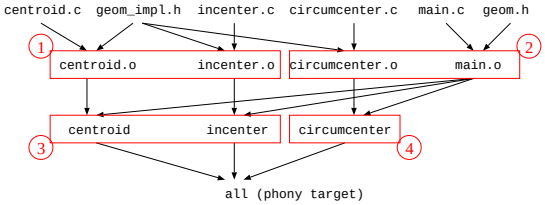

図 5　make_j(2, execmd, all) を実行した時のファイルの依存関係

以上から、並列実行が可能になると、依存関係のないターゲット同士を同時に実行できるようになったことが分かる。

次に、ジョブ数を増やした場合を考える。

```
?- make_j(4,execmd,all).
cc -c main.c -o main.o
cc -c centroid.c -o centroid.o
cc -c incenter.c -o incenter.o (1)
cc -c circumcenter.c -o circumcenter.o
cc centroid.o main.o -lm -o centroid
cc circumcenter.o main.o -lm -o circumcenter (2)
cc incenter.o main.o -lm -o incenter
true.
```

図 6　make_j(4, execmd, all) の実行結果

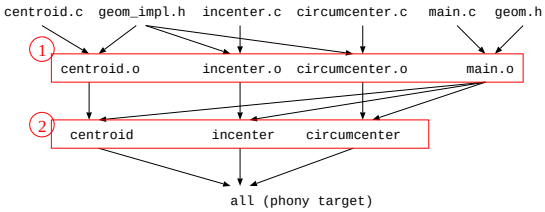

図 7　make_j(4, execmd, all) を実行した時のファイルの依存関係

図 6 は make_j(4, execmd, all) を実行した結果であり、図 7 は make_j(4, execmd, all) を実行した時のファイルの依存関係である。こちらも make_j(2, execmd, all) の実行結果と同様に、先に 4 つ全てのオブジェクトファイルを作成し、その後 3 つ全ての実行ファイルを作成している。

図 8 は make_j(6, execmd, all) を実行した結果であり、図 9 は make_j(6, execmd, all) を実行した時のファイルの依存関係である。この依存関係の例では、1 回目で同時に実行できるターゲットは centroid.o, incenter.o, circumcenter.o, main.o の 4 つであるため、ジョブ数を 5 以上にしても、最初に出力されるコマンドは 4 つとなる。

```
?- make_j(6,execmd,all).
cc -c centroid.c -o centroid.o
cc -c incenter.c -o incenter.o
cc -c circumcenter.c -o circumcenter.o
cc -c main.c -o main.o
cc centroid.o main.o -lm -o centroid
cc incenter.o main.o -lm -o incenter
cc circumcenter.o main.o -lm -o circumcenter
true.
```

図 8　make_j(6, execmd, all) の実行結果

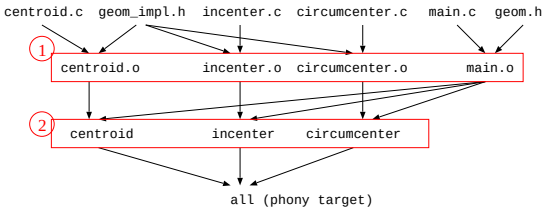

図 9　make_j(6, execmd, all) を実行した時のファイルの依存関係

2.3.2　ターゲットの処理が printcmd の場合

オブジェクトファイルと実行ファイルが存在しないときに、ターゲット all に対して printcmd の処理を行い、オブジェクトファイルと実行ファイルを作成するコマンドを表示する場合の実行例を以下で説明する。Mipl にはターゲットを作成するコマンドを並列化せずに表示する make_n 述語があり、それと make_j 述語とで比較を行う。

図 10 は make_n(all) を実行した結果であり、図 11 は make_n(all) を実行した時のファイルの依存関係である。

```
?- make_n(all).
cc -c centroid.c -o centroid.o
cc -c main.c -o main.o
cc centroid.o main.o -lm -o centroid
cc -c incenter.c -o incenter.o
cc incenter.o main.o -lm -o incenter
cc -c circumcenter.c -o circumcenter.o
cc circumcenter.o main.o -lm -o circumcenter
true.
```

図 10　make_n(all) の実行結果

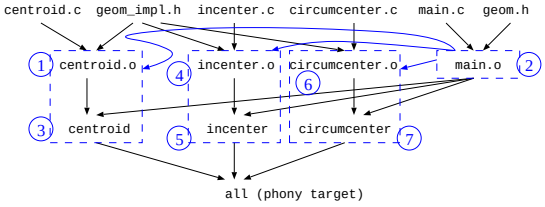

図 11　make_n(all) を実行した時のファイルの依存関係

```
?- make_j(2,printcmd,all).
cc -c main.c -o main.o
cc -c circumcenter.c -o circumcenter.o
cc -c centroid.c -o centroid.o
cc -c incenter.c -o incenter.o
cc centroid.o main.o -lm -o centroid
cc circumcenter.o main.o -lm -o circumcenter
cc incenter.o main.o -lm -o incenter
true.
```

図 12　make_j(2, printcmd, all) の実行結果

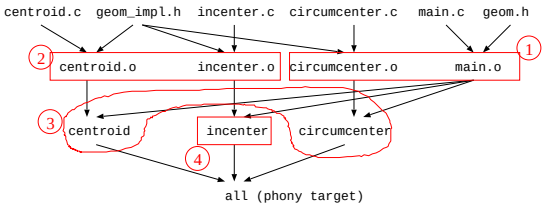

図 13　make_j(2, printcmd, all) を実行した時のファイルの依存関係

また、図 12 はジョブ数を 2 に指定した make_j(2, printcmd, all) を実行した結果であり、図 13 は make_j(2, printcmd, all) を実行した時のファイルの依存関係である。

図 10・図 11 と図 12・図 13 を比較すると、make_n(all) では 1 つのオブジェクトファイルを作成するコマンドを表示した直後にそのオブジェクトファイルを必要とする実行ファイルを作成するコマンドを表示するということを繰り返し行っているが、make_j(2, printcmd, all) では先にオブジェクトファイルを作成するコマ

ンドを 2 つ表示し、その後全てのオブジェクトファイルを作成するコマンドを表示し終わってから実行ファイルを作成するコマンドの表示を行っている。

以上から、execmd の場合と同様に、並列実行が可能になると、依存関係のないターゲット同士を同時に実行できるようになったことが分かる。

2.3.3 ターゲットの処理が touchfile の場合

オブジェクトファイルと実行ファイルが存在しないときに、ターゲット all に対して touchfile の処理を行い、オブジェクトファイルと実行ファイルをタッチする場合の実行例を以下で説明する。Mipl にはファイルを並列化せずにタッチする make_t 述語があり、それと make_j 述語とで比較を行う。

```
?- make_t(all).
touch centroid.o (1)
touch main.o (2)
touch centroid (3)
touch incenter.o (4)
touch incenter (5)
touch circumcenter.o (6)
touch circumcenter (7)
true.
```

図 14　make_t(all) の実行結果

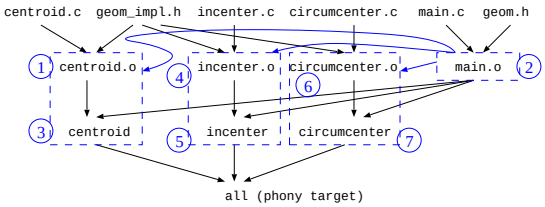

図 15　make_t(all) を実行した時のファイルの依存関係

図 14 は make_t(all) を実行した結果であり、図 15 は make_t(all) を実行した時のファイルの依存関係である。また、図 16 はジョブ数を 2 に指定した make_j(2, touchfile, all) を実行した結果であり、図 17 は make_j(2, touchfile, all) を実行した時のファイルの依存関係である。

```
?- make_j(2,touchfile,all).
touch incenter.o
touch main.o (1)
touch circumcenter.o
touch centroid.o (2)
touch incenter
touch centroid (3)
touch circumcenter (4)
true.
```

図 16　make_j(2, touchfile, all) の実行結果

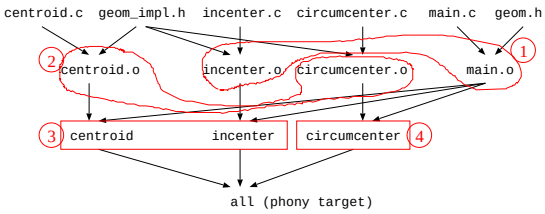

図 17　make_j(2, touchfile, all) を実行した時のファイルの依存関係

図 14・図 15 と図 16・図 17 を比較すると、make_t(all) では 1 つのオブジェクトファイルをタッチした直後にそのオブジェクトファイルを必要とする実行ファイルをタッチするということを繰り返し行っているが、make_j(2, touchfile, all) では先にオブジェクトファイルを 2 つタッチし、その後全てのオブジェクトファイルをタッチし終わってから実行ファイルのタッチを行っている。

以上から、execmd, printcmd の場合と同様に、並列実行が可能になると、依存関係のないターゲット同士を同時に実行できるようになったことが分かる。

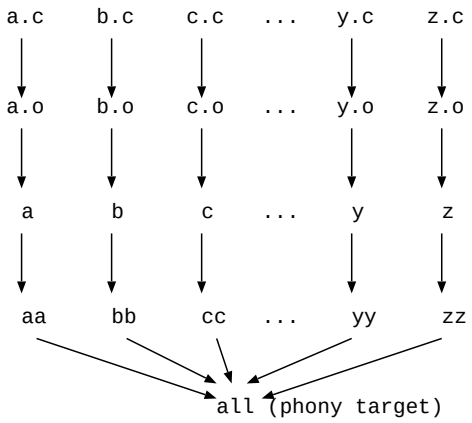

図 18　ファイルの依存関係

3　実験

make_j 述語を使用する場合と、使用しない場合とでターゲットの処理にかかる時間を比較した。

3.1　条件

条件として、1 から 10 億までの数字を順番に足していき、最後に合計を出力する a.c から z.c までの 26 個のソースファイルがあり、それらが図 18 のようなファイルの依存関係を持つ場合に、それら全てのコンパイルおよびコンパイルした実行ファイルの実行を行うターゲット”all”の処理時間を、make_j(number, execmd, all) 述語と、ターゲットを作成するコマンドを並列化せずに実行する make_1 述語とで比較した。make_j 述語を使用する際、ジョブ数を 2,4,8,16,26 の 5 通りに指定し、時間を計測した。今回はソースファイルの数が 26 個であるため、32 ではなく 26 を指定した。

処理時間として real time を計測し、各述語ごとに 5 回計測し、その平均値を採った。また、ターゲット”all”の処理時間はコンパイルの時間だけでは比較にならないため、コンパイルした実行ファイルの実行まで行った。

3.2　実験環境

実験環境は表 1 の通りである。

表 1　計算機の環境

計算機	1	2
OS	macOS 12.6.2	Debian GNU/Linux 11(bullseye)
CPU	Intel Core i5 5350U@1.80GHz	Intel(R) Core(TM) i7-4702MQ CPU@2.20GHz
CPU の個数	2	8
メモリ	8GB	16GB

3.3 結果

結果は表 2 の通りである。

表 2　ターゲット”all”の処理時間 (秒)

述語/計算機環境	1	2
make_1(all).	89.4062	62.5
make_j(2,execmd,all).	46.179	32.1314
make_j(4,execmd,all).	33.5166	18.4088
make_j(8,execmd,all).	32.0542	11.9726
make_j(16,execmd,all).	31.8942	10.5352
make_j(26,execmd,all).	31.755	10.4066

CPU が 2 つの場合、ジョブ数を 2 にすると、直列実行よりも 1.936 倍、つまりおよそ 2 倍早く処理を終えられている。ジョブ数を 4,8,16,26 にしても、それぞれ直列実行よりも 2.668 倍、2.789 倍、2.803 倍、2.815 倍と、およそ 2.5 から 3 倍早く処理を終えられていることが分かる。

一方、CPU が 8 つの場合、こちらもジョブ数を 2 にすると、直列実行よりも 1.945 倍、つまりおよそ 2 倍早く処理を終えられている。ジョブ数を 4 にすると、直列実行よりも 3.395 倍、つまりおよそ 3.5 倍早く処理を終えられている。ジョブ数を 8,16,26 にしても、それぞれ直列実行よりも 5.22 倍、5.932 倍、6.005 倍と、およそ 5 から 6 倍早く処理を終えられていることが分かる。

以上より、make_j 述語を使用すると、直列実行よりもターゲットの処理にかかる時間が短くなることが確認できた。

4　まとめ

本研究では、SWI-Prolog を使用して make の-j オプションに相当する機能である make_j 述語を実装した。Prolog とスレッド機能を用いて記述することで、並列実行しない場合の make の処理と同様に、「ターゲットを make するには、依存するものを並列に make でき、それらが全て成功してかつ、ターゲットを make するためのコマンドが実行できればよい」という、後ろ向き推論の形で自然に記述できた。また、make_j 述語でのターゲットの処理時間を計測し、make_j 述語以外の述語と比較することで、処理時間の短縮を確認した。

今後の課題としては、並列に実行される各ジョブからのターミナル出力が混ざるのを避けることと、あるターゲットがエラーで終了した場合に、他のターゲットがその情報を得ることなく make し続ける状況を避けることが挙げられる。

また、Mipl プロジェクト全体の課題としては、大規模な例を Mipl で記述し、今後必要となる機能が何であるか検証する、などが考えられる。

5 謝辞

本研究及び本論文の執筆にあたり、最後まで丁寧に指導してくださった新出尚之准教授、研究に対してご助言をくださった鴨浩靖准教授に深く感謝の意を表します。

また、Mipl の研究を共に進めてきた鴨研究室の皆様、多くのご助言をくださった新出研究室の皆様に感謝いたします。

付録 A make_j 述語のソースコード

```
/* 指定されたジョブ数でターゲットの処理を並列に実行する */
make_j(N,Target) :-
    make_j(N,execmd,Target).
make_j(N,Do,Target) :-
    retractall(jobNum(_)),
    (between(1,N,N1),assert(jobNum(N1)),fail;true), !,
    retractall(target_state(_,_,_)),
    target_check(Do,Target), !.

target_check(Do,Target) :-
    is_list(Target), !,
    make_parallel_list(Do,Target).
target_check(Do,Target) :-
    make_parallel_list(Do,[Target]).

/* ターゲットのリストを並列に make し、全ての make が終わるまで待つ */
make_parallel_list(_,[]) :- !.
make_parallel_list(Do,TargetList) :-
    all_make_launch(Do,TargetList),
    target_wait(Do,TargetList),
    forall(member(Target,TargetList),target_state(Do,Target,done(_))).

all_make_launch(_,[]) :- !.
all_make_launch(Do,[Target|TargetRest]) :-
    make_launch(Do,Target),
    all_make_launch(Do,TargetRest).

target_wait(Do,TargetList) :-
    thread_wait(forall(member(Target,TargetList),
(target_state(Do,Target,done(_));target_state(Do,Target,failed))),[]),
```

```
    cleanup_terminated_threads.
/* 終了済みスレッドの片付け
   各スレッドはゴールとしてはみんな成功で終わると想定
   そのためスレッドの取得にあたり property として status(true) を指定している */
cleanup_terminated_threads :-
    with_mutex('\0', (thread_property(X, status(true)),
    catch(thread_join(X), _, fail), fail; true)).

/* ターゲットの make がまだ開始されていなかったら新しいスレッドを作り、make を開始 */
make_launch(Do,Target) :-
    with_mutex(Target,(target_state(Do,Target,_) -> true;
       assert(target_state(Do,Target,launched)), fail)), !.
make_launch(Do,Target) :-
    rule_extended(Target,SourceList,CmdList), !,
    assert((target_state(Do,Target,launched))),
    thread_create((make_launch(Do,Target,SourceList,CmdList)),_).
make_launch(Do,Target) :-
    assert((target_state(Do,Target,failed))).

/* SourceList 内の要素を並列に make */
make_launch(Do,Target,SourceList,_) :-
    \+ make_parallel_list(Do,SourceList), !,
    assert((target_state(Do,Target,failed))).
make_launch(Do,Target,SourceList,CmdList) :-
    make_parallel_list(Do,SourceList),
    make_launch_sub(Do,Target,SourceList,CmdList).

/* ターゲットの場合分け */
make_launch_sub(Do,Target,_,CmdList) :-
    phony(Target), !,
    onejob(call(Do,phony,Target,CmdList)),
    get_time(TargetTime),
    assert((target_state(Do,Target,done(TargetTime)))).
make_launch_sub(Do,Target,_,CmdList) :-
    \+ exists_file(Target), !,
    onejob(call(Do,create,Target,CmdList)),
    get_time(TargetTime),
    assert((target_state(Do,Target,done(TargetTime)))).
make_launch_sub(Do,Target,SourceList,_) :-
    time_file(Target,TargetTime),
```

```
    forall(member(Source,SourceList),
           (target_state(Do,Source,done(SourceTime)),TargetTime >= SourceTime)),
    assert((target_state(Do,Target,done(TargetTime)))).
make_launch_sub(Do,Target,_,CmdList) :-
    get_time(TargetTime), !,
    onejob(call(Do,rebuild,Target,CmdList)),
    assert((target_state(Do,Target,done(TargetTime)))).
make_launch_sub(Do,Target,_,CmdList) :-
    \+ onejob(call(Do,_,Target,CmdList)),
    assert((target_state(Do,Target,failed))).

jobNum_max_onejob(Goal) :-
    findall(X,jobNum(X),L),
    max_list(L,M),
    onejob(M,Goal).

/* ジョブの制限 */
onejob(Goal) :-
    repeat,
    (retract(jobNum(N))
    ;
     thread_wait(jobNum(_), []), fail
    ), !,
    (Goal -> X=true; X=fail),
    assert(jobNum(N)), !, X.
```

参考文献

[1] Masami Hagiya. Prolog Shell ---- Prolog with Modality. 情報処理学会研究報告プログラミング (PRO) Vol.1983 Np.48, pp. 1–8, 1983.

[2] 横田晴香. Prolog による make の実装. 2016 年度卒業論文, 奈良女子大学理学部情報科学科, 2017.

[3] 桂春奈. Prolog によるビルドツールの実装. 2018 年度卒業論文, 奈良女子大学生活環境学部情報衣環境学科生活情報通信科学コース, 2019.

[4] 鄭依彤. Prolog による make の実装. 2019 年度卒業論文, 奈良女子大学生活環境学部情報衣環境学科生活情報通信科学コース, 2020.

[5] 著者多数. SWI-Prolog. `https://www.swi-prolog.org`. (最終閲覧: 2023 年 2 月 7 日).