PROM, Version 1.0

User's manual

Thilo Kielmann
Institute of Theoretical Computer Science
Technical University Darmstadt
Germany

kielmann@iti.informatik.th-darmstadt.de

1 Introduction

Maintenance of large, portable software systems often leads to requirements which cannot be solved by the traditional MAKE tool. Abstract naming schemes for files and programs, as they are used by preprocessors in actual MAKE tools, are fundamental for a more general solution. But a preprocessor is not powerful enough for all requirements. Some kind of "database" with the information for making specific files is needed.

Abstract names looking very close to PROLOG terms and the need for a "knowledge base" lead directly to the idea of having a PROLOG interpreter doing the file update task.

This manual describes PROM, a MAKE tool fully implemented in PROLOG. It allows tailoring makefiles for all kinds of software systems. PROM's makefiles (so called *promfiles*) are based on sets of powerful default rules serving the specific needs. Those rules usually contain parts which may be evaluated specifically for every file in use which will be seen later. For a brief description of the concepts behind PROM please see [1, 2].

Although PROM's power depends directly on the rulesets in use, this package unfortunately does not include such sets. If you find PROM useful for your purposes, you are always welcomed to send questions, comments, bug reports, rulesets or any kind of "good idea" directly to the author[1]. Please see the file `LICENSE` for terms under which may use PROM.

2 Calling PROM

While calling PROM from the (UNIX) shell, you have to enter parameters. At least, you have to denote a target to be updated. Besides several targets you may give additional command line options the order of which is arbitrary. Example:

```
prom -n obj(sample) -f mypromfile
```

Here, PROM will use `mypromfile` and print the actions necessary to update `obj(sample)`.

Possible parameters are:

- `-f` <*promfile*>
  Specifies a promfile. Default is `promfile`.
- `-I` <*path*>
  Specifies a searchpath in which promfiles included via `search include` (see below) will be searched. Multiple searchpaths are allowed. They are used in the command line's order.

[1] Comments will also be welcomed even if you are of the opposite opinion...

- `-n`
  No actions will be performed. PROM will only print the actions necessary to update the targets.

- `-d`
  Show dependency graphs for target files. PROM will print a graph for each target which recursively shows all sources on which it depends. The rules involved are also shown. No actions will be performed.

- `-k`
  Show knowledge base after reading the promfile(s). PROM will print all knowledge-base entries created by the promfile in use. No actions will be performed.

- `-help`
  Prints a help message explaining the possible options. No actions will be performed.

The environment variable `PROM` will be evaluated additionally to the command line. Its contents are treated like options entered after the ones of the actual command line. This feature may be used for options repeatedly in use.

Please also note a minor inconvenience based on the behaviour of common UNIX shells. They usually evaluate special characters like the parentheses used in PROLOG terms which are used by PROM for mentioning target files. Please prepare to properly quote terms containing such special characters. You should also take care of filenames conforming syntactically to PROLOG's variable names. They also should be quoted accurately to avoid unexpected results.

The inconvenience with parentheses interpreted by UNIX shells could not be resolved up to date. The kind reader is invited to provide the author with "good ideas" for resolving it. We are currently thinking about a Motif-based user interface which will avoid any shell, but this is subject to future releases...

3 Writing promfiles

3.1 Promfile syntax

The promfile syntax is a compromise between interpretation efficiency and familiarity with the MAKE syntax. The latter is achieved by predefined operators which allow to form terms looking close to traditional makefile entries. These operators are: `create`, `default`, `define`, `depend`, `include`, `search`, and '`:`'.

As a side effect, orientation on PROLOG terms gives more freedom in writing promfiles. Whitespace characters can be inserted for clarity of structure. They are no longer crucial like newlines and tabs in traditional makefiles.

Figure 1 shows the promfile syntax. It is a context-free grammar, known from PROLOG, with terminal symbols enclosed in brackets. The start symbol is `entry`. As a convention of PROLOG's term reading facility, each term has to be completed by a dot ('.').

3.2 Promfile semantics

PROM processes promfile entries of the types listed below.

1. `define` <*term*> `=` <*terms*>`.`
   With `define`, the expansion of a term is specified and entered into the knowledge base. Terms are expanded to lists of terms.

```
entry --> [define], term(Term), [=], terms(Expansion).
entry --> [default], [define], term(Term), [=], terms(Expansion).
entry --> [depend], term(Target), [:], terms(Sources).
entry --> [create], term(Target), [:], terms(Sources),
          [-->], actions(Actions).
entry --> [include], term(Filename).
entry --> [search], [include], term(Filename).

actions(A)           --> action(A).
actions([A1|Others]) --> action(A1), [','], actions(Others).
action(A) --> term(A), { A =.. [call|_] }.

terms(T)           --> term(T).
terms([T1|Others]) --> term(T1), [','], terms(Others).
term(T) --> constant(T).
term(T) --> structure(T).

constant(C) --> [C], { atom(C) ; var(C) }.
structure(S) --> [S], { S =.. [Functor|Arguments],
                        Functor \== ',', Arguments \== [] }.
```

Figure 1: Promfile Syntax

2. `default define` <*term*> `=` <*terms*>`.`
   This definition will only be entered into the knowledge base, if there does not already exist (on interpretation time) any other definition for a term matching the new definition. This feature is useful for defaults in system promfiles which will only be used if there is no corresponding definition in application promfiles.

3. `depend` <*target*> `:` <*sources*>`.`
   With `depend`, one declares that a file called the "target file", depends on a list of other files. These files are treated as sources, independent from the creation rule chosen for the target file. The target file's sources are the source files mentioned in the creation rule united with the ones from all `depend` declarations matching the target file name.

4. `create` <*target*> `:` <*sources*> `-->` <*actions*>`.`
   `create` declares a creation rule for a target file. It implicitly declares a set of source files for this target which will be used if this rule is triggered. The list of actions declares which programs will be executed using this rule.

   Actions must be terms in the form of `call(...,...)` with arity greater or equal to one.

5. `include` <*filename*>`.`
   The content of a file will be inserted in this promfile.

6. `search include` <*filename*>`.`
   Like `include`, but the file will be searched through a predefined search path. Using this feature, the appropriate system promfile for the target machine may be included.

There are three special features which can be used in `define` entries. They are based on capabilities of the term expansion facility of PROM.

1. concatenation
   The '+' operator concatenates its arguments to one single atom.

2. external evaluation
   The functor `eval` interprets its structure components as an external program call. The output of the program called is taken as term expansion.

3. empty expansion
   A very special term expansion feature is its behaviour in the case of an undefined term. In general, "undefined" means the impossibilty of matching the term with a knowledge base entry. Undefined terms will be expanded to nothing. This feature is only applied to terms forming structures. Atoms which cannot be matched remain unchanged.

   For example, one may use the term `compiler_flag(`*program*`)` which will only be expanded for programs in need of such a flag.

3.3 A complete example

In the following, principles of promfile writing are treated on example. First, we consider a file containing rules for creating executable binaries from C source code. It resides in a search-path directory.

```
Rules
create exe(File) : objlist(File)
      --> call( cc_cmd, link_opts, objlist(File), '-o', exe(File) ).

create obj(File) : c(File), eval( get_includes, c(File) )
      --> call( cc_cmd, cc_opts, cc_flags(obj(File) ),
                 '-c', c(File), '-o', obj(File) ).
```

Assume, programs should be developed in parallel for SunOS and for transputers using the Meiko Computing Surface. So there exist two system directories, each for one operating system. In these directories reside files defining system-specific defaults.

```
sunos/Defaults
search include 'Rules'.
define cc_cmd = 'gcc'.
define c(File)   = File + '.c'.
define obj(File) = File + '.o'.
define exe(File) = File.
```

```
meiko/Defaults
search include 'Rules'.
define cc_cmd = 'mcc'.
define c(File)   = File + '.c'.
define obj(File) = File + '.o-tr'.
define exe(File) = File + '.tr8'.
```

All files mentioned so far are application-independent and may be used for several projects. The examples show definitions of program names and file naming conventions, the latter using uninstantiated variables. The rules contain system- and module-specific optional parts which will only be used if necessary (remember "empty expansion"). The creation rule for `obj()` files shows how an external program (`get_includes`) is used for dynamically adding source files to the file-dependency graph.

Now consider a small example-application consisting of two modules: A kernel and a user interface. The user interface needs a special include path. The linker shall use static libraries. The promfile needed consists of the following four lines.

makefile
search include 'Defaults'.
define objlist('application') = obj('kernel'), obj('interface').
define cc_flags(obj('interface')) = '-I/usr/include/X11'.
define link_opts = '-Bstatic'.

These four lines are sufficient for compilation for both architectures, because the appropriate defaults are chosen while starting PROM. If PROM is started to make `exe('application')` for the SunOS platform the following actions will be performed if `exe('application')` is not "up to date".

performed actions
gcc -c kernel.c -o kernel.o
gcc -I/usr/include/X11 -c interface.c -o interface.o
gcc -Bstatic kernel.o interface.o -o application

4 Installation

PROM has been designed as a stand-alone tool. It's current implementation is based on SWI-Prolog which is freely available under terms similar to the ones concerning PROM.

4.1 Installation for using SWI-Prolog

If you like to install PROM based on SWI-Prolog, please perform the following actions:

1. Create an executable file for SWI-Prolog on your machine. Although success in this point is very likely, it belongs to your own responsibility to get SWI-Prolog compiled for your specific machine. To make porting easier, you need *not* port the `unexec` feature to your machine. So you may set `O_STORE_PROGRAM` to `0` in your machine description file. You succeeded if you were able to execute `make swipl` in the source code directory of SWI-Prolog without errors.
2. Change your actual directory to the one containing PROM's source code.
3. Edit the `Makefile`. Setup the following macros to fulfil the needs of your system:
   - `SWIPLDIR` — the directory in which you made `swipl`
   - `BINDIR` — the directory in which the startup shell script for SWI-Prolog (`pl`) shall reside.
   - `LIBDIR` — the directory in which executables for SWI-Prolog and PROM shall reside. This may be identical to `BINDIR`.
4. Edit the startup shell script `pl`. Set the variable `LIBDIR` to the value you also entered in the `Makefile`.
5. Enter: `make install`. If it succeeds, PROM is ready to use.

4.2 Installation for using your favourite PROLOG

If you wish to use PROM based on your favourite PROLOG it is on your own responsibility to get all parts working together. Nevertheless, we provide you with the following information about PROM's intrinsics.

- There exist two sets of PROLOG sources for use of PROM in "batch" mode and interactively (for testing purposes). They only differ in the startup file.
  - "Batch" mode:
    `startbatch.pl debug.pl expand.pl main.pl make.pl scanner.pl`
  - Interactive mode:
    `startup.pl debug.pl expand.pl main.pl make.pl scanner.pl`
- The goal for starting prom is `run/2` (of file `main.pl`).

  It's first parameter is a list of options. Valid options will be automatically created by the startup program in `prom.c`. They are processed by the predicate `opt/1` (of file `main.pl`) where valid options can easily be seen.

  The second parameter is a list of targets where each target is represented by a term.

  For example, the startup call `prom -n obj(sample) -f mypromfile` will result in the call `run([noaction,(file,mypromfile)],[obj(sample)])`.
- PROM's source code uses the following predicates of SWI-Prolog, which might be unavailable or different in your favourite PROLOG.
  - PROM's internal knowledge base is maintained (for runtime efficiency) using the predicates `recorda/2`, `recordz/2`, `recorded/2`, `recorded/3`, and `erase/1`. They may easily be replaced by the standard predicates `asserta/1`, `assertz/1`, and `retract/1`.
  - External programs are called using `shell/2`, where the second parameter is used to receive the program's exit status.
  - PROM uses the following predicates provided by SWI-Prolog instead of which you should use equivalent ones of your PROLOG system. This concerns:
    * `exists_file/1` for testing a file's existence
    * `access_file/2` for testing if a file may be accessed for reading
    * `time_file/2` for retrieving file timestamps
  - PROM reads promfiles using SWI-Prolog's stream-based I/O. If your PROLOG does not support this, you have to work around here, too.
  - `concat_atom/2` is used for creating atoms out of lists of atoms. You may need a work-around, too.
  - Finally, `gensym/2` is used to create unique rule numbers. Maybe, you have to write your own number generator.

References

[1] Thilo Kielmann. `PROM`*: A flexible,* `PROLOG`*-based* `make` *tool.* Report TI-4/91, Institute of Theoretical Computer Science, Technical University Darmstadt, 1991.

[2] Thilo Kielmann. Using PROLOG for Software System Maintenance. In *Proceedings of the First International Conference on The Practical Application of Prolog*, volume 1, London, UK, April 1992.