tmk – A Multi-Site, Multi-Platform System for Software Development

Hartmut Schirmacher, Stefan Brabec*
Max-Planck-Institut für Informatik
Im Stadtwald, 66123 Saarbrücken

Abstract

`tmk` is a tool that embeds the functionality of `make` in the scripting language `Tcl` in a very simple and convenient way. Furthermore, `tmk` allows higher levels of abstraction via *modules* and a flexible *configuration* framework. In addition to using `tmk` simply as a replacement for `make`, the users can create *projects* with global methods, objects, and options, and extend or modify the globally defined tasks using per-directory control files similar to the traditional `Makefile` concept.

We give a brief overview of `tmk`'s core concepts, such as target and dependency definition, exception handling, and parameterization of targets and modules. Furthermore, we show some examples of how to use `tmk`'s configuration system for multi-platform software development and projects shared by multiple sites.

1 Introduction and Related Work

Task automation plays an important role in the context of complex structures and computations. In areas such as software development and system administration, the tool of choice for doing this has been `make` for a long time, nowadays mostly replaced by GNU `make` [8]. Both basically work by checking file timestamps and executing shell commands that update the corresponding files if necessary.

1.1 Motivation

Although GNU `make` is a sophisticated tool that provides a lot of very special functions, many users have found it inconvenient to use for different reasons.

First, `make` and GNU `make` have a very special (some say cryptic) syntax of their own [8], and the user has to understand both this syntax and that of the command shell used for executing the actual commands.

*{schirmacher,brabec}@mpi-sb.mpg.de · http://www.mpi-sb.mpg.de

Second, the proprietary syntax also implies a lack of control structures and abstraction concepts within the `Makefile`. GNU `make` adds some more structural elements, but it is still not suited for scripting in general.

Third, the communication between `make` and the called shell scripts is quite primitive, especially from the script back to `make`. The basic understanding is that `make` tests whether an action seems necessary, and then calls a command or script that does the actual work.

Finally, another inconvenience with `make` is the lack of real support for multi-directory processing and inter-directory dependencies, which makes it quite hard to manage large project trees with `make` in a transparent fashion. Moreover, architecture- and site-dependent settings cannot be hidden from the `Makefile` user, which complicates the design of a portable `Makefile`. A similar problem is the construction of variants, which can only be accomplished using various workarounds that make each `Makefile` still more complex.

1.2 `Perl` and `cons`

One promising way of avoiding many of the above-mentioned problems is to embed `make`'s functionality into a powerful and well-established scripting language such as `Perl` or `Tcl`. Bob Sidebotham has created such a tool called `cons` [1]. `cons` uses control files written in the the `Perl` language, and allows to define so-called *construction environments* that hold a number of variables for parameterizing the building rules. `cons` addresses most of the problems mentioned above, including native scripting features, multi-directory projects, abstraction from rules and configuration, and variant building. However, the basic language is `Perl`, the syntax of which does not appeal to everybody. Moreover, the concepts introduced by `cons` are very powerful, but they do not necessarily lead to very simple and transparent control files.

1.3 `Tcl`, `tclmake`, and `bras`

Up to our knowledge two tools have been presented that try to bring together `make` and `Tcl`. The first one, called `tclmake` [5], processes files written in a subset of regular `make` syntax, but with the commands for each rule written in `Tcl`. Although `tclmake` makes it possible to use `Tcl` commands for building the targets, unfortunately the strange `make` syntax is kept, and target definitions cannot be well embedded into scripts. As stated by the author, `tclmake` does not aim to replace `make`, and there is no support for any abstraction layer on top of the core `make` functionality.

"Another kind of `make`" is presented by Harald Kirsch's `bras` tool [3]. Motivated by similar reasons to those presented in Sec. 1.1, Kirsch has implemented the core functionality of `make` as a set of `Tcl` procedures. `bras` allows to define targets and dependencies in a more abstract and flexible way than `make`, and especially it is possible to define arbitrary conditions that trigger a rule, not just

those based on a file's time stamp. bras also supports explicitly specified inter-directory dependencies and a globally defined rule database. However, it does not extend to such abstraction layers as architecture-dependent or variant builds, or site-dependent configuration. Additionally, subdirectory processing is limited (due to variable scope problems), and bras cannot properly choose among multiple rules for the same target.

2 tmk Overview

The design of tmk has been driven by the demand for two things: a simple system for managing larger software projects without having platform- or site-specific code in each Makefile, and a scripting environment that is combined with the core functionality of make. As common basis for achieving both goals, we have chosen to embed make-like functions into Tcl. Additionally, the tmk core natively supports architecture-dependent output, multi-directory processing, and things such as exception/exclusion handling.

On top of the tmk core, we have added additional abstraction layers by a module mechanism and a centralized configuration system. Through this, it is possible to remove any platform- or configuration-specific code from the control files.

In the context of this paper we can only briefly sketch tmk's components. If you want to know more about a specific topic, please have a look at the tmk tutorial and tmk reference manual, both available on the world wide web [6, 7].

2.1 The tmk core

As it has already been demonstrated in [3], it is relatively easy to embed the core functionality of make into Tcl. The control files for tmk are simply Tcl scripts, called TMakefiles. From the user's point of view, the tmk core consists of the following three procedures:

```
target {target-list} {src-list} {rule-script}
depend {target-list} {dep-list}
build  {target-list}
```

target defines how to create a target from a set of source files or *primary dependencies*. depend is used to declare additional *secondary dependencies*. Primary dependencies alone are used to select the appropriate rule (if there are multiple candidates) for each target, whereas secondary dependencies simply define additional preconditions before the dependent target can be built. build declares *default targets* that can be overridden on the command line.

Rule Triggering. In a way similar to make, rules are triggered when the target does not exist or any of the dependencies is newer than the target. In contrast to make, targets without primary dependencies are only built if they do not exist. Only if a target depends on the special symbol ALWAYS_BUILD, it is built unconditionally.

Target Patterns. Glob-style pattern matching and special variables are used for defining target *classes* and target-dependent expressions. The following example shows how different glob-matched parts of the target name as well as the complete name can be used in the source file expression and in the rule script:

```
target pic*.step*.* {pic$0.step[expr $1 - 1].$2} {
  puts "creating $TARGET from $SRC"
}
build pic34.step5.jpg
```

This example will produce the following output:

```
creating pic34.step5.jpg from pic34.step4.jpg
```

Exceptions and Exclusions. In addition to these most basic features, `tmk` also has a way of handling exceptions and exclusions. An *exclusion* means that some target will not be built and will not appear as dependency in any rule. Targets that loose all their primary dependencies because of excluded source files will also be skipped. An *exception* temporarily overrides the values of some variables for just some targets. Exceptions also allow to replace the rule completely by a different one.

Output Directory. Another important feature of the `tmk` core is that all targets can be placed in an architecture-dependent output directory. If switched on, this mechanism automatically and transparently augements target and dependency names by a directory named `$ARCH`. This allows to generate code separately on different machines without any further effort. Furthermore, the generated files can be well distinguished from the original ones.

Target State. Targets and dependencies are *cached*, and it is possible to query and modify their *state*. For example, a rule that is triggered can perform a more sophisticated test in order to determine whether the target really needs to be updated. If not, the script simply marks the target as untouched, thereby not falsely triggering any further rules.

2.2 Modules and Configuration

On top of the core, `tmk` has a *module* mechanism that allows to globally store rules, options, and procedures for certain classes of tasks. Modules are explicitly requested in the control files in order to allow the user to choose the right set of methods for the specific task, and they are parameterized through global or namespace-relative variables. Have a look at Section 4 for some examples of module usage.

Site-dependent variables (e.g. installation paths) are not defined inside the module, but rather in the appropriate *site-config* files that are processed by `tmk`'s central *configuration system*. Similar to this, `tmk` reads *arch-config* files that define architecture-dependent options, like for example a procedure for how to call the compiler and linker for a certain task.

Tcl defines a large number of basic commands that transparently hide the underlying operating system. Therefore, module writers and users can design portable and transparent scripts and rule databases. The configuration variables and procedures allow to extend and modify the predefined configuration in a simple and convenient way, without touching the code in the individual TMakefile.

3 tmk's Configuration System

As already mentioned, tmk's control files can easily be designed to be platform and site independent[1]. Usually, porting a project to a different platform starts with re-writing and patching lots of Makefiles and/or configuration scripts. Automation for this task exists, but is usually limited to certain systems. For X11-based systems, configuration and portability can be achieved with tools like imake [2], whereas in the non-UNIX world (Windows 95/98/NT, MacOS, etc.) multi-platform projects are managed using so called *Integrated Development Environments* (IDE) that provide virtually the same view for different underlying architectures.

Major drawbacks of nearly all those development tools are that they are either too complex and inconvenient for everyday use, or they do not provide the desired flexibility.

The approach presented in this paper is slightly different: Instead of treating Makefiles as system-specific, all details of the underlying hard- and software are encapsulated centrally by tmk itself. In this scenario, the TMakefile is no longer a system interface, but an abstract mechanism to describe how all the files in a project should be processed to build the final products.

3.1 Multi-Platform Support

The first step in building an abstraction layer is already done by Tcl itself since its available for a wide range of systems. In contrast to traditional shell command based system, such as make or autoconf [4], the Tcl language provides a well-defined set of comprehensive structural elements (procedures, loops, lists, variables, associative arrays) as well as a large number of architecture-independent library functions (filename handling and file operations, network communications, resource manipulation, systems information queries).

tmk extends this abstraction concept further by hiding platform-specific details such as compilers and software packages. The principal idea behind this is sketched in Fig. 1. The configuration system is split into two major branches. The first covers architecture-specific information such as compiler environments, operating systems, and system-specific utilities, while the other branch is used to

[1] *site* in this context means the local computing environment, e.g. the location and versions of libraries, compilers etc.

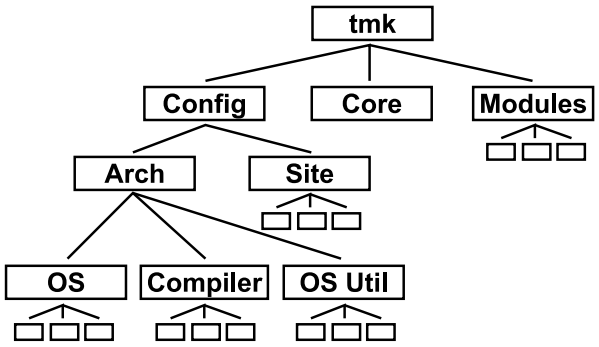

Figure 1: The components that build up `tmk`. The configuration is split into two major branches to separate architecture-specific configuration from site-related issues like package existence and installation paths. The CONFIG/ARCH branch is split further to distinguish between operating-system support and things like compiler environments and OS-specific helper tools.

describe the site-specific environment, e.g. information about the availability of packages as well as local installation paths and library names.

For example, the CONFIG/ARCH/COMPILER branch defines a *meta compiler* interface for compiling object files, linking executables and libraries, and maintaining dependency information. Rather than constructing the necessary command lines by simple textual substitutions and flag combination, the compiler meta-functions are comprehensive procedures that can easily be adopted even for very uncommon compiler syntax or semantics.

So far the described concept has proved to be convenient for integrating several different compiler environments, such as

- SGI MIPSPro Compiler System
- GNU Compiler Collection (gcc)
- Borland C/C++ Compiler 5.5 (BCC)
- Microsoft Visual C/C++
- SUN WorkshopPro.[2]

In order to add support for compilers that are available for more than than one operating system, the actual platform-specific settings (CONFIG/ARCH/OS branch) are decoupled from the compiler environment. The widely used gcc for example is available for various systems so the corresponding *meta compiler* procedures can be used on many operating systems without any further modifications. Currently `tmk` 's list of supported platforms include

- SGI IRIX
- Linux

[2]All of these are included via their command line interface. Currently there is no support for special features such as project managers or graphical interfaces.

- FreeBSD
- SUN Solaris
- Cygnus/RedHat CygWin
- Microsoft Windows 95/98
- Microsoft Windows NT.

The third branch, CONFIG/ARCH/OS UTIL, contains platform-specific utilities, e.g. for hardware information queries and dependency generation[3].

Maintenance and extension are simplified due to the modular structure of the configuration, since side effects and code replicas are minimized. So we are positive that the system can be easily adopted to additional compilers and operating systems. This is especially true for all UNIX/POSIX based platforms, but should also extend to others like BeOS and MacOS.

3.2 Multi-Site Support

Another very important topic when designing a make tool for software development is finding a comfortable and yet flexible way of integrating e.g. third-party software packages. At the moment, nearly all configuration utilities such as `autoconf` [4] rely on description files that are usually part of a package release. Often, this mechanism cannot be used directly for certain reasons. First, and most important, software packages used during the development phase may be non-stable prereleases or contain special patches which should not be integrated into the standard installation of a running system. Secondly, software development is a very time-consuming task so all configuration issues should be reduced to a minimum by re-using information which remains constant on a certain machine or in a certain computing environment.

`tmk`'s solution for this is a "best of both worlds" approach. Firstly, standard software installations are supported by gathering information via system-specific tools. For example on `X11`-based machines, `tmk` can obtain information about the location of various libraries and header files via the `imake` [2] configuration tool. This is done by creating a dummy `Imakefile` which is then converted to a regular Makefile using `xmkmf`. Examining this Makefile gives information about nearly all `X11`-specific details, such as include paths or the required libraries to build an executable. Future plans include the integration of package managers such as RedHat's `rpm` or Debian's `dpkg`.

Secondly, `tmk` provides a mechanism for non-standard installations. It is often the case that developers, although they are working on different projects, wish to share special software packages. Instead of performing intensive copy-and-paste actions from other project makefiles, it makes more sense to have a centralized package mechanism which can be easily accessed and extended. This *software database* is integrated in `tmk` using site-specific configuration files (CONFIG/SITE branch in Fig. 1).

[3] In cases where the compiler is not capable of doing this by itself.

After the platform-specific configuration is done (Section 3.1), `tmk` starts searching for a suitable configuration file in the site-config directory[4]. The description files found in this phase are normal `Tcl`-scripts which add package descriptions depending on platform-specific settings. As an example, one might add support for the TIFF library (installed in an uncommon place) by just appending a few lines like

```
addAndSet TIFF_INCDIR { /opt/pckg/tiff-v3.5.3_patched/include }
addAndSet TIFF_LIBDIR { /opt/pckg/tiff-v3.5.3_patched/lib }
addAndSet TIFF_LIBS   { tiff }
```

to the actual site-config file. The process of using the TIFF-Library now simplifies to a single `module { TIFF }` call, regardless of the underlying operating system, compiler or software installation.

These mechanisms greatly simplify the process of concurrent software development because whole projects or even single sub-directories can be shared without any further modifications. Since system- and software-specific details are encapsulated by `tmk`, all developers work in a virtually identical environment.

4 Examples

This section gives a few short examples that show how `tmk` can be used. If you are interested in details, please have a look at the `tmk` tutorial [6], which contains many examples with more elaborate explanations.

C++ Compilation and Linking

When using the provided C/C++ modules, compilation and linking is made really easy. Have a look at the following example:

```
module cxx
lappend PROGRAMS {test1 prog2}
lappend SYSLIBS  {m pthread}
lappend EXCLUDE  stupid_test.cc
```

This very basic example will search for all files with a suffix that indicates C++ code (e.g. `.cc`, `.cpp`, `.C`, `.cxx`), and generate the following targets (on a UNIX platform):

- compile `.o` files from all C++ source files (but not `stupid_test.o`)
- create a library from all object files except `test1.o` and `test2.o`
- create the executables `test1` and `test2` by linking them with the local library and the specified system libraries

[4] This search is done by examining files which match a certain pattern, e.g. `${DOMAIN}` or `${HOST}-${DOMAIN}`.

QT Library and 'moc'

There are several modules that can be combined especially well with the C++ module, as for example that for Troll Tech's QT library[5].

```
module {cxx qt}
lappend PROGRAMS ...
lappend SYSLIBS  ...
```

As you can see, this example does not differ much from the previous one, but additionally the following things happen:

- `tmk` looks for all header files (matching a number of predefined and customizable suffixes) that contain the keyword `Q_OBJECT`,
- for all matching header files, the QT precompiler 'moc' is called, and a corresponding C++ file `basename.moc.C` is generated,
- the moc'ed source files are treated as the "normal" source files in the directory, meaning that they are included in the library, or linked to become executables, or whatever is appropriate,
- the `qt` library is added to `$SYSLIBS`,
- module dependencies are resolved, so that the appropriate support libraries (e.g. X11 or Win32 libs) are added automatically.

Code Project Trees

In the case of a multi-directory project, `tmk` employs the concept of *project libs* in addition to system libraries. Project libs are specified by their path relative to the project root. For example, if you are in some subdirectory of project `A` and want to add the libraries from project `A`'s subdirectory `a1/a2` and project `B`'s subdirectory `b1/b2/b3`, you can do this as follows:

```
module {cxx ...}
lappend PROGRAMS ...
lappend SYSLIBS  ...
lappend PROJLIBS A/a1/a2 B/b1/b2/b3
```

In this case `tmk` will add the appropriate include and libary paths (including the architecture-dependent subdirectory), and do everything needed for linking the executables with the specified project libs.

Another nice feature is that using the notation of project libs, you can easily collect parts of your projects from different places. `tmk` will search for the project libs in the root of your current project directory as well as in all paths specified in the `$PROJ_LOCATIONS` variable. Since the actual file name of the library is never used directly, it is possible for `tmk` to assign each library a unique name so that libs will never get mixed up.

[5] `http://www.troll.no`

5 Conclusions and Future Work

Our experience has shown that software development is simplified a lot through the use of `tmk`'s configuration and module concepts. It is both possible to perform very uncommon tasks in individual parts of a project, and to use the predefined modules in a very simple way.

This allows novel users to start or join a project without first getting familiar with complicated make utilities, the local installation, or compiler versions and flags. Moreover, the developer does not loose the flexibility of an all-purpose automation tool, and also gains the transparency of a powerful high-level scripting language. `tmk` has also proved to serve extremely well for projects that are shared by collaborating groups of developers and that get installed on several different platforms.

Future work on `tmk` includes distributed and parallel processing and a more sophisticated support for package versions. One could also think of designing a GUI on top of `tmk` in order to simplify tasks like project management and configuration issues.

We would also like to increase the number of supported platforms and software packages, and collect additional modules written for `tmk`. We hope that more people will find `tmk` useful and will support its further development.

References

[1] Bob Sidebotham. Software construction with Cons. *Perl Journal*, 3(1), 1998.

[2] Paul Dubois. *Software Portability with Imake*. O'Reilly, 1996.

[3] Harald Kirsch. bras — another kind of 'make'. `http://wsd.iitb.fhg.de/~kir/brashome`

[4] David MacKenzie and Ben Elliston. Autoconf. creating automatic configurations scripts. `http://www.gnu.org/software/autoconf`.

[5] John Reekie. tclmake – a tcl-only make-like utility. `http://www.eecs.berkeley.edu/~johnr/code/tclmake`

[6] Hartmut Schirmacher. *A Tutorial to tmk*. `http://www.tmk-site.org/doc/`.

[7] Hartmut Schirmacher and Stefan Brabec. *tmk Reference Manual*. `http://www.tmk-site.org/doc/`.

[8] Richard Stallman and Roland McGrath. GNU Make. `http://www.gnu.org/software/make`.