Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

THE RUNTIME CREATION OF CODE FOR PRINTING SIMULATION OUTPUT

John H. Reynolds

Computer Science Department
Mary Washington College
Fredericksburg, Virginia 22401-5358

ABSTRACT

Historically, the handling of output requirements for large simulations has been one of annoyance and inconvenience for developers. The unnecessary waste of computer resources, as well as manpower, continues to mount as larger and larger simulations are created.

The Automated Code Handler for Output Operations (ACHOO) is a tool that supports a methodology whereby output requirements for a computer program, and the source code for the statements to support them, are determined at execution time based on user-supplied input. Thus, the user can tailor output requirements to a specific need or scenario. The function of the programmer is reduced to instrumenting pre-designated areas where potential monitoring of program behavior is desired. This instrumentation activity, consisting of inserting calls to subroutines that ACHOO creates, eliminates the need for manual creation of actual output statements and accompanying format specifications during computer program development and subsequent maintenance.

Even though ACHOO, in its present configuration, supports FORTRAN programs only, this does not preclude its adaptation to other languages. During design, attempts were made to identify, isolate, and document language-dependent functions so that module substitution could occur to accommodate other languages.

ACHOO—coded in SIMSCRIPT II.5—was developed by the author during consulting activities within the System Simulation Branch of the Strategic Systems Department at the Naval Surface Warfare Center (NSWC) in Dahlgren, VA.

1. INTRODUCTION

In today's world of software engineering, heavy emphasis is placed on tools, i.e., computer programs that aid in developing and/or executing other computer programs. This emphasis is in recognition of the fact that many of the manual, time consuming, and error prone tasks performed in the past can best be done by a computer. Some of the characteristics of these tools are such that they may: (1) yield a reduction in the time and cost to produce software, (2) reduce the execution overhead of a model both in terms of memory and central processor time, (3) simplify initial computer program development and subsequent life cycle maintenance, (4) be reused across different models, (5) provide the user population with better methods for interfacing with their models by offering standardized processing methodologies.

Since 1976, the Systems Simulation Branch at NSWC has been using modern software engineering practices for developing large scale FORTRAN simulations, the first of which was the TRIDENT Computational Simulation (TRICS) [Reynolds 1980]. During the TRICS effort, some tools meeting the criteria stated above were created. In particular, a general purpose input processor was developed [Goyette and Owens 1982; Reynolds 1980] along with the notion of a Configuration Description Language (CDL) and supporting processor [Lemoine 1978; Reynolds 1980].

The basic premise behind these earlier tools is to delay until runtime the creation of source code, normally written by programmers, by making it dependent upon specific user requirements. To digress, the CDL approach permits any commitment to a particular scenario or model configuration to be delayed until runtime. This also implies that the source code representing the controlling executive (main program) does not exist prior to execution. From the user's point of view, the CDL provides an alternative to specifying 0's and 1's by offering a much more natural and application-dependent vocabulary for describing model flow and the interconnectivity of model subsystems. Furthermore, users need not be concerned with disabling those unfamiliar areas of a large model that they are not interested in exercising. Since the CDL processor translates user desires into the source code supporting their specific intentions, programmers no longer need to create and maintain general purpose executives to accommodate all potential users. Thus, at runtime, users only pay for the address space necessary to contain the support modules directed into memory by their tailor made executives.

The Automated Code Handler for Output Operations (ACHOO) was developed under a similar "delay-until-runtime" premise. Specifically, it is a tool that can accept the names of user-supplied output variables and automatically create the source code of modules to support varying output requirements for a wide range of users exercising the same model. It is the intent of this paper to provide an overview of the technique; specific details for applying the concept are left to the appropriate documentation [Reynolds 1988].

2. MOTIVATION FOR DEVELOPING ACHOO

ACHOO could be described as an attempt to take the path of least resistance. Specifically, it offers automation as a substitute for any attempts at modifying the behavior or changing the attitudes of analysts and programmers in their usual handling of output requirements for large simulation models. Outlined in this section are some of the classical problems which inspired the development of ACHOO.

2.1 Total Output Requirements Not Always Known

At the onset of computer program design, the total known output requirements sometimes range from "none" to "partial". In some instances the reluctance to address these requirements early is predicated by the SMOP attitude. That is, it's a "simple matter of programming" to add output support code on a piecemeal basis as demand dictates. However, in other instances the analyst just does not know the totality of output requirements at program inception. For example, a favorite practice is to add unplanned code to print the results of intermediate calculations in trouble areas that crop up during testing and debugging. Many times this code remains in the model, under the control of input switches, for future activation if wholesale program changes are ever made that require extensive retesting.

Regardless of the motivation, missing output requirements make it difficult for the designer to create complete and intelligently partitioned databases. Also, a well structured model exhibiting strong modularity tends to deteriorate in these attributes when new output activityrequires collecting variables whose scopes are difficult to change from a local to global status.

2.2 Expanding Output Requirements

Later additions of code to implement new and expanding output requirements can best be described as a maintenance headache for all concerned. First, analysts may have to interface with programmers who, in turn, must insert and verify new code or juggle existing output lists and formats. Second, care must be taken to honor configuration control procedures because of the usual permanency of changes made. Last, additional care must be exercised not to disturb existing databases or files that other users do not want to see altered.

2.3 Too Much Overlap and Redundancy

It is extremely difficult to satisfy output requirements for a wide community of users of a particular model. Each user may have different reasons for executing the model, which is usually reflected as differing output requirements. For the most part, users can agree on a common kernel of information that remains fairly stable. It is when extensions occur beyond this nucleus, and the need to hide them from other users, that overlap and redundancy tend to creep in. It can range from multiple executions of the same case (with appropriate output controls enabled/disabled) to partially redundant output operations that transfer identical data to more than one file. The end result, not to mention the execution and memory overhead, is that many output files end up as subsets of other files.

2.4 No Standardization of Post Processing

Too frequently, analysts rely on the printed page and the process of pick and choose to isolate quantities of interest in order to detect anomalies in generated output or to prepare input to drive other simulations. It is just this frustrating process that leads to the special capture-the-specifics output changes which can contribute to the redundancy problem cited earlier.

A partial solution to the problems already outlined is to design models to create common databases that permit users to take advantage of offline processing such as interactive plotting. Quite frequently, plotting is accomplished by highly specialized and inflexible code that has been locked into the model. A better approach is to let analysts interact with a graphics terminal and selectively create plots from a common database. In this way, the pictorial behavior of a model can be examined from many different numerical and analytical viewpoints. In addition, analysts can be selective in those plots for which hard copies are desired. In a similar vein, interactive post processing also permits the selective reformatting of data for generating special reports or preparing input for use by other models.

2.5 Overhead of Output Code

Many of the large scale FORTRAN simulations developed at NSWC run on Control Data Corporation (CDC) computers under control of operating systems that do not support virtual memory concepts. Hence, memory usage has always been a big concern in the CDC environment. It is not uncommon for simulations to have 10% to 20% of their memory requirements devoted to code to support worst case output requirements. Yet, in many cases, only a small fraction of this code will be exercised at any one time. The penalty for the inactive code is degraded throughput and a more expensive run.

Of course, there is the cost incurred for the manual creation of the initial output support code during model development. These manpower costs continue when output adjustments are made during the life cycle. Not insignificant are the ecological costs in the unnecessary waste of paper, containing too much information, when one continues to ignore the benefits of interactive post processing.

An example of cost can be derived from the TRICS model mentioned previously. Its output code, representing approximately 10% of the total code written, was estimated to have cost nearly $40,000 to produce! Also, based on an estimate of initial TRICS usage, the cost for tying up memory with inactive code containing rarely used output options was estimated, in 1976 dollars, at $8K to $10K per year.

3. OBJECTIVES OF THE 'ACHOO' APPROACH

ACHOO offers a chance to reduce, significantly, the intensity of the manual labor usually employed in creating and modifying output source code. It does so by adopting the philosophy that code generated automatically is expendable. In turn, this implies that response to changing output requirements and better utilization of memory can best be done by automatically recreating or suppressing affected output modules to fit user needs.

Discussion of some of the available features should illustrate how ACHOO attempts to meet the objectives of: minimizing the roles of programmers, more user flexibility, and laying the foundation for better post processing activity.

3.1 PROBE Concept Reduces Programmer Role

Although it is impossible to get programmers completely out of the loop, it is certainly possible to minimize their presence. For the most part, analysts can identify early in the development cycle where they may want to monitor, or "probe", program behavior. However, the content of what is to be printed is usually subject to change.

The idea of a PROBE is introduced as any area within a model for which the potential for formatted output exists. (When output is generated it is in the form of mnemonic-value pairs.) Each PROBE is assigned a subroutine name to be invoked by the model; the actual code is not created for these subroutines until runtime. Thus the role of the programmer is reduced to inserting calls to probe subroutines while ACHOO does the rest. For each area of the model to be monitored, the programmer inserts the following:

CALL probe.name$_i$ (output.file, message.tag)

The name of the probe, probe.name$_i$, represents the ith subroutine name chosen by the programmer for the ith probe. The first argument, output.file, is the integer unit designator of the file on which probe output will be written. ACHOO assumes that the model establishes this unit-file connection based on conventions established within the model and the environment under which it is executed. The message.tag argument in this context allows access to the MESSAGES routine (see next section) to occur from within PROBE routines prior to printing the contents of probes. If a message is not desired for any probe, then message.tag is passed as one or more blanks.

At runtime, the content for those probes of interest can be declared. It should be noted that the programmer does not have to provide protection to suppress calls to non-existent or inactive probes. ACHOO automatically generates stubs (dummy routines) for those probes that are deactivated or for which no content is specified. It should be apparent that the PROBE concept does offer users a chance to control memory requirements for their particular output operations within a model. Also, programmer duties are simplified when and if new probes are requested later in the life cycle. Because of the nature of such additions (inserting "calls") the modularity of a well designed and structured program is preserved.

3.2 MESSAGES Feature for Output Annotation

In many large simulations, the appearance of short, informative messages throughout the output can go a long way in describing the occurrence or passage of events such as "THRUST INITIATION", "THRUST TERMINATION", and so forth. In many instances only messages are desired. In others, they may be accompanied by the printing of parametric values.

ACHOO offers the MESSAGES feature which permits the runtime definition or alteration of messages. The system will create the code for a generalized message routine that can be called directly by the simulation or indirectly from a PROBE. The latter implies that different messages could be displayed if a user wanted to invoke the same PROBE at different locations within a program. Wherever an explicit call is required, the programmer inserts the following:

```
CALL message.routine (output.file, message.tag)
```

The name of the routine, message.routine, is chosen by the programmer. The first argument is the same as discussed in the previous section. The last argument, message.tag, is a text string representing a label identifying a specific message. The labels and accompanying messages are supplied and maintained as a part of the ACHOO input environment.

3.3 BINARY.LINE Concept for Offline Processing

The BINARY.LINE feature allows for unformatted, "line-by-line" output from a programmer-designated area within a model. (For trajectory simulations, this form of output is usually referred to as "time line" output.) It can be viewed as a special probe having the attendant features and benefits associated with the PROBE discussed earlier. The key to this feature is that it allows users to get maximum information from a single execution while deferring to offline processing the plotting or editing of this information.

As with a PROBE, users delay their intentions regarding the content of the BINARY.LINE until runtime. If the logical BINARY.LINE envisioned by the user requires more than one physical record, the routine generated by ACHOO will perform the necessary multiple writes.

Only one BINARY.LINE routine is scripted for a model. Of course, it could be called from different areas within a model. However, from a purely functional point of view, it would more than likely be invoked from one, centralized location within the model hierarchy. Regardless, the programmer sets up the interface as follows:

```
CALL routine.name (binary.file)
```

Again, routine.name is chosen by the programmer and of course must be unique with respect to all routine names comprising the model, including probes. The unit designator, binary.file, identifies the file to receive the binary output where it is assumed that the model opens the file prior to first usage.

3.4 General Features

At runtime, users have many options available when specifying output requirements for a particular session [Reynolds 1988]. To mention a few, there are editing directives like ADD and REMOVE for altering the contents of PROBE, MESSAGES, or BINARY.LINE modules. Variables can be printed under multiple formats; e.g., one may want to see a floating point variable as just that and in hexadecimal as well. Entire arrays can be printed by simply stating their names. Array elements, including ranges of elements, can easily be specified. There is a SYNONYM feature that allows a different name to be printed for the actual program mnemonic. Global attributes may be specified that apply to all probes unless overridden within those probes that want contrary values. For example, it can be specified that the output of each PROBE will be printed starting at the current position within the output page. Thus, the exceptions that want to start at the top of a new page can override the current global setting from within.

ACHOO eliminates the need to go through the Boolean orgies normally associated with testing input options to determine whether a particular section of a program will display "brief," "medium," or "verbose" output. In these instances, each subsequent level is usually an addition or extension of a previous level. If a PROBE(s) exists for each level, then each logical level can be captured using the MACRO feature offered by ACHOO. Thus, one has a means to facilitate control of logical groupings of probes which can be referenced by macro name instead of trying to recall and reference individual constituents.

There are three directives (ACTIVATE, DEACTIVATE, and NOGEN) that permit easy manipulation of probes and macros as complete and unmodified entities. ACTIVATE requests that code be generated for a PROBE or MACRO unconditionally. DEACTIVATE requests that stub code be generated for a named PROBE or all probes embedded in a MACRO. This allows an instrumented simulation to continue to call probes which are simply no-ops after deactivation. The NOGEN directive forces ACHOO to suppress code generation for a PROBE or MACRO. For example, it would be desirable and efficient to inhibit code generation for invariant and frequently used probes that have been transferred to the model library.

4. SYSTEM CONFIGURATION

Figure 1 illustrates where ACHOO fits in the overall execution of a simulation model. The input files shown influence the direction taken by ACHOO in satisfying any particular set of output requirements. (Minimally, one of these files must always be present.) The DEFAULT PROBE ENVIRONMENT file is optional but is available as a master source for predetermined and tested probes. The RUNTIME PROBE REQUIREMENTS file must be an exact replica of a default file if no default is supplied. However, if a default file exists then the runtime file allows additional directives that delete, expand, or modify the default environment. In addition, the runtime file could contain the specifications for new probes appearing for the first time. Individual records appearing in both input files are free-form, i.e., with the exception of some keywords, there is no dependency on rigid formats or column boundaries.

Two output files are always generated by ACHOO: the PROBE REPORT file and the PROBE SOURCE CODE file. The former lists probe modifications, activated/deactivated probes, errors detected, and so forth. The latter contains the routines, one for each probe, generated by ACHOO. These can be compiled and, during the loading process, linked with the "calls" previously inserted by the programmer throughout the computational routines residing on the model library. Thus, the final load module contains only the output support necessary for this particular execution of the model. As mentioned earlier, the runtime deactivation of probes does not require that the corresponding code that invokes them be removed from the model. That is, ACHOO will script stubs for these probes to create the effect of a no-op if they are entered.

A third output file created by ACHOO represents a template that supports the BINARY.LINE feature. This formatted file, which contains information necessary for intelligent and responsive post processing activity, includes the date of the run, number of physical records per logical record, names and modes of each variable, and so on. As illustrated in Figure 1, the executing model creates a fourth output file of binary (unformatted) records by repeatedly invoking the ACHOO-created BINARY.LINE routine which records the line--by-line output.

5. SUMMARY

For years, simulation developers at NSWC have had to contend with output requirements that seem to vary directly with the number of users desiring access to a particular model. ACHOO offers an automated alternative to the labor intensive approaches employed in the past. It can be classified as a generic, off-the-shelf software engineering tool in that it does not embody the attributes of any particular simulation model. Nor, to the degree possible, does it reflect the native environment in which it was developed.

Users of large simulations have the complete freedom to modify and extend the contents of any PROBE through simple-to-use editing directives. They also have the ability to activate/deactivate, en masse, logical groupings of probes through the MACRO feature. Thus, individual users can now force a model to adapt to their specific output needs instead of contending with one that disappoints the majority.

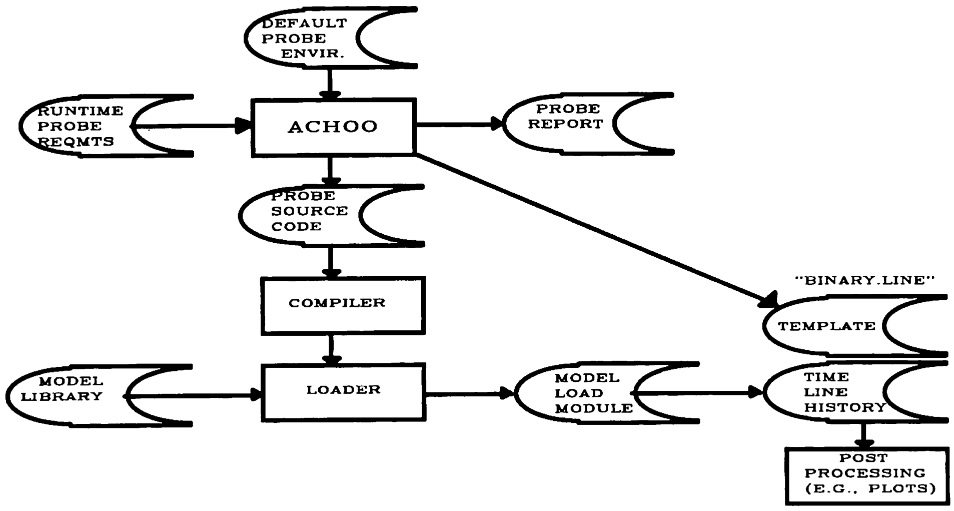

Figure 1. ACHOO-Model Interplay

4.1 Who Should Use ACHOO?

There is no doubt that ACHOO would be overkill for a tiny model containing one invariant probe that is always invoked. However, what about a fairly large model with numerous probes that are always invoked, will "never change", and "no additional probes will ever be needed?" Experience has taught most of us that the latter two conditions usually turn out to be false, implying ACHOO would be beneficial. Regardless, ACHOO could be used in a stand-alone mode to create the source code for the original probes. The compiled probe routines could be placed on the model library and ACHOO forgotten. If, later on, the output requirements do become unstable, then ACHOO could be reactivated, along with the original input files, and dropped into the execution stream as depicted in Figure 1.

The point being made is that users can flip flop on their commitment to the ACHOO philosophy. Only the designer and/or the user of the model who knows the output requirements, and their potential for instability, can decide to what degree ACHOO will be used.

REFERENCES

Goyette, P.J., and Owens, D.J. (1982), "INPUTP (General Purpose Input Processor) User Guide", Technical Report NSWC TR 3880, NSWC, Dahlgren, VA.

Lemoine, D.J. (1978), "Configuration Description Language Processor Design Disclosure", Technical Report NSWC TR 3881, NSWC, Dahlgren, VA.

Reynolds, J.H. (1980), "Evaluation of Contemporary Software Engineering Techniques for a Large FORTRAN Simulation", *The Journal of Systems and Software* 2, 6, 131-140.

Reynolds, J.H. (1988), "User Guide for ACHOO (Automated Code Handler for Output Operations)", Technical Report NSWC TR 88-245, NSWC, Dahlgren, VA.