

# Using PROLOG for Software System Maintenance

Thilo Kielmann

Institute of Theoretical Computer Science

Technical University Darmstadt, Germany

Internet: kielmann@iti.informatik.th-darmstadt.de

## 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 task. As a prototype, `prom` has been implemented which is introduced at the end of this paper.

## 1 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.

Unfortunately, there are a lot of people who are bound to use `make` to handle systems on many directories and with more than a couple of files. Large software systems usually need to be maintained in several versions to fit different purposes or to run on different machines. For example, consider a graphics toolset which has to deal with different output devices (displays, printers, plotters, . . .) while it is intended to be available for many different operating systems. The problems become worse if several programmers are working on closely related parts of the source code.

The wide range of problems in this area is known as “Software Configuration Management” (SCM). SCM systems like `SHAPE` [ML88] or `ADELE` [Est88] claim to overcome `make`’s restrictions. They replace `make` by improved versions for creating specific configurations of software systems.

There already exist some SCM tools which exploit logic-based approaches: `ADELE` uses constraints formulated in first-order logic. Another approach (as described in [SB90]) works on the basis of PROLOG proofs. The authors propose several predicates to be added into the PROLOG interpreter for treating file contents and program calls. From our point of view, it would be better to have a PROLOG-“program” managing a knowledge base about the software systems to be maintained. On this level, PROLOG’s facilities like term unification and reasoning with first-order predicates can be exploited in an adequate manner.

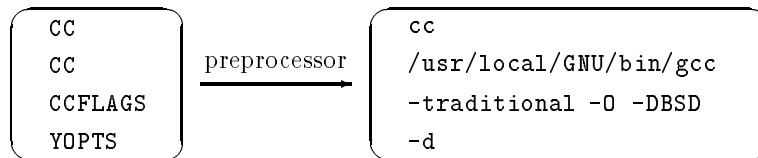
Our former work on `make` tools for large portable software systems (see [Kie90]) specified the programmer’s need of an adequate and structural view on the application. Current `make` systems operate on a bunch of hardcoded features and notations. Instead of this, one should be able to describe software systems on an abstracted level. So programmers should be able to deal with components of a system instead of “juggling” with compiler switches and filename extensions.

A “good” `make` tool should provide an abstracted view on the components of a software system by hiding system-specific details under an abstract notation.

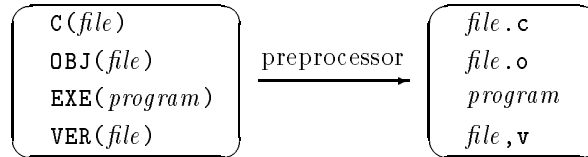
## 2 Using a preprocessor

In the following, we consider a `make` tool which uses the C preprocessor for its input files. The C preprocessor’s macro facilities are the key to the abstracted notation mentioned above.

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.



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:



While exclusively using these names, one can easily state relations between the files in use without mentioning any machine-dependent name conventions.

Such C preprocessor macros are only helpful if definition and utilization are 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 modules.
- 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 the **make** tool 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 specify 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 a makefile’s creation rules. 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 declarations also occur in the system directories together with the declaration of the rules.

### 3 Problems not solvable by a preprocessor

On an idealized view, all information needed to create a program out of its sources is located 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 explained with the following example. We consider a file generation rule in the syntax known from **make** in which a target file is separated from its sources by a colon while the subsequent lines describe the actions to be performed.

The most obvious “exception” from default rules is the creation of an executable binary 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 may try to formulate the list of object files as a preprocessor macro, too. But this fails, because a preprocessor is in principle unable to expand a macro “conditionally” depending on its parameters.

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 and modify 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.

## 4 Using a PROLOG interpreter as a make tool

From the experience with the previous approach we conclude that makefile entries 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.

This reasons suggest PROLOG as a predestined candidate for the implementation of a **make** tool. A closer look at the demands of such a tool shows that PROLOG is indeed powerful enough to perform all necessary tasks. The set of requirements includes file reading, computation of actions to be performed, and execution of external programs. This all can be done by standard PROLOG predicates. The only necessary extension is needed for getting timestamp information of the files. Depending on the capabilities of the PROLOG interpreter at hand, one can achieve this via calls to external programs or via adding some code if there exists an interface.

## 5 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.

One major part of information in the knowledge base is extracted from two external sources. First, the system makefiles located in so-called system directories declare terms to map the abstract notation onto the requirements of the target machine. They form the system-specific part of the knowledge base. Second, the application makefiles located in the source code directories declare terms concerning files of the specific application. They form the application-specific part of the knowledge base.

Applying the update algorithm, the necessary 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. 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 PROLOG predicates which represent the necessary information. A first group declares creation rules and dependencies between files. A second group describes how terms of the abstracted notation have to be expanded depending on the target machine. The last group collects information about the files involved in the update process. They concern matters like existence, timestamp, and actuality with respect to the makefiles in use.

## 6 The realization of prom

**C-Prolog**, Version 1.5, was used for implementation of **prom**. This is a small but fast interpreter for 32 bit UNIX machines available in source code. The interpreter was enhanced by a builtin predicate **timestamp** for retrieving timestamps of files. **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 aim was to build a stand-alone software tool. So the PROLOG interpreter is embedded in a startup script which performs the system-dependent settings before automatically executing the file update process.

Concerning implementation and knowledge representation, a more detailed description of **prom** can be found in [Kie91].

## 7 Makefiles interpreted by prom

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**. 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**.

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.

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 **:'**. **prom** processes makefile entries of the six types listed below.

1. **define** *<term>* = *<list of terms>*  
**define** declares how to expand terms. Terms are constants or structures in the sense of PROLOG.
2. **default define** *<term>* = *<list of terms>*  
**default define** declares system-wide default terms which may be overridden by application specific declarations.
3. **depend** *<target>* : *<list of source files>*  
**depend** declares on which sources a target file depends. Target and source files are terms.
4. **create** *<target>* : *<list of source files>* --> *<list of actions>*  
**create** declares a creation rule for a target file. Actions are terms in the form of **call**(...,...) with arity greater or equal to one.
5. **include** *<file>*  
The content of *file* will be inserted in this makefile.
6. **search include** *<file>*  
*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**. First, the '+' operator concatenates its arguments to one single atom. The special functor **eval** interprets its structure components as an external program call. The output of the program called is taken as term expansion. "Undefined", non-atomic terms will be expanded to nothing. For example, one may use the term **compiler\_flag**(*program*) which will only be expanded for files in need of such a flag.

## 8 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 example. First, we consider a file containing rules for creating executable binaries from C source code. It resides in a system directory.

Rules
<pre> create exe(File) : objlist(File)     --&gt; call( cc_cmd, link_opts, objlist(File), '-o', exe(File) ).  create obj(File) : c(File), eval( get_includes, c(File) )     --&gt; call( cc_cmd, cc_opts, cc_flags(obj(File) ),               '-c', c(File), '-o', obj(File) ). </pre>

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
<pre> search include 'Rules'. define cc_cmd = 'gcc'. define c(File) = File + '.c'. define obj(File) = File + '.o'. define exe(File) = File. </pre>

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

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 makefile 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
```

## 9 Conclusion

PROLOG’s facilities of term unification and reasoning with predicate logic form an excellent basis for a `make` tool capable of dealing with large software systems. A powerful set of creation rules allows a declarative description of a software system’s components. Programmers only need to state attributes of and relations between an application’s components. Thus makefiles are reduced to maintainable sizes. They only contain application-specific information and may therefore also be generated by a SCM tool.

## 10 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.

## References

- [Est88] Jacky Estublier. Configuration management, the notion and the tools. In *International Workshop on Software Version and Configuration Control, Grassau, FRG*, pages 38–61, Jan. 1988.
- [Fel79] Stuart I. Feldman. Make—a program for maintaining computer programs. *Software—Practice & Experience*, 9(4):255–265, 1979.
- [Kie90] Thilo Kielmann. Cake für MS-DOS. Semester Project, Institute of Theoretical Computer Science, Technical University Darmstadt, Sep. 1990. (In German).
- [Kie91] Thilo Kielmann. *PROM: A flexible, PROLOG-based make tool*. Report TI-4/91, Institute of Theoretical Computer Science, Technical University Darmstadt, 1991.
- [ML88] Axel Mahler and Andreas Lampen. An Integrated Toolset for Engineering Software Configurations. In *ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, pages 191–200, Nov. 1988.
- [SB90] Paul Singleton and O. Pearl Brereton. *An infrastructure for experimental logic-based SCM*. TR 90.02 Computer Science Dept., Keele University, UK, 1990.