`prom`: A flexible, Prolog-based `make` tool

Thilo Kielmann

TECHNISCHE HOCHSCHULE DARMSTADT

Report TI-4/91   October 1991
Institute of Theoretical Computer Science

Abstract

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, as shown in this paper, 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 job. As a prototype, `prom` has been implemented which is introduced at the end of this paper.

Introduction

The classical UNIX tool `make`, as introduced in [Fel79], is commonly used for compiling a program which consists of a couple of source files. Furthermore, all of the files are normally located in a single directory and are dedicated to be compiled on one single machine. For this kind of application, `make` has sufficient power.

But with growing software systems and with the high connectivity of computers, `make` reaches its limitations whereas the principle idea of having a program to update files by a set of relations between them is still powerful enough. Because of this reasons an improved `make` tool was designed and implemented.

The requirements of large, portable software systems

The special requirements of large software systems originate in their size. This kind of program is too large to fit into a single directory following the intention of grouping only several related files to keep the overview.

In designing a software system, one usually divides it into several abstract components, like classes of objects or modules. We consider classes instead of single design components because we need multiple implementations for different target machines or with different behaviours. As general term, we concern modules as a concept to support information hiding as defined in [Goo73]:

> "Modularity denotes the ability to combine arbitrary program modules into larger modules without knowledge of the construction of the modules."

In the following, we deal with so-called "module classes". They are defined as the set of all possible implementations which fulfil the specification of a module. Although these sets have an infinite number of elements, `make` tools only treat those files which represent a module class in form of its interface specification and its existing implementations. A brief description of a class hierarchy for software development can be found in [Sch89].

Module classes may be divided recurrently. This leads to a "consisting relation" between the components of a software system describing which component consists of which ones. Figure 0.1 shows an example of a software system consisting of sample module classes.

Under the guideline of information hiding, the module classes are separated and interact only using their interfaces. An obvious way of separating module classes is to transform the graph of the consisting relation directly into a tree of directories. Each directory contains exactly the modules which build its module class. Each module class is accompanied by its makefile, in which the relations between its files are collected. So we can change the implementation of one part of our software system without affecting the others.

As we consider our software system to be portable, we also want to "port" our makefiles with a minimum of changes. A first step towards this aim is done by `make` with the provided macros. So we do not need to take care of names and command line options of the standard compiler tools. But this is not sufficient. What we need is a complete abstraction from command line syntax and file naming conventions. The only information in a makefile should describe relations between the involved files. Therefore we need a notation in the makefiles which allows us to mention files completely abstracted from the names required by the underlying operating system. This notation is the key for keeping multiple versions of a software system in a single directory tree. An example is a UNIX machine with the directory tree of a software system on one of its file systems. This file system may be visible (for example via NFS) on several other UNIX machines (like SYSTEM V, BSD, SUNOS, AIX,...). For assembling on all different machines we only need to support a specific mapping from the abstract notation in our makefiles to the machine-dependent names and syntactical conventions.

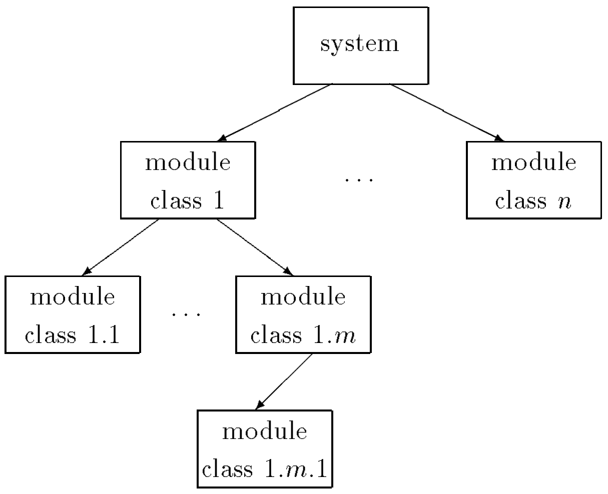


Figure 0.1: The module class hierarchy of a software system.

This also works with more heterogeneous operating systems, for example AIX and MS-DOS which may share a file system via DOS MERGE. And even if there are target machines “off line” from the source directory tree, we can assemble programs using this method. We only need to transport the source files (including the makefiles) initially.

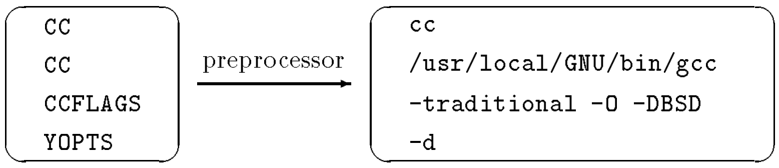
As we have seen, a `make` tool should supply an abstracted view on the components of a software system which hides the system-dependent platform. In the next section a first approach using a preprocessing tool is described.

Using a preprocessor

In the following, we consider `cake` as a `make` tool which uses the C preprocessor for its input files. `cake` provides us with several additional features like transitive and dynamic rules. But the main advantage arises from the C preprocessor, whose macro facilities are the key to the abstracted notation mentioned above. For a brief description of `cake` and its use for portable software systems see [Som87] and [Kie90].

Qualitative improvements with a preprocessor

In the following, we will see how to use the C preprocessor to abstract from system dependent notations. First, we can use simple macros similar to those known from `make`. The following figure shows some examples. For instance, using one of the first two lines allows to hide the C-Compiler currently in use.



```
CC           cc
CC           /usr/local/GNU/bin/gcc
CCFLAGS      -traditional -O -DBSD
YOPTS        -d
```

Parameterized macros are much more powerful. They allow to classify files. In the following, we speak of file classes—the sets of files denoted by these parameterized macros. Some very common examples are:

$$\begin{tikzcd} \boxed{\begin{array}{l}\texttt{C(}\mathit{file}\texttt{)}\\ \texttt{OBJ(}\mathit{file}\texttt{)}\\ \texttt{EXE(}\mathit{program}\texttt{)}\\ \texttt{VER(}\mathit{file}\texttt{)}\end{array}} \arrow[r, "\text{preprocessor}"] & \boxed{\begin{array}{l}\mathit{file}\texttt{.c}\\ \mathit{file}\texttt{.o}\\ \mathit{program}\\ \mathit{file}\texttt{,v}\end{array}} \end{tikzcd}$$

While exclusively using these names, one can easily state relations between the files in use without mentioning any machine-dependent names. Looking at the example of the network global file system from which one source code should be assembled for several different target machines, name conflicts must be resolved. For this purpose, we only need to provide the C preprocessor with the right macro definitions. For example, we define `OBJ(`*file*`)` as *file*`.o-sun3`, *file*`.o-sun4` or *file*`.o-hp`, depending on the target machine.

Such C preprocessor macros are only helpful if definition and utilization is separated in different files. In principle, there are two sets of files in use:

- On one hand, there are counterparts to traditional makefiles located in the source code directories. Here, the information is placed concerning the files realizing the corresponding module class. Sub-module classes are also declared here.
- On the other hand, there are files containing the mapping from abstract (e.g. file class) names to system-dependent syntactical forms. These files are located in a so-called "system directory". Such a directory must exist for each target machine. The appropriate one will be selected when starting `cake` by specifying a search path for the C preprocessor.

In addition to the file classification we also have to describe the relations between the files. These relations declare the creation of files out of each other. Examples are:

```
LINK( target, list of object files, options, libs )
MAKE_ARCHIVE( archive, list of objects )
```

Those relations are used as part of file creation rules, as they are known from `make`. Here, the relations form the actions which have to be performed when the rule is triggered. The actions are written as program calls. As they are system-dependent, their declaration also occurs in the system directories together with the declaration of the rules.

Problems which cannot be solved by a preprocessor

On an idealized view, all information needed to create a program out of its sources is localized in the system directories. This works well, as long as there are only "default" actions to be performed. So the special cases have to be expressed in the source file directories. But this causes problems which can be seen from the following examples. We consider file generation rules in the syntax known from `make` like

```
target : list of sources
         list of actions
```

The most obvious "exception" from default rules is the creation of an executable out of object files. Their number is unknown and their names are usually completely different (independent) from the name of the executable. A simple rule for this could be:

```
EXE(program) : OBJ(one) OBJ(two) OBJ(three)
        LINK( EXE(program), OBJ(one) OBJ(two) OBJ(three) )
```

This rule cannot be stated as a default for the reasons mentioned above. To overcome this, one might propose the following modification:

```
EXE(program) : OBJLIST(program)
        LINK( EXE(program), OBJLIST(program) )
```

Using this rule, one has to declare `OBJLIST` for each executable in a source directory makefile. But this is impossible, because the preprocessor avoids the redefinition of its macro `OBJLIST` from the second definition on. This is because a preprocessor interprets such a definition as a rule for pattern substitution instead of a "Prolog fact" like declaration.

One could also think of forming individual macro names for each executable. This looks like

```
EXE(program) : OBJLIST_program
        LINK( EXE(program), OBJLIST_program )
and
define OBJLIST_example OBJ(one) OBJ(two) OBJ(three)
```

But this approach fails because of the two-step evaluation. First, the preprocessor transforms the rule into:

```
program : OBJLIST_program
        cc -o program OBJLIST_program
```

So the intended match between `OBJLIST_program` and `OBJLIST_example` cannot be performed afterwards in the update process of the `make` tool.

This phenomenon also affects other file classes. For the reasons stated, it is impossible to set compiler switches for individual files only. Every time a file has special requirements, one has to copy the entire set of rules for all target machines.

But that kind of information was intended to be located in the system directories. This violates information hiding between the software system and the specialties of the target machines and leads to severe update problems for the makefiles with growing size of software systems and the number of supported machines.

Using a Prolog interpreter as a `make` tool

From the experience with the previous approach we conclude that entries in makefiles should have declarative nature. These declarations should be entered into a knowledge base. Together with a powerful set of default rules, this knowledge base is an adequate basis on which file updates can be performed.

The abstract notation for files, programs, and actions which was introduced together with the preprocessor concept looks very close to Prolog-like terms. The terms inside the default rules have to be unified with actual filenames. Prolog's term unification facility is an excellent way to do this. Unbound variables in arbitrary positions of terms provide us with almost any degree of flexibility in formulating file creation rules.

Feasibility of implementing `make` with Prolog

The reasons mentioned above suggest Prolog as a predestined candidate for the implementation of a `make` tool. A closer look at the demands of such a tool will show that Prolog is indeed powerful enough to perform all necessary tasks. The complete set of requirements for updating files out of each other is the following:

- Read files.
  For reading makefiles Prolog offers the predicates `see` and `read`.
- Compute the actions to be performed.
  A kind of "update algorithm" can be formulated as Prolog predicates.
- Execute external programs.
  For this purpose, the standard predicate `system` can be used. The Prolog interpreter should take the exit code of a program to succeed or fail with `system`. If it does not take the exit code, one has to ensure otherwise that defective files will always be deleted.
- Get information about timestamps of files.
  This is the only problem which has to be solved outside the Prolog functionality. Possible solutions are (in order of increasing execution speed):

  1. An external program (called by `system` predicate) writes the timestamps into a file which can be read by the Prolog interpreter.
  2. An external program delivers a return value which is interpreted as a timestamp. But interpretation of such a return value must be supported by the Prolog interpreter.
  3. A builtin predicate `timestamp` will be added into the Prolog interpreter. This requires the source code of the Prolog system.
  4. The Prolog interpreter may offer an interface for executing C code as part of the Prolog program.

A Prolog-`make`'s knowledge base

The core of a Prolog-based `make` is its knowledge base. It is created out of makefiles and of timestamp information of the files constituting an application. Figure 0.2 shows an overview of the information involved in the update process which is also described in the following text. The names of the file classes used in the figure only serve as examples without profound meaning.

One major part of information in the knowledge base is extracted from two external sources:

- system makefiles
  Located in a so-called system directory, they declare terms to map the abstract notation onto the requirements of the target machine. They form the system-specific part of the knowledge base.
- application makefiles
  Located in the source code directories, they declare terms concerning files of the specific application. They form the application-specific part of the knowledge base.

Applying the update algorithm, the status information is collected. It consists of file attributes like existence and update time. The algorithm operates in two phases:

- The first phase creates a file-dependency graph out of the knowledge base. Unlike the consisting-graph of the application, this graph is acyclic but in general no tree.

- The second phase applies creation rules beginning with files without successors in the dependency graph, using its topological sorting.

The knowledge base is constituted by the following PROLOG predicates which represent the information mentioned above:

- `create`
  File creation rules, like the ones known from traditional `make`, consist of a target file, a list of source files, and a list of actions to create the target out of the sources.
- `depend`
  With this predicate, dependencies between files are declared. This is needed if additional source files which are not mentioned in a creation rule are used. An example is a header file included by a C program.
- `define`
  These definitions describe the expansion of terms.
- `exists`
  This predicate succeeds if a file exists.
- `timestamp`
  This predicate represents the relation between files and their update times.
- `isok`
  This predicate succeeds if a file is up to date in the sense of the makefiles in use. It is needed, because the file-dependency graph is not a tree. Every file will be marked as `isok` after its creation. So intermediate files are created only once per update process.

The realization of `prom`

The concepts lead to the implementation of `prom`. We start its description with the definition of the syntax of makefiles interpreted by `prom`.

Syntax of makefiles

Unexpectedly, the makefile format became a critical design feature. In a first approach, the syntax chosen was very close to those files interpreted by traditional `make` or `cake`. But this leads to severe runtime problems because of the character-oriented interpretation. The second version, as introduced in the following, was based on PROLOG terms. It can be easily interpreted by the standard predicate `read`.

The 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 makefiles. White-space characters can be inserted for clarity of structure. They are no longer crucial like newlines and tabs in traditional makefiles.

Figure 0.3 shows the makefile 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 `read`, each term has to be completed by a dot ('.').

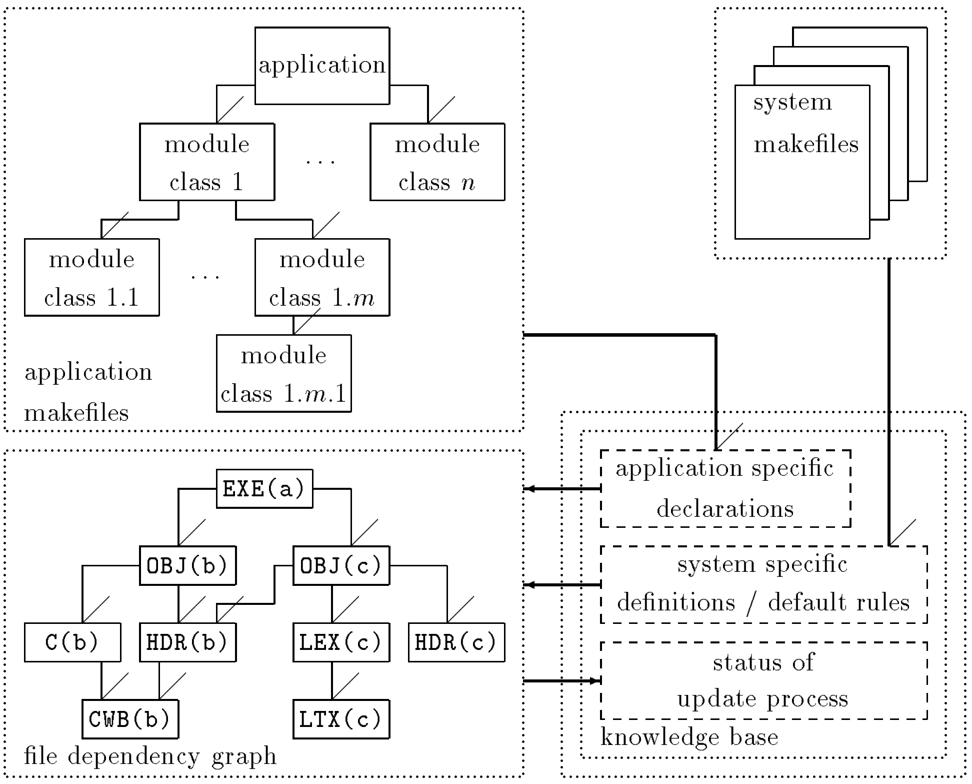


Figure 0.2: Information involved in the update process with sample application makefiles and file-dependency graph.

Semantics of makefiles

`prom` processes makefile entries of the types listed below.

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

2. `default define`:
   This definition will only be entered into the knowledge base, if there does not already exist any other definition for a term matching to `Term`. This feature is useful for defaults in system makefiles which will only be used if there is no corresponding definition in application makefiles.

3. `depend`:
   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`:
   `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.

```
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 0.3: Makefile syntax of `prom`.

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

5. `include`:
   The content of a file will be inserted in this makefile.

6. `search include`:
   Like `include`, but the file will be searched through a predefined search path. Using this feature, the appropriate system makefile for the target machine will 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 files in need of such a flag.

The underlying PROLOG interpreter

C-Prolog, Version 1.5, was used for implementation. This is a small but fast interpreter for 32 bit UNIX machines available in source code.

This interpreter was enhanced by a builtin predicate timestamp for retrieving timestamps of files. This could be done very easily with only few changes to the source code.

prom was tested on HP 9000, Series 400 and PCS Cadmus 9600. Because C-Prolog claims to run on all 32 bit UNIX machines, prom can be used on this set of platforms.

The startup mechanism

The aim is to build a stand-alone software tool. So we want the PROLOG interpreter to automatically start with processing makefiles and updating target files. C-Prolog offers a mechanism using a startup file which can automatically be executed after invocation of the PROLOG interpreter. We use a startup file which first defines the special operators, then reads makefiles and finally updates a target file.

The name of the target file is read from a second startup file which is accessed prior to the makefiles. All command line options are passed to prom through this second startup file. Especially the system-dependent search path is transferred this way. So prom uses the appropriate mapping from its abstract names to system-dependent syntaxes.

The implementation

prom has been implemented under the paradigm of literate programming, as it was introduced in [Knu84]:

"Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do."

Programming literate, one writes documents in which program code and documentation text are integrated. A so-called WEB system extracts the program source out of these documents for machine interpretation. In absence of a PROLOG-WEB, we used MAKEPROG to write code for prom. MAKEPROG, as introduced in [Sch90], is a generic WEB processor without language-specific features. It can be used for arbitrary languages.

The source code of prom is structured as shown in Figure 0.4. It consists of three major components with functionalities described in the following:

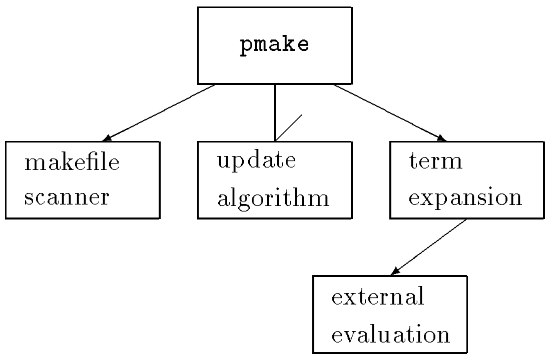


Figure 0.4: Internal structure of prom.

- **makefile scanner**
  The scanner reads makefiles, performs directory searches, checks the input syntax and transforms information into the knowledge base.
- **update algorithm**
  The update algorithm works as described above. It creates a file-dependency graph for the target file and triggers necessary actions.
- **term expansion**
  The term expansion facility performs transformations from terms into lists of atoms which represent literals in system-dependent syntaxes. A separate subcomponent is the interface to external programs which can be used to evaluate special terms.

Making use of `prom`

`prom` has been used for maintenance of several software systems. It has been found appropriate for large applications which have to be maintained in several versions and for several target machines. Multiple search paths allow system-wide and user-specific "system makefiles" to tailor adequate sets of rules and definitions for any kind of application.

In the following, principles of makefile writing for `prom` are treated on examples. Our first example simply shows definitions. We define the name of a program with `cc_cmd`. `obj(File)` defines the naming convention used for object files. Notice that `File` is an uninstantiated variable! While expanding this term, `File`'s current instantiation is concatenated with '`.o`' using the '`+`' operator.

```
define cc_cmd = 'gcc'.
define obj(File) = File + '.o'.
```

The second example shows an overridable default definition. An application-specific makefile may contain a line with the unconditional definition. This line must be read before the system makefile (with `default define`'s) is included.

```
default define project_dir = '$HOME'.
define project_dir = '/usr/foo/bar/application'.
```

Now, we consider an example of a file creation rule. Here, an executable file will be created out of corresponding object files. The application makefile contains the application-dependent definition of `objlist`. The last two lines of this example show the executed command when this rule is triggered for `exe(application)`.

As an unusual example, the "`depend`" line declares the dependency of each executable file from the makefile.

```
create exe(File) : objlist(File)
       --> call( cc_cmd, objlist(File), '-o', exe(File) ).
define objlist(application) = obj(one), obj(two).
depend exe(File) : makefile.

exe(application)
   ⟹ gcc one.o two.o -o application
```

The last example shows two features: First, `eval(...)` calls `get_includes` which outputs names of files included by `c(File)`. These files are added to the file-dependency graph as sources of `obj(File)`.

Second, `cc_flags` is an example of file-specific compiler switches. The creation rule contains `cc_flags(obj(File))` as a "hook". The last four lines show the commands executed for one file with `cc_flags` defined (`obj(file1)`), and one file without (`obj(file2)`). Notice that `cc_opts` is undefined for all files in this example!

```
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) ).
define cc_flags(obj(file1)) = '-I/usr/include/X11', '-DBSD'.

obj(file1)
    ==> gcc -I/usr/include/X11 -DBSD -c file1.c -o file1.o
obj(file2)
    ==> gcc -c file2.c -o file2.o
```

Future work

The following problems remain when working with `prom`. There may occur collisions of predefined PROLOG operators with term literals. For example, it is impossible to have a directory *include*, because of the operator having the same name. A possible solution may be the redefinition of the syntax avoiding any operator. This would lead to "traditional" PROLOG terms like `define(a,b)` instead of `define a = b`. But this modification reduces readability of makefiles.

Another inconvenience comes from commonly used UNIX shells. While entering the name of the target file, one has to express it in a term. The parantheses used here are interpreted by the shell. So one has to quote target filenames, like `pmake "exe(application)"`.

Further developments should improve robustness and error handling. Prevention of infinite loops and treatment of contradictions in makefile declarations belong to this field.

The list of "user wishes" also contains some debugging features. As an example, one can think of making the file-dependency graph visible to users. Other debugging information like rule identification could be useful, too.

Acknowledgements

I wish to thank the members of System-Programming-Group at TU Darmstadt for many helpful discussions; especially Prof. Waldschmidt for reviewing earlier versions of this work and Marion Dirscherl for improving the English version of this text.

Bibliography

[Fel79] S. I. Feldmann: Make—A Program for Maintaining Computer Programs. In *Software—Practice & Experience.* Vol. 9, No. 4 (1979), pp. 255–265.

[Goo73] G. Goos: Language characteristics. In F. L. Bauer, editor, *Advanced Course on Software Engineering*, Vol. 81 in Lecture Notes in Economics and Mathematical Systems, chapter 2, page 54. Springer, 1973.

[Kie90] T. Kielmann: "CAKE für MS-DOS". Semester Project, Institute of Theoretical Computer Science, Technical University Darmstadt, September 1990. (In German).

[Knu84] D. E. Knuth: Literate Programming. In *The Computer Journal*, Vol. 27, No. 2 (1984), pp. 97–111.

[Sch89] J. Schrod: "YADD—Yet Another DVI Driver Family". Diploma Thesis, Institute of Theoretical Computer Science, Technical University Darmstadt, January 1989. (In German).

[Sch90] J. Schrod: "The MAKEPROG System". Technical Report TI-3/90, Institute of Theoretical Computer Science, Technical University Darmstadt, 1990.

[Som87] Z. Somogyi: CAKE—A fifth generation version of make. In *Australian Unix system User Group Newsletter*, Vol. 7, No. 6 (April 1987), pp. 22–31.