

# Architecture for a software construction system

*QEF software process automation system*

White Paper

by David Tilbrook, B.Sc., M.Sc.

Copyrights apply. All rights reserved.  
Qef is a registered trademark of qef.  
Content copyright David Tilbrook, B.Sc., M.Sc.  
This revision prepared and edited by Cleanscape Software International, Brent Duncan  
Director, with permission of author: bd  
Other trademarks and copyrights may apply.

*Architecture for a software construction system: QEF software process automation system:*  
*White Paper*  
Rev. 2 - 5.18.01 bd  
April 3, 2001

**Comments to:**  
Cleanscape Software International  
1530 Meridian Ave, Suite 200, San Jose, CA 95125  
Tel: (408) 978-7000 Fax: (408) 978-7002  
E-mail: [qef@cleanscape.net](mailto:qef@cleanscape.net)

## Contents

<b>1</b>	<b>Abstract.....</b>	<b>1</b>
<b>2</b>	<b>Introduction.....</b>	<b>1</b>
<b>3</b>	<b>Architecture.....</b>	<b>4</b>
	The Parameter/Configuration Database Construction.....	4
	Script Preparation.....	5
	Back-End Interpretation.....	5
	The Commonest Implementations.....	5
<b>4</b>	<b>Features.....</b>	<b>6</b>
	lclvrs.....	6
	Script Generation.....	7
	The Preprocessor -- qefpp().....	8
<b>5</b>	<b>The Back-Ends.....</b>	<b>9</b>
	QSH.....	9
	MIMK and QMK.....	9
<b>6</b>	<b>The QEF Software Process.....</b>	<b>10</b>
<b>7</b>	<b>QEF Drawbacks.....</b>	<b>10</b>
<b>8</b>	<b>Conclusion.....</b>	<b>11</b>



# Architecture for a software construction system

## *QEF software process automation system*

*By David Tilbrook, B.Sc., M.Sc.*

## 1 Abstract

**Most** organizations do not have a comprehensive or effective approach to the management of the software construction process. This paper describes *QEF* (Quod Erat Faciendum, that which was to be made), a system for large-scale software construction developed by the author and in use at a number of major organizations. The rationale, strengths and weaknesses of the architecture are discussed.

## 2 Introduction

**Many** organizations have a great deal of difficulty actually producing their software products from source. Furthermore, most organizations do not have a comprehensive approach for the porting of their product to new platforms, the safe development of their product without interfering with production or other developers, the management and fixing of old releases, or the simultaneous use of multiple releases on the same platform.

Measuring the cost of the lack of these capabilities is virtually impossible. But it is the author's belief and experience that the reliable, timely and efficient approach to the management of the software construction process in a way that promotes an effective development process can yield substantial benefits both in terms of decreased costs and increased quality.

This paper describes the architecture of the *qef* system. *qef* is a family of tools used to manage processes typically used to do software construction. In the beginning, circa 1983, *qef* was developed to provide a mechanism whereby the construction of software products could be described in a host-independent way. Over the years it has been enhanced to encourage and support a better software process and a comprehensive approach to software development and maintenance and release engineering.

In hindsight, the major objectives of the *qef* implementation were to achieve the following:

- **Simplified Expression of the Construction**  
The expression of the construction process as provided by the user should be as simple as possible. Simple statements of what is to be done should be all that is required.
- **Effective Full and Incremental Constructions**  
The system must ensure correct, accurate and efficient construction of the products. Furthermore, the system must ensure that an incremental build (i.e., using previously built temporary files) ensures consistency between the resulting product and which that would be built by a full construction.
- **Effective Parameterization/Configuration Mechanism**  
Comprehensive mechanisms must be provided to specify, test and use parameters and configuration controls throughout the construction and software process.
- **Effective Process Control and Monitoring**  
Mechanisms to launch and monitor the construction process must be provided.
- **Promote an Effective Software Process**  
The system must support and encourage the use of better software development strategies throughout the software life cycle.

Whereas the objectives of the *qef* implementation and research were to achieve the above, one must examine at length the major objective or purpose of a construction system. A construction system manages the processes that transform source into a product. The organizations that can produce their products from source, and only source, reliably, efficiently and on demand are in a very small minority. The construction system must provide the mechanisms whereby the ability to produce the product from the source is almost taken for granted, but can be tested easily, cheaply and frequently.

An important implicit assumption of this objective is that if the construction system is applied a second time to the same sources, it will produce the same product. Unfortunately even fewer organizations can reproduce the identical product if the source is relocated or some other aspect of the run-time environment has changed (e.g., a different user with a different environment).

The construction system should help ensure that the ability to reproduce the product depends as little as possible on things that are not administered source.

Furthermore, the construction system must provide an incremental build facility; that is, given that some or all of the product files and intermediate build-files exist, the system skips processes that are not required to make the product identical to the version that would be produced by running the entire production process from source alone.

In meeting these stated objectives a number of characteristics and features have evolved as being of paramount importance:

- **Portable**  
The user-provided construction system input must be the same for all systems.
- **Flexible**  
The construction system must support the use of the most appropriate tool. The system should have a simple and consistent way to specify the tool to be run and its input.

- **Complete and Comprehensive**  
The construction system must facilitate and encourage the provision of the complete set of construction options. The construction system must also provide mechanisms whereby all aspects of the system are managed using the same mechanism in a consistent manner.
- **Dynamic and Responsive**  
Managing software construction is about responding to change. A construction system must respond appropriately to changes in the file system, the files themselves and the build parameters. The appearance or disappearance of a file should be appropriately reflected in the construction commands. Changes in the files that would be found by various search paths (e.g., the `cc -I` flag) should result in the appropriate recalibration of the dependencies.
- **Consistent**  
The construction system should present a consistent and simple interface across all platforms, projects, and users. Constructing a software product should not require any special knowledge.
- **Clear and Concise**  
The actual processing involved in a construction can be described very simply, but comprehensive and complete *make* files can run to hundreds of lines. The system should derive everything else required automatically, while still providing simple mechanisms to override defaults in the general and specific cases.
- **Simple**  
This is indeed one of the most difficult characteristics to achieve -- software construction is exceedingly complex -- but must be if the system is to be accepted and used successfully.
- **Highly and easily configurable**  
When dealing with software that is to be built on many different platforms by many different users, the number of configuration parameters that can be required is staggering. Most projects will require scores of parameters to specify construction controls, names of tools, tool flags, search paths, special directories and files, and other build and semantic options and controls. The construction system must provide a comprehensive, universal, debuggable, and extendible mechanisms to specify, test, manipulate, and use parameters. Furthermore, it is imperative that these parameters be available to any tool -- not just the construction system. Note: the C preprocessor is not such a mechanism!
- **Facilitates and encourages complete source and object separation**  
It is absolutely essential that a construction system support and encourage the total separation of the master source, source under development, and object files. There should be one and only one true read-only source file system which is shared by all users. Files to be modified should be copied to and changed in a user's own work space, thereby ensuring that their changes are not visible to others until ready and authorized. It should be trivial to create and link the separate directories. Furthermore, it is essential that the source path mechanism be usable by any application (e.g., the version system).

- **Provides suitable controls**

A construction system must provide a simple consistent mechanism to manage large trees of directories and to support the constructions of part or all of those trees. Furthermore, it the construction system should provide mechanisms to monitor its progress and to halt it.

### 3 Architecture

As a construction system must be oriented towards the management of the construction of large complex systems on multiple platforms simultaneously, a major feature of the system is dealing with the complexity of the construction process. *qef* uses a structured, layered approach to deal with this complexity. In many ways, this architecture parallels the use of structured programming and high-level programming languages to deal with and create large programs. Abstraction and information-hiding is used to control the amount of information that one must deal with at any level.

In some ways, *qef* is similar to *make*. There is a text file that contains the construction control script; the user invokes *qef* with arguments specifying files or constructs to be created; ultimately, commands are invoked, often by a *make*-like process, to create the required objects or perform the required tasks.

As such, *qef* can be viewed as a replacement for *make*. However, the most important features of *qef* are for configuring and controlling processes and preparing the input for the back-end.

Thus *qef* is primarily a driver of other processes. Rather than attempting to solve all the problems using a monolithic program, *qef* provides facilities and structures to select, configure and control other processes. As will be seen, this approach provides flexibility in configuring the construction process, while ensuring that there is a single universal interface.

*qef*'s processing is roughly divided into three stages: construction of the build parameters/configuration database, script preparation, and back-end processing.

#### The Parameter/Configuration Database Construction

The first stage invokes a program called *lclvrs*, which prepares a database of the build parameters and configuration information for use by other programs. The information is represented in a hierarchical database at the top of the current tree and distributed through the source tree. Configuration files at a particular level of the tree apply to all sublevels of the tree. This parallels the lexical scoping used in programming languages -- configuration information is only visible where it is required.

*lclvrs* finds and processes the relevant files for the current directory and outputs a binary file that can be directly loaded by other programs to retrieve the various parameters, options and controls provided via the *lclvrs* files. Parameters are used to specify search paths, process controls, build options, the names of special files and directories, tool names and flags, and so on. In this and other documents the convention used to specify the use of a *lclvrs* variable's value is either `@Variable` or `@Array[value]`, "@" being the *lclvrs* precursor variable escape.

## Script Preparation

The major purpose of the script preparation stage is to transform an as simple as possible specification of the processing to be done, into the back-end command to be run and the input to that command. This transformation can range from the naming of a back-end process and its input file via `lclvrs` parameters, to the more common three-stage process of the creation of a list of source files, script generation using the source list as arguments or input, and the macro processing of the generated script. Practically any command may serve as a script generator, but two programs, `qsg` and `qefdirs`, are used most of the time. `qsg`, `qefdirs` and the macro processor are described briefly at a later point.

## Back-End Interpretation

The third stage is the back-end which usually does the real work. In most instances, this will be a shell or *make*-like program. Some back-ends that are specifically designed for use by `qef` are discussed in later sections. The actual back-end to be run is specified by `lclvrs` variable or a preprocessor symbol.

## The Commonest Implementations

While this architecture allows a wide range of processing models, in practice, two models are used in the majority of directories that contain conventional processing.

In directories of directories the user provides a list of the directories that contain constructions, the type of processing the directories support, and their dependencies on other directories. The list may also partition sets of directories into individually selectable sets. The script preparation stage invokes the program `qefdirs`, which transforms this information into a *make*-like script, which provides labels (i.e., targets) to perform aggregate operations (e.g., All, Install, Post, Test), or processing for named directories or sets of directories. The recipes for an operation are usually just to invoke `qef` in the named directory with the appropriate argument.

The most common model (used in approximately 75% of 1200 directories examined in various Toronto sites) is used for directories that contain source files to be processed. Once the configuration database has been assembled, a snapshot of the file system is generated. This generation uses configuration information to determine the search paths to be used and the suffixes of relevant files. The source database is typically input to the script generation stage.

That this database is created at the beginning of processing is significant. It has benefits both in terms of debugging the construction as the initial conditions are preserved and it is also efficient. Tools that combine view-pathing and rule-inference result in many unnecessary accesses to the file-system meta-data.

The script preparation is done using `qsg`, an algorithmic programming language. The configuration and source databases, plus a `qsg` script, are processed to generate the necessary recipes to do the construction. `qsg`'s output is processed then by the macro processor, the output of which is sent as input to the back-end `qmk`, a *make*-like program described at a later point.

Although the above may appear complicated, most users are unaware of the actual processing. A `qef`file that invokes the above is often as simple as:

```
Begin qsg -M  
commands @argv
```

This example is far-fetched; the average size of the 1,200-sample *qef* files was seven lines.

## 4 Features

This section describes some of the more important features of the major *qef* tools.

### lclvrs

lclvrs is run as part of *qef*'s initialization to find and interpret the lclvrs files for the current directory, and build a temporary database of variable/value pairs that other tools will access directly. lclvrs files are used to specify process controls, build options, the names of special files and directories, search paths, tool names and flags, special library mappings and search mechanisms, and other such values. lclvrs provides a variety of options to select the form of the output, list the selected files, report where specified variables are set, or to output special values.

In most situations, lclvrs will find and interpret: *root.vrs* at the top of the current tree -- provided by developer (using the program *rootvrs*) to indicate the root of a tree and specify links to other trees; *config.vrs* at the top of the current tree -- copied from annotated prototype and modified by developer as required -- used to contain all configuration settings and user options; *tree.vrs* at the top of the master tree -- used to specify project parameters and controls; *hostsys.vrs* a host provided file containing host-specific controls such as library mappings and search paths; and *qeffile*, in the corresponding source directory -- the directory specific settings and local construction controls. The *qeffile* also usually contains the input to the script generator or the back-end.

In addition to the above, a user can create files called *leaf.vrs* and *branch.vrs* in the current tree to specify temporary lclvrs settings without changing source files. lclvrs settings can also be specified via the *qef -L* flag. This feature is often used to specify the type of binaries to be produced as in:

```
qef -LDEBUGGING echo # produce "debuggable" program  
qef -LPROFILING echo # produce "profiling" program
```

The *hostsys.vrs* file will set the C flags and library search mechanisms as required to create the requested type of binary.

The lclvrs language provides simple keywords to set, manipulate and test variables, paths and associative array elements, to do flow control ("if" and "switch" -- no loops), and output fatal or non-fatal diagnostics. lclvrs also offers a set of functions that can be used in keyword argument lists to test and manipulate strings and/or variables, search or test for files, and to retrieve user or environment information.

One of the most important *lclvrs* features is that it is a separate program whose sole purpose is to process the configuration files and prepare data for other processes use. Any tool that requires *lclvrs* settings either executes *lclvrs* or reads a previously prepared *lclvrs* output. Configuration information need not be passed via command line arguments or environment variables thereby ensuring consistent application of the tool no matter through whatever invocation mechanism was used.

## Script Generation

*qef*'s ability to invoke an arbitrary shell command coupled with the preprocessor to prepare the input to an arbitrary back-end is an extremely powerful and important feature that has been a crucial component of the *qef* system since its birth (see [Tilbrook 86]). In *qef*'s early years there were a large number of special purpose script generators but all but one (*qefdirs*) have been replaced or superceded by *qsg*, which is used about 80% of the time.

*qsg* is a fairly simple programming language, specifically designed to create input for other processors from simple shell-like commands. Because *qsg* is used to generate data to be read by arbitrary languages it has a number of lexical characteristics such as minimal quoting -- consider as a contrary example the difficulty of running *awk* from *tcl*. For performance reasons it is compiled to a very efficient intermediate representation either at run-time or to prepare object files for possible inclusion in a library of *qsg* scripts. The library mechanism facilitates provision of a rich set of standard procedures. These standard library functions can, for most constructions, reduce the entire build specifications as provided by the user to simple host-independent single-line commands.

The set of *qsg* library scripts deliver the type of functionality that make inference rules are intended to supply. The actual construction details are hidden in the library allowing the end-user to trivially create portable construction specifications.

*qsg* commands consist of a keyword or the name of a procedure, file or *qsg* library member, and an argument list. The keywords provide flow and I/O control, variable manipulation, procedure and flag definition, and debugging controls. Argument lists are simply strings that may incorporate the values of variables and/or functions. The value of a variable is used by `@' followed by the variable's name as in:

```
# invoke "commands" script for arguments specified by "argv"  
commands @argv
```

Functions are provided to read files or pipes, evaluate *lclvrs* expressions, find or check files, etc. Function calls look like "@(function flags args ...)" as in:

```
# a read line from previously opened file or pipe  
set var @(readline fid)
```

Various manipulations of a variable or function value can be performed using tilde postfix operators. For example, "@argv\*" is replaced by the number of elements in the variable *argv*. Tilde operators are provided to select matched elements, selected parts of individual elements (e.g., the directory name, tail, suffix), or to perform substring replacements. However, the normal user uses a very small subset of *qsg*'s facilities when creating a *qeffile*. Most *qeffiles* will consist of a few invocations of the *qsg* library members as in:

```
# install *.dat source files in directory _DestDir_/lib
install -d _DestDir_/lib @argv*x/dat/

# compile and link example.c and parser.y program example.c parser.y

# create and install dtree library with a version module library -v -n dtree @argv-x/c.y.l/
```

In the above "install", "program", and "library" are qsg scripts that have been compiled into the qsg pseudo-code and installed in an archive. A typical qsg library script will produce output in a form suitable for the chosen back-end. The qsg scripts are built to deal with host-dependent conventions, although in many cases they output qefpp macros to deal with host dependent names and mappings.

An example of qsg output would be of limited value. "program file.c" will generate 30 to 60 lines, depending on the host. The output would contain recipes and dependencies to produce the program, the installed program, the object modules, the assembler versions, the purify, quantify, and pixie versions (if supported), to lint file.c, and to remove any intermediate and/or installed files.

One of the significant advantages of this approach, when compared with *make's* inference/suffix rules or *imake's* macros, is that qsg creates scripts that provide complete and comprehensive facilities from succinct portable specifications. Many qeffiles consist of one or two qsg commands, yet the equivalent *make* scripts are huge. In one of the author's source directories (containing 70 c, lex, yacc, and shell programs) a three-line qsg script produces the equivalent of a 3,500 line *make* script.

## The Preprocessor -- qefpp()

qefpp()'s primary role is much the same as the C preprocessor -- to define macros and replace embedded instances of those macros by their values and to provide primitive flow control.

However, qefpp() offers a number of important features far beyond cpp's capabilities as illustrated by part of qsg's output for "program eg.c":

```
eg: _Touch_(cc) _Libs_( _FndSrc_(eg.c) eg.o _T_cc _F_cc _F_cc_o _F_cc_o[eg] eg.o _CcLibs_() -o eg
```

The above uses the following qefpp built-in macros:

- `_Touch_(cc)` replaced by list of files called `cc` in the `@TouchPath` directories. Thus the target file "eg" is dependent on any existing "Touch cc" file. To force the recompilation of all C sources, one touches a file called "cc" in a `@TouchPath` directories. The script generators output these Touch dependencies for every significant tool thereby providing a simple and consistent way to force the re-execution of processes.
- `_FndSrc_(eg.c)` qefpp searches the `@SrcPath` directories for the first instance of a file called "eg.c" and replaces the macro by its path.
- `_Libs_(eg.c)` Replaced by the libraries for file `eg.c`. A variety of configuration parameters are used to create this list. The `"_CcLibs_0"` is almost the same except that the list of libraries might be modified to be suitable for a `cc` command.
- `_T_cc, _F_cc, _F_cc_o[eg]` Symbols beginning with `"_T_"` and `"_F_"` are for tool names and flags respectively. Such symbols are treated specially in that they are

automatically imported from the *lclvrs* database as predefined symbols and also have special default values (the symbol name minus the "\_T\_" prefix for \_T\_ symbols, the empty string for \_F\_). Also note that associated array elements are support to allow the specification of tool flags for specific files.

In the initial implementation of *qef* (circa 1983) the preprocessor played a much more important role. Its importance has been greatly reduced as the script generators and *lclvrs* now provide better mechanisms for managing variables and special constructions. However, it still serves to perform simple macro processing and variable substitution, which greatly reduces the complexity of the script generation process.

## 5 The Back-Ends

**Almost** any non-interactive program may be used as a *qef* back-end. As such *qef* has been used for a wide range of applications, not just software construction (e.g., driving *empire*). Using standard tools (e.g., *make*, *sh*, *tar*, *rdist*) is not uncommon. In some instances programs have been created to be used as *qef* back-end for a single application. However, three programs (*qsh*, *mimk*, and *qmk*), were created to be general purpose *qef* back-ends and one of them is used in most directories.

### QSH

*qsh* is just a much reduced shell command interpreter, with features to control command echo and exit on failure. It also monitors the *qef* halt file so as to stop processing when required. *qsh* is used when the required constructions are unconditional or can be selected by the script generator (i.e., *make* would just get in the way).

### MIMK and QMK

For the first 4 years, *qef* used *make* as its primary back-end, but that caused a number of problems. *make* is not standard (implementations vary greatly); does not provide the required consistency mechanisms; does not provide some of the required semantics such as monitoring the *qef* halt file; and cannot run recipes in parallel.

*mimk* was developed in the mid-80s to be the *qef*'s engine of consistency. *mimk* is based on *make* with extensions to provide recipe and dependency attributes, improved consistency checking and parallel recipe executions. *mimk* was designed to be a back-end for *qef*. As such it does not need some of *make*'s facilities. It does not have suffix rules -- the script generators provide complete recipes. It does not have variables -- *qefpp* does it all.

However, *mimk* has its own limitations and is being replaced by *qmk*. *qmk* is *make/mimk*-like in purpose. However, its syntax has been extended to provide for extended control and attribute semantics and easier generation by *qsg*. The major reason for its creation is to provide an improved consistency checker and lazy dynamic dependency list generation.

## 6 The *QEF* Software Process

One of the major objectives listed in the first section was the promotion of an effective software process. *qef* does not require any particular process methodology -- users may use *qef* in much the same way they would use *make*. However, a number of tools and features have been implemented to support the total separation of the source and object trees and source view pathing. These mechanisms allow multiple independent constructions to share a common source tree. Furthermore each build can specify the optional insertion of a user's own source override tree to contain those files modified by the user.

A typical *qef* using site will set into place procedures whereby someone (e.g., a librarian) will be responsible for maintaining a version of the product that is consistent with a master source tree. This responsibility is usually fulfilled by running *qef* at the top of an object tree for each required platform and checking the results on a daily basis. These product trees will be searched for any product files that the developers need during construction that are not being built by the developers themselves. Meanwhile developers make their source modifications to files copied to or created in their own source trees, thereby protecting the master source users and other developers from their changes. Developers will then build a version of the product incorporating their modifications in their own directories. The simple configuration mechanism, the run-time script generation and the comprehensive consistency mechanism ensures that the constructed products are virtually identical to that that would be built from the current master sources if it incorporated the modified files.

Some mechanisms are required to facilitate interfacing to the versioning system being used (e.g., to check files in and out from a remote directory) and some procedures or policies are necessary to ensure that librarian has the required control over the publication process, but both are fairly easily created for most versioning systems.

One of the benefits of the *qef* system is that the construction scripts themselves, due to their simplicity and portability and run-time processing, rarely need modification by the developers to incorporate new constructions. If modifications are required, they will normally be platform independent thereby eliminating any need for extensive work by the system librarian to adapt to those platforms that the developer did not test.

Adaptation of such a scheme is not necessary to take advantage of *qef*'s benefits, but those organizations that have used such an approach do claim substantial rewards, albeit subjectively.

## 7 *QEF* Drawbacks

There may be a small performance penalty using *qef*. The overhead of running `lclvrs`, `sls`, `qsg`, and `qefpp0` is not negligible but not excessive -- less than a second per directory on a 486, `sgi`, `sparc`, or `rs6000`. But many of the scripts and tools use techniques to avoid gratuitous time stamp propagation, thus in many situations *qef* avoids work that other systems would perform. Furthermore, the parallel process facility and other features actually reduce the build cycle time. One site that audits build cycle times reported a 66% reduction in the time taken to build their product.

Gaining acceptance from developers is sometimes difficult due to their resistance to change and their suspicion of radically different or novel approaches. Furthermore, a lot of the problems *qef* solves are problems that do not concern programmers or ones that they don't believe they have. There was a similar resistance to *make* initially. However, when challenged on programmer acceptance of *qef*, the author asked a user how he would feel about going back to using *make*. His reply was that it would be similar to going back to DOS.

One of the necessary characteristics for a construction system that was listed in the first section was that the system needed to be simple. *make* is fairly simple in its syntax and semantics, but the *make* files themselves and the actual use of *make* to control large scale projects can be exceedingly complex. *qef* uses a collection of small languages which may make it seem complicated. Indeed, *qef*'s processing is complex, but its complexity is largely hidden from its users. The greatest difficulty in promoting *qef* is presenting it to an audience that currently uses *make*. Programmers are notoriously reactionary and resistant to change. They expect to understand *qef* sufficiently to deal with complex problems in the first ten minutes and forget how long it took them to achieve a similar understanding of *make*. *qef* is a complex system, although the individual components are simple and the *qef* files are invariably tiny. The complexity lies in their combination to achieve the other requirements.

## 8 Conclusion

The major differences between *qef* and other construction systems is the separation of the parameters system (i.e., *lclvrs*), the flexible run-time script generation and preparation and the ability to specify the use of arbitrary back-end interpreters. While these unconventional approaches and the unfamiliar representations do present marketing problems, the resulting system does meet the objectives discussed in the first section. Testimonials from satisfied and enthusiastic users could be presented as a more objective defense of this claim, but will not be in the interest of ensuring that this paper does not appear to be a marketing document.

## 9 Bibliography

- Feldman 78 S.I. Feldman, *Make -- A Program for Maintaining Computer Programs*, Unix Programmer's Manual, volume 2A, Seventh Edition, January, 1979; Bell Laboratories, N.J.
- Stenning 89 Vic Stenning, *Project Hygiene*, Usenix Software Management Workshop, New Orleans, 1989.
- Tilbrook 86 D.M.Tilbrook and P.R.H. Place, *Tools for the Maintenance and Installation of a Large Software Distribution*, EUUG Florence Conference Proceedings, April, 1986, USENIX Atlanta Conference Proceedings, June 1986.

Tilbrook 90 David Tilbrook and John McMullen, Washing Behind Your Ears -- or -- The Principles of Software Hygiene, Keynote address, EurOpen Fall Conference, Nice, 1990.