A CASE-Oriented Configuration Management Utility

W. Tuppa and S. Selberherr
Institute for Microelectronics, TU Vienna
Gußhausstraße 27–29/E360, A–1040 Vienna, Austria/Europe
+43–1–58801/3680 tuppa@iue.tuwien.ac.at

Abstract— **Traditional make utilities usually lack the necessary functionality for the management of multiple configurations in one source code tree. Our CASE–oriented configuration management utility, the Viennese Make Utility (VMake) is platform-independent and runs currently on a number of *UNIX* systems and on *VMS*. VMake supports, in addition to common make features, a number of CASE tasks like automatic code generation, version management (using RCS and CVS from a common repository), and automated high-level source code processing features, like language bindings between C, LISP and FORTRAN and extraction of reference manuals. VMake maintains automatically a private project file which contains up-to-date symbolic definitions of source code files, modules, libraries, language binding mechanisms, application executables, and all build targets. Dependencies between these objects are extracted from local description files or generated automatically from source code files. This enforces compact description files and allows for efficient management of large–scale software projects. VMake is based on a publicly available LISP interpreter[1].**

Keywords— **configuration management, automatic code generation, languages, parallelization**

1 Introduction

Most of the commonly used configuration management systems are based on the make utility by Feldman[2]. Extensions are made to this basic tool either by modification of the make functionality or by preprocessing higher–level configuration description files to generate the low–level Makefiles required by make. The concept of make is based in the incremental execution of rules that successively transform code objects (files) until certain build goals are reached. The applicable rules are comprised by a built–in part (default rules) and optional extensions provided by the user. Among popular make implementations, different features and peculiarities arising from the close proximity to the operating system prohibit the direct exchange of the description files and are a subtle burden to portable software systems. Independent of the particular implementation, a common drawback is that an increasing number of modules complicates the maintenance of the project information significantly and hence confines the application of make–based configuration management to small–scale software projects.

For configuration management of the well known large–scale X Windows System (X11), the imake utility[3], a preprocessor to standard make, was created. Using the C preprocessor, Makefiles are generated from small description files by accessing rules and system configuration data which are stored in additional global files. This approach benefits from the reduction of complexity and maintenance effort thanks to the use of standardized higher–level rules for the description of the modules. imake adds at least one additional

```
SRCS  = main.c
OBJS  = main.o
LOBJS = my1.o my2.o
DEPLIBS = libmy.a

/* build library */
NormalLibraryTarget(my,$LOBJS)
/* generate program */
ComplexProgramTarget(myprog)
```

Fig. 1. Standard entry in an Imakefile

pass to the build process to create the description files for make and doing so it deletes any saved dependency information (which must be regenerated too) from the Makefiles. The reduction in size of the description files and the availability of global project information enables the management of larger scale software systems. However, the global information consisting of rules, system dependencies (e.g., where to find include files and installed libraries), and definitions of global objects (e.g., libraries), is contained in a separate set of files, which are provided and maintained manually by the software engineer.

Other approaches extend make by directly adding new functionality like multiple goal evaluation at the same time or inclusion of sub–description files (for commonly used data). A common freely available extended make implementation is gmake from the GNU project. Another way goes jam[4], which reads all description files, which are based on a simple language by it's own, to generate a full dependency tree (on every invocation) and then builds all goals in a second pass.

The approaches mentioned are still lacking some features needed for the efficient management of large–scale software projects. Several advanced commercial packages can be found which address the CASE process as a whole and overcome most particular problems of configuration management, but the implementations are closely connected to the underlying system and are therefore not portable among different operating systems[5].

2 VMake

VMake employs a small number of standardized higher–level rules to reduce the complexity of local module description files, but overcomes the aforementioned insufficiencies of imake by maintaining all global project information automatically in a **VMake**–internal global project context file. This file is generated from the information of the local description files and holds the time of the last modification of each local description file (in addition to the time stamps of all managed source code files) and is updated automatically. Changes to the local description files are recognized and the (partial) regeneration of the dependency information is done automatically. In addition to the local project dependencies inter project dependencies are recognized automatically (only the names of required projects must be given in the top level description file of the project). Fig. 2 shows a **VMake** example description file for the same program as shown in Fig. 1 by imake. Since **VMake** is based on `LISP`, the syntax chosen is a subset of `LISP` so that the `LISP` reader can be used for parsing.

```
;; this defines a name for the directory
(Module-Directory My-dir)
;; compile main source file
(CC-Target My-C-main :source "mymain.c")
;; compile library objects
(CC-Target My-C-objects :source "my1.c" "my2.c")
;; build library
(Library-Target My-C-library
                :libname "my"
                :objects My-C-objects)
;; generate program
(Program-Target My-C-program
                :progname "myprog"
                :objects My-C-main
                :libraries My-C-library)
```

Fig. 2. Example of a **VMake** description file

The rule `Module-Directory` defines a symbolic name for the source code directory. The `CC-Target` rules are used to compile the main object and the library object files. The object files are never named explicitly in the description file but are referred to by a symbolic name. This symbolic name is automatically added to **VMake**'s global name space and can be accessed in any other local description file within the project (and in other projects which depend on this project). All files are accessed by their symbolic name which has to be unique within all projects. In Fig. 2 the symbol *My-C-Main* is bound to "mymain.o" under *UNIX* and bound to "MYMAIN.OBJ" under *VMS*. Thus, hiding the system–specific file names and other system dependencies through symbolic names, the same description file can be used on entirely different operating systems. With the rule `Library-Target` a library is generated and bound to the symbolic name *My-C-Library*. The objects for the library are specified by the symbolic name *My-C-objects*. Furthermore, the name of the library is only given partially and extended internally to "libmy.a" (*UNIX*) or "LIBMY.OLB" (*VMS*), respectively. Since the flag `:shared` is not given, this library is generated as an archive library (if supported by the operating system a shared library could be generated on demand). Finally the program is generated from the main object and the library. Again the program name is defined without extension (this is added by **VMake** automatically). The generated executable code is automatically linked (by a symbolic link, if supported by the operating system, or a hard link) into a common directory for executables to shorten user's search path for later execution.

make and imake–based approaches usually must perform multiple passes over a project source tree to reach a certain build goal, during which many possibly non–needed objects and libraries are built. **VMake** exploits the fine–grained global dependency information to rebuild an utmost concise superset of the really required objects. Thus, only the necessary files are updated to speed up the rebuild cycle time. Furthermore, no unnecessary checks of non–needed files are done (make and imake scan their description files many times and gather just the same information in each invocation). **VMake** allows easily to combine multiple projects into bigger ones. In the top level description file of **VMake** the engineer can specify inter project dependencies by specyfing the symbolic names of the required sub–projects in the actual project. The required information is automatically read by **VMake** (either from a working or installed project) and dependencies are checked globally over project boundaries on demand.

3 Hiding System Dependencies

Path names, file names, compiler invocation, compiler options and arguments, generation of libraries and programs, execution of applications, redirection of input and output, error handling, and return codes are all highly system–dependent features. **VMake** encapsulates all system–dependent functionality in generic transformation functions which map the symbolic definition to actual compiler or linker system calls, by using either a simple configuration file or by overriding the generic functions with specific implementations for more complex tasks (e.g., for building shared libraries under IBM's AIX operating system).

4 CASE operations

4.1 Tool Abstraction Concept

VMake uses a *Tool Abstraction Concept* (**TAC**) for generating language bindings of functional modules and constants for different programming languages. Currently, bindings can be generated from `C` to `FORTRAN`, from `FORTRAN` to `C`, and from `C` to `LISP`. The automatic support of multi–language programming has proven valuable for two reasons. First, writing the required stub code manually is a tedious and error–prone task, and secondly, multi–language interfaces between compiled languages are highly system–dependent. Usually, language binding is done by the programmer by writing `C` files with some `#ifdef/#endif` pairs to generate code for the different platforms. This works quite well under *UNIX* because parameters are handled mostly the same way and only the function naming differs. In contrast to *UNIX*, *VMS* has a totally different handling of strings (especially under `FORTRAN`). In this case the stub code has to be written in a totally different way. In addition to this problem some other code is often integrated into the stub, adding functionality, which does not belong to the actual function binding. Generating the stub code automatically (from a description) avoids both of these problems.

The **TAC** module of **VMake** scans the source code file (similar to a preprocessor) and extracts information from the function definitions and special formal comments, as depicted in Fig. 3. The comment `/***TF` starts the definition of a **TAC**–able **Function**. The comments after the function arguments consist of a formal description of the argument characteristics and a textual documentation part which is also used for the documentation extraction facility.

```
/***TF counts the number of occurrences of a
    character within a string. The start and
    end of the search range can be specified
    to simplify substring operations. */
/***R myStrChar myStrReverseChar */
int /* [:not-ok 0] */
   myStrCount(
         char *str,    /* [IN] input string to search */
         char ch,      /* [I] character to search for */
         int start,    /* [I :opt :key :default 0] start
                          index for search */
         int end)      /* [I :default strlen(str) :opt
                          :key] end index for search */

{
  /* implementation of function */
}
```

Fig. 3. **TAC** documented function

In the example in Fig. 3, all parameters are used as input `[I]` and `str` may be given as `NULL` pointer `[IN]`. To bind the function `myStrCount` to another language, the definition is used in the description file of the module implementation.

```
(Module-Directory MyModule)
(Define-TAC-Interface TAC-module
       :files "mysrc.c" ; source file of function
       :module "my"
       :source-domain C)
```

To generate a `LISP` binding for the `C` function `myStrCount` (which is part of the module `"my"`) somewhere else in the project tree the rule

```
(Create-TAC-Interface TAC-LISP-Interface
          :modules "my"
          :target-domain VLisp)
```

has to be used in the description file where the language bindings shall be generated. All the **TAC** information that has been extracted by **VMake** is tied to the symbolic name and prefix of the module. Once defined, this information can be used for the generation of multiple language binding interfaces. The rule

```
(Create-TAC-Interface TAC-FORTRAN-Interface
          :modules "my"
          :target-domain FORTRAN)
```

is used to create a `FORTRAN` binding for `myStrCount`. The system dependency of a leading and/or trailing underline and the case of the function name is handled automatically by **VMake**. The conversion of `LOGICAL`s, character arrays (`C` strings) and indices is done between `FORTRAN` and `C`. The usage of the resulting `LISP` and `FORTRAN` functions is depicted in Fig. 4.

```
(my::str-count str #\Space :start 5 :end 15)
 ;; search for Space character with in limits

CHARACTER*20 STR
INTEGER COUNT
CALL MYSC(STR, ' ', 5, 15, COUNT)
```

Fig. 4. Example of `LISP` and `FORTRAN` call of **TAC**–bound `C` function `myStrCount`

A stub code in `C` is generated from the extracted information to generate the language binding between source and target language. This stub code needs than to be linked to the application. In the case of `LISP` a single object file is generated, whereas for `FORTRAN` an object library is generated to avoid the linking of non–needed functions.

The **TAC** is also used for the extraction of reference manuals from the source code. A function documented with a `/***F` comment, as depicted in is parsed by **VMake** and a LaTeX reference manual entry for that function is generated for a reference manual.

4.2 Universal Function Generator

The *Universal Function Generator* (**UNFUG**)[6] provides language–independent, advanced preprocessing to generate repeated program code sequences with slight variations. It uses so called template and tuple files which are combined to produce a compilable output source code file in an arbitrary programming language. The template file consists essentially of source

code with occasional meta–strings (variables), which are replaced with actual values from the tuple file during the **UNFUG** run. In Fig. 5, the **UNFUG** com-

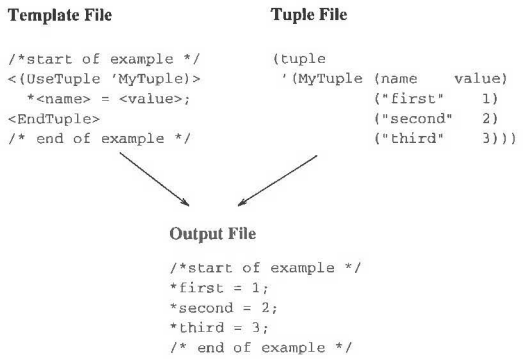

Fig. 5. Example of **UNFUG** generated code

mand < (UseTuple 'MyTuple) > selects the tuple to be used with the template file (the angle brackets "<" and ">" delimit **UNFUG** code). **UNFUG** may be used recursively and multiple nested loops are supported. Full `LISP` functionality is accessible for code generation using balanced "<(" and ")>". A typical application of **UNFUG** is the generation of a set of specific functions from a generic function template and a tuple holding the specific information.

4.3 External code generators

External code generators (like `yacc` and `lex`) are directly supported by the two rules `Yacc-Target` and `Lex-Target`. The code generator used should be available on all supported platforms of the project (if not, the generated code is copied to a release too, but cannot be regenerated from a modified source later). The generated files are protected against modification by making them read–only. Also the output base filename is the same as the input file to avoid name conflicts with multiple generated parsers.

4.4 Software Installation

To build a software release, all modules of a project must be installed under an installation directory. As **VMake** knows all global include files, public libraries, and executable programs it can automatically put them into respective installation directories. Only for additional installable, otherwise unmanaged files (like `README` files and data files) a dedicated installation directory must be specified in the module description file. In addition to the installed files, **VMake** creates an installation project file which can be imported from other projects as a dependent project.

4.5 Release/Patch Generation

VMake supports source code level releases and patches between releases. The basic process is similar to the software installation, but a full second instance of the managed source code is created. For later patch generation a save file is generated with size/time information of the released files. For a patch this information is compared to the actual working information and used to find all changed, new and deleted files. The patch information is stored in the patch file, which can be applied by **VMake** to update a release by using the update option. During patch generation the save file and the release are updated. So in addition to the patch file, a full, patched release tree is generated.

4.6 Version Management Interface

VMake supports the *Concurrent Version System*[7], a public–domain version management system based on RCS[8]. **VMake** reads CVS' special files and upon request, prints lists of all source files modified with respect to the repository, and of all files not currently checked in, and of all files under control of CVS but not known to **VMake**. This automatism helps to detect and avoid version/configuration management inconsistencies in an early stage of the software production process (e.g., before the test phase). On request **VMake** asks the user to commit all changed source files (a modification description is required and put into the repository for logging), inserts new files into the repository and removes non–needed files due to changed sources. This feature is very useful when several people are working on the same or dependent projects to garantee a consistent repository.

5 Using **VMake**

VMake operates in both batch and interactive mode. The batch mode is similar to the make operation. After a build goal has been reached, **VMake** terminates. This is useful when **VMake** is controlled by another program, like automatic (overnight) scripts that check out the current version of a project, compile it, and perform all self–tests. The interactive mode is used by programmers during program development. After **VMake** is started and the global context file has been loaded, commands are read from `stdin` and are executed consecutively. If an error occurs, **VMake** reenters the interactive loop.

5.1 Operation

Similar to make, **VMake** uses a "depth first" algorithm[2]. **VMake** stores the source file meta data in a tree which is identical to the project directory tree. Each source file is a node in the tree. Include files, object files, libraries and executables are added in their respective position into the *dependency* tree. In contrast to make, **VMake** treats include files and all other implicitly managed files the same way as the files appearing explicitly in **VMake** rules. To verify a goal, **VMake** takes the list of all subgoals on which the current goal depends and recursively evaluates each subgoal (without creating a new process in contrast to make which creates for every directory

level at least one process). While checking the dependencies **VMake** executes any goal in background (automatic code generation, language binding, compilation, etc.) and parallelizes the build process over a workstation cluster (using remote shells under *UNIX* and the batch queuing system under *VMS*). The maximum number of jobs/workstation can be specified. This speeds up the build process significantly typically (by a factor of 4, using seven DEC Alpha workstations compared to local execution).

Since the entire dependency and actuality information is available during the evaluation of goals, **VMake** suffices with a single pass (in contrast to imake and make which usually require multiple runs to reach a certain build goal) and rebuilds only the required files. Due to this strategy, **VMake** is significantly more efficient than make, whenever the source code involved is spread around several different directories. In contrast to most make based configuration management systems, **VMake** uses only one process for all dependency checking and uses only a single subprocess per operation. This helps to save resources which is important for restrictive operating system configurations.

5.2 Rules and Goals

Rules contained in module description files declare source files and possible goals and define explicit relationships between all files of a module, as well as inter–module relationships. Despite the intentionally small set of rules offered, the implied semantics and eventually the actions taken by **VMake** are quite extensive. Currently **VMake** needs only 28 different rules to define all dependencies from sources to build goals. A build *goal* is either a single file (any object file, auto–generated source file, library, executable, self–test output file, etc.) which may be specified with its system dependent name or its symbolic name, or one of the more complex predefined goals, e.g. the well known goals *"all"* or *"clean"*.

6 Conclusion

Platform independence has been one of the major design requirements for **VMake**. Only an `ANSI C` compiler to compile the `LISP` interpreter is required for porting **VMake** to another platform. It is currently running on *VMS* (`VAX` and `ALPHA` systems) and 7 different *UNIX* systems.

The presented one-pass concept for building software projects, due to an open and extensible architecture and due to the consequent utilization of LISP–features, offers a functionality which goes beyond the scope of the sole building process. The integration of CASE utilities for source code verification, maintenance and formal verification is a straightforward task and yields a homogeneous tool which is centered around the classical "rule, target, and goal" philosophy of the make process.

6.1 Current Usage

VMake is currently used to manage the *Viennese Integrated System for Technology CAD* (`VISTA`[9]) which consists of about 18M–Byte source code (in `C`, `FORTRAN` and `LISP`), or almost 650K–Lines of code (75kLines thereof are generated automatically by VMake) by **UNFUG**[6] and **TAC**. Within `VISTA`, **VMake** manages 40 libraries and 30 executables in multiple dependent projects. Several foreign source code modules which have been contributed from different institutions have been successfully integrated in the portable build process.

Acknowledgements

The presented work has been sponsored by the research laboratories of AUSTRIA MIKRO SYSTEMS at Unterpremstätten, Austria; DIGITAL EQUIPMENT at Hudson, USA; HITACHI at Tokyo, Japan; MOTOROLA at Austin, USA; NATIONAL SEMICONDUCTOR at Santa Clara, USA; SIEMENS at Munich, FRG; and SONY at Atsugi, Japan, and by the "Forschungsförderungsfonds für die gewerbliche Wirtschaft", project 2/285 and project 2/299, as part of ADEQUAT (JESSI project BT1B), ESPRIT project 7236, and by ADEQUAT II (JESSI project BT11), ESPRIT project 8002.

References

[1] D. M. Betz,
XLISP: An Object-oriented LISP,
Version 2.0, Peterborough, NH, 1988.

[2] S. I. Feldman
Make — A Program for Maintaining Computer Programs Software — Practice and Experience, Vol.9, p.255–265, 1979

[3] T. Brunhoff, J. Folton
imake — C preprocessor interface to the make utility
X11R4, X11R5 and X11R6 release

[4] C. Seiwald
JAM — Make(1)redux Version 2.00
Jam(1) and Jambase(5) manual pages, Volume47, comp.sources.unix archive, 1995.

[5] Spectrum staff
The case for CASE tools
IEEE Spectrum, Vol.27, p.78–81, November 1990

[6] F. Fasching
Viennese Integrated System for Technology CAD Applications — Data Level Design and Implementation
Dissertation, TU Vienna, 1994
Österr. Kunst– und Kulturverlag, Vienna, ISBN 3–85437–093-8

[7] P. Cederqvist
Version Management with CVS
documentation for cvs 1.3.1, last updated 5.April 1993, FTP–able from ftp.think.com

[8] W. F. Tichy
RCS — A System for Version Control
Software — Practice and Experience, Vol.15(7), p.637–654, 1985

[9] S. Selberherr, F. Fasching, C. Fischer, S. Halama, H. Pimingstorfer, H. Read, H. Stippel, P. Verhas, K. Wimmer
The Viennese TCAD System
Proc.Int. Workshop on VLSI Process and Device Modeling, Oiso, Japan 1991