Proceedings of the IASTED International Conference
Artificial Intelligence, Expert Systems, and Neural Networks
August 19-21, 1996 - Honolulu, Hawaii, USA

A CASE–Oriented Configuration Management Agent

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
http://www.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 agent, the Viennese Make (VMake) is platform-independent and runs currently on a number of *UNIX* systems and on *OpenVMS*. VMake supports, in addition to common make features, a number of CASE tasks in sub–agents, like automatic code generation, version management (using publically available RCS and CVS from a common repository), and automated high-level source code processing features, like language bindings between C, LISP and FORTRAN as well as 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 efficient management of large–scale software projects. VMake is based on a publically available LISP interpreter[1].**

Keywords— **configuration management, automatic code generation, parallelization, CASE, object oriented techniques**

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 upon 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. make itself is directory oriented, e.g. a new evocation is needed for any recursion or use of subdirectories. There is no information exchange between different evocations of make.

For configuration management of the well known large–scale X Window 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 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 size reduction 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 and programs), is contained in a separate set of files, which have to be provided and maintained manually by the software engineer.

Other approaches extend make by directly adding new functionality like multiple goal evaluation in parallel or inclusion of sub–description files (for commonly used data and global information similar to imake). A common freely available extended make implementation is gmake from the GNU project. All advanced approaches based on make are still directory oriented and use a recursive evocation for subdirectories. All global data has still to be maintained manually for each system. Jam[4] goes with a different solution. It reads all description files in a directory tree (and so avoids the problems of being directory oriented and multiple scans of the same description files), which are based on a simple language by its own, to generate a full dependency tree on every invocation and then builds all goals in a second pass. In this case all information is handled by one evocation of jam.

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 Vienna Make

The Vienna Make (**VMake**) employs a small number of standardized high-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 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 from the project description file in the top level directory of a working project. Only the *symbolic names* of required projects must be given in this project description file. Those projects can be either working projects with source available or projects which have been globally installed by **VMake** (see 3.4) earlier.

Fig. 1 shows an example **VMake** description file for a small program in a working project. 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. 1. 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 generated 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. 1 the symbol *My-C-Main* is bound to the system specific file name. 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 with `Program-Target` the executable program. The generated executable 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.

2.1 Algorithm

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 files are rebuilt. **VMake** exploits the fine–grained global dependency information (over directory and project boundaries) to rebuild an utmost concise superset of the really required files. Internally **VMake** identifies each file by a unique LISP object which stores all available information about that file. The main agent checks the dependencies between objects (and possibly rebuilds the dependencies from modified source files) in a "depth first" algorithm similar to make and sends a rebuild request to the checked objects if required. If execution of a goal is required, the appropriate command is inserted into the sub–agent of **VMake** for later execution which occurs in parallel on a defined workstation cluster. This speeds up the build process significantly (by a factor of four, using seven DEC Alpha workstations compared to local execution). In contrast to make **VMake** needs no recursive evocation and solves the goal evaluation in one pass since all information is available in the main agent. Due to this strategy, **VMake** is significantly more efficient than make whenever the source code involved is spread around several different directories and/or projects. This helps to save resources which is important for restrictive operating system configurations.

Dependencies are stored in **VMake** per directly included source files only (in contrast to make where all include files over all levels are needed as dependencies). This speeds up the dependency generation process since no files are multiply scanned during dependency generation and so allows dependency generation on the fly on every run of **VMake**. The automatically extracted global information about projects is saved by **VMake** for every configuration in a special place to be reloaded on the next evocation.

2.2 The Queuing Agent

This sub–agent of **VMake** is activated by objects if they need to be rebuilt. In this case the queuing agent checks if all required inputs of a goal are already available and then queues it for building else it delays this job back until all required objects are built. Objects ready for building are executed in parallel on the defined hosts and/or batch queues for the actual configuration. The main agent can request information about the current status of the queuing agent.

3 CASE Operations

VMake has built-in support with sub-agents for different CASE-operations needed by large-scale software projects.

3.1 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. 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.

3.2 Tool Abstraction Concept

VMake uses a *Tool Abstraction Concept* (**TAC**) for generating language bindings of functional modules and constants for different programming languages. 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*, *OpenVMS* has a totally different handling of strings (especially using `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. 2. Currently, bindings can be generated from `C` to `FORTRAN`, from `FORTRAN` to `C`, and from `C` to `LISP`. The comment `/***TF` starts the definition of a TAC-able Function. The comments following the function arguments consist of a formal description of the argument characteristics and a textual documentation part which is used for the documentation extraction facility.
In the example shown in Fig. 2, all parameters are used as input `[I]` and `str` may be a `NULL` pointer `[IN]`. To bind the function `myStrCount` to another

```
/***TF [:reference myStrChar myStrReverseChar]
    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. */
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. 2. **TAC** documented function

language a **TAC** definition interface rule must be used in the **VMake** description file so that it can be used later with the create interface rule to bind the function to a different target domain (e.g. `LISP` or `FORTRAN`). A stub code in `C` is generated from the extracted information to generate the language binding between source and target language. This stub code then needs to be linked to the application.

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

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 code generator used should be available on all supported platforms of the project. For release generation the automatically generated code is copied into the release in case the external code generator is not available on the target platform.

3.4 Project Installation

To build a software release, all modules of a project must be installed under a global 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, otherwise unmanaged files (like `README` 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. This file holds all dependency and installation information required for use by **VMake** as an installed project. In contrast standard tools like make usually copy all files every time and do not create an information file. One additional feature is that only modified files, especially include files, are installed if an older release already

exists. So only changed sources get new time stamps and again only a minimal set of files has to be rebuilt in the working project. Files that are no longer required in an installation are automatically removed from the global directory.

3.5 Project Release

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 generating a patch this information is compared with the actual state of the working project and is used to find all changed, new and deleted files. The patch information is stored in the patch data 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.

3.6 Repository Interface

VMake supports the *Concurrent Version System*[7], a public-domain version management system based on RCS[8]. **VMake** reads CVS special files and informs the developer of modified, added and removed files in the working version. When checking in files into the repository modification comments to all modified files are inquired from the programmer and the repository is brought up-to-date. This mechanism helps to detect and avoid version/configuration management inconsistencies in an early stage of the software production process (e.g., before the test phase). To allow easy update from the repository **VMake** is able to update on project basis from the repository. The developer is informed of updated files and a warning is issued for every conflict between local modifications and the repository state.

4 Interactive Usage

In addition to the batch mode similar to make **VMake** supports an interactive mode where once started the main agents enters an interactive loop. Some state variables of **VMake**, like the actual goal, working project, or flags can be modified and then the goal evaluation can be started. On error **VMake** stops the goal evaluation and reenters the interactive loop.

In addition to the interactive command line mode an interface to Emacs has been written to allow running **VMake** as a subprocess within in Emacs and analyze the output of **VMake** by the editor, e.g. to jump to lines with error messages from the compilations (similar to the make mode of Emacs).

5 Conclusion

Platform independence has been one of the major design requirements for **VMake**. Only an ANSI C compiler for generating the LISP interpreter is required for porting **VMake** to another platform. It is currently running on *OpenVMS* (Vax and Alpha systems) and 8 different *UNIX* platforms. 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.

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, 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