A Configuration Management Utility with CASE-Orientation

W. Tuppa and S. Selberherr
Institute for Microelectronics, TU Vienna
Gußhausstraße 27-29/E360, A-1040 Vienna, Austria/Europe
Phone: +43-1-58801/3680 Fax: +43-1-5059224
Email: 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. It supports, in addition to common make features, a number of CASE tasks like automatic code generation, version management (using RCS and CVS), and automated high-level source code processing features, like language bindings between C, FORTRAN and LISP. To foster re-use of source code modules, the proper modularization is verified by *VMake*. The tool 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 S.I.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. The description files for two prominent examples, a generic UNIX make, and the VMS MMS utility are shown in Fig. 1.

```
PROGRAM = myprog
MAIN    = mymain.o
OBJECTS = my1.o my2.o
LIBRARY = libmy.a
# generate program
$(PROGRAM): $(MAIN) $(LIBRARY)
      cc -o $(PROGRAM) $(MAIN)
$(LIBRARY)
# build library
$(LIBRARY): $(OBJECTS)
      ar -alcvs $(LIBRARY)
$(OBJECTS)
# additional dependencies
# generated by makedepend
mymain.c: myinc.h
my1.c: myinc.h
my2.c: myinc.h
```

```
PROGRAM = MYPROG.EXE
MAIN    = MYMAIN.OBJ
OBJECTS = MY1.OBJ,MY2.OBJ
LIBRARY = LIBMY.OLB
# generate program
$(PROGRAM) : $(MAIN) $(LIBRARY)
      LINK /EXE=$(PROGRAM)
$(MAIN),$(LIBRARY)
# build library
$(LIBRARY) : $(OBJECTS)
      LIB/CREATE $(LIBRARY) $(OBJECTS)
# additional dependencies
MYMAIN.C : MYINC.H
MY1.C : MYINC.H
MY2.C : MYINC.H
```

Fig. 1 Simple make and MMS description file

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 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 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.

```
SRCS  = main.c
        OBJS  = main.o
        LOBJS = my1.o my2.o my3.o
        DEPLIBS = libmy.a

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

Fig. 2 Standard entry in an Imakefile

Imake adds at least one additional 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). The reduction in size of the description files and the availability of global project information enables the management of larger software systems. However, the global information consisting of rules, system dependencies and definitions of global objects, 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. A common extended make implementation is

gmake from the GNU project[4]. Another way goes jam [5] , 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[6]. Several advanced commercial packages[7] 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.

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 per project global context file. This file is generated from the information of the local description files. 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 (only the names of required projects must be given in the top level description file of the project). Fig. 3 shows a *VMake* description file for the program shown in Fig. 1 and Fig. 2. Since *VMake* is based on LISP, the same syntax is chosen 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. 3 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. All files are accessed by their symbolic name, which has to be unique within all projects. In Fig. 3 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'. Finally the program is generated from the main object and the library. 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.
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 unneeded 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 unneeded 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 specifying 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.

2.1 Hiding System Dependencies

VMake encapsulates all system-dependent functionality in generic transformation functions which map the symbolic definition to actual compiler or linker calls, 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). To keep system-dependent files apart from the source code and to enable program building for multiple platform or project configurations within a single source code tree, *VMake* stores all compiled object files, libraries, executables, etc., in configuration-dependent subdirectories in the source code tree. Only source code files and configuration-independent code is stored in the actual source directories and shared between different configurations. Automatically generated code is marked read-only to prevent accidental changes by the programmer.

3. CASE Operations

3.1 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 between C and FORTRAN 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. 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. 4. 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.

```
/***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 :opt :key :default
                          strlen(str)] end index for
                          search */
{
  /* implementation of function */
}
```

Fig. 4 *TAC* documented function

In the example in Fig. 4, all parameters are used as input ([I]) and the parameter "str" may be given as NULL pointer ([IN]). To bind the function "myStrCount" to another language, the definition

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

is used in the description file of the module implementation. 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 LISP)
```

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 the 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".

3.2 Universal Function Generator

The Universal Function Generator (*UNFUG*) 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 compile-able output source code file in an arbitrary programming language (see Fig. 5). 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.

```
Template File               Tuple File                  Output File
/* start example */         (tupe                       /* start example */
<(UseTuple 'MyTuple)>       '(MyTuple(name value)       *first = 1;
 *<name> = <value>              ("first"    1)          *second = 2;
<EndTuple>                      ("second"   2)          *third = 3;
/* end of example */            ("third"    3)))        /* end of example */
```

Fig. 5 Example of UNFUG generated code

The *UNFUG* command <(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.

3.3 External code generators

External code generators (like yacc and lex) are directly supported by the two rules Yacc-Target and Lex-Target. 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 in the same directory.

3.4 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 rule must be used. In addition to the installed files, *VMake* creates an installation project file. The information of this file is used by *VMake* to get all required information about the installed project (in contrast to working projects) so that it can be used as dependency in working projects by different users.

3.5 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 is updated. So in addition to the patch file, a full, patched release tree is generated to allow generation of a full release file.

3.6 Version Management Interface

VMake supports the Concurrent Version System (CVS)[8], a public-domain version management system based on RCS[9]. *VMake* reads CVS's special files and upon request, prints a list of all source files modified with respect to the repository, and of all required files which are currently not checked in. This automatism helps to detect and avoid version/configuration management inconsistencies in an early stage of the software production process. The feature is useful when several people are working on the same or dependent projects to guarantee a consistent repository.

4. 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.
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.

4.1 Current Usage

VMake is currently used to manage the Viennese Integrated System for Technology CAD (VISTA[10,11]) which consists currently of about 18M-Byte source code (in C, FORTRAN and LISP), or almost 650K-Lines of code (75kLines thereof are generated automatically by **VMake**). Within VISTA, **VMake** manages 40 libraries and 30 executables. Several foreign source code modules have been successfully integrated in the portable build process which have been contributed from different institutions. In addition to avoiding tedious coding of multi-language programming interfaces, the **TAC** has proven as invaluable tool for the easy and smooth extension of the LISP interpreter. Moreover, the concise description files and the notions established by **VMake** enable the developer to grasp the organization and structure of projects more easily and to manage the build processes of even large projects effectively.

Acknowledgments

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, 9, 1979, 255-265

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

[4] R.M. Stallman and R. McGrath, "GNU make Version 3.63", Jan. 1994, FTP-able from prep.ai.mit.edu or mirrors

[5] C. Seiwald, "JAM - Make(1)redux} Version 2.00", Jam(1) and Jambase(5) manual pages, Volume 47, comp.sources.unix archive, 1995.

[6] C. Jones, "Case's missing elements", IEEE Spectrum, 29, 1992, 38-41

[7] Spectrum Staff, "The case for CASE tools", IEEE Spectrum, 27, 1990, 78-81

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

[9] W.F.Tichy, "RCS - A System for Version Control", Software - Practise and Experience, 15(7), 1985, 637-654

[10] S.Selberherr and F.Fasching and C.Fischer and S.Halama and H.Pimingsdorfer and H.Read and H.Stippel and P.Verhas and K.Wimmer, "The Viennese TCAD System", Proc. Int. Workshop on VLSI Process and Device Modelling, 1991

[11] S.Halama, "The Viennese Integrated System for Technology Applications, Architecture and Critical Components", (Vienna: Österr. Kunst- und Kulturverlag, 1995)