EPA/600/R-99/030

Chapter 18

INTEGRATION OF SCIENCE CODES INTO MODELS-3

Jeffrey Young[*]
Atmospheric Modeling Division
National Exposure Research Laboratory
U.S. Environmental Protection Agency
Research Triangle Park, NC 27711

ABSTRACT

A complete group of CMAQ science codes has been integrated into the Models-3 system following a set of rules to ensure compatibility and compliance with design principles that enable modularity and flexibility and that allow easy modification and replacement of science process components. This chapter describes the concept of classes and modules, the Models-3 input/output application programming interface, science code configuration management, and model building and execution concepts

[*]On assignment from the National Oceanic and Atmospheric Administration, U.S. Department of Commerce. Corresponding author address: Jeffrey Young, MD-80, Research Triangle Park, NC 27711. E-mail: yoj@hpcc.epa.gov

18.0 INTEGRATION OF SCIENCE CODES INTO MODELS-3

18.1 Introduction

Integration of the Community Multiscale Air Quality (CMAQ) science into the Models-3 design paradigm takes place at the level of transforming the science, described by systems of partial differential equations, into a discretized and parameterized numerical model. The integral parts that are involved in the process are the coding language (Fortran 77, currently), an operational design, a set of coding rules, computer platforms, model data access methods and storage management, and a public code repository. Models-3 is a user interface system designed to facilitate the development and use of simulation models by scientists, model developers and regulatory users. The Models-3 framework provides a high-performance computational structure for a community modeling system that currently contains distinct emissions, meteorological, and chemical transport air quality models. In this chapter we define a model as a single, complete executable built from a set of compiled subroutines that define either a core environmental simulation model or a processing system that provides data to a simulation model. The framework also links the models to data management tools and analysis/visualization software.

A complete set of CMAQ science codes has been integrated into the Models-3 system. The development and integration of these codes has followed a set of principles relating to design and implementation concepts that are discussed in this chapter.

We have promulgated a small set of software requirements and integration rules related to these concepts so that science codes can be developed that conform to Models-3 requirements. With this approach, the general user community may readily develop and execute models that use the rich and growing base of science codes and data that reside in Models-3. A user may contribute to that base and allow others to integrate his developments by applying these integration rules.

With access to all the released, or publicly available source files, a developer can supplement his own codes and readily develop and test model versions within a highly modularized building environment. On the other hand, people whose primary interest is studying the effects of control strategies or regulatory applications can easily build and execute standard model versions for different emissions scenarios and modeling domains.

The Models-3 and CMAQ systems have been developed to foster community access, and to facilitate the improvement of the CMAQ system over time by the easy inclusion of new scientific models and modules developed by researchers in the larger scientific community.

The CMAQ system contains a number of models needed to carry out an air quality model application. Defining a “model” to be an executable built for a specific application, the codes that make up the model are relatively general. To build a specific model, the user selects a set of fundamental criteria that determine the application, such as the modeling domain, the chemical mechanism, and physical process solvers. The reader is referred to Chapter 15 for details. Defining a generic model as the set of codes from which a particular application can be built, the following generic CMAQ models are currently available in the system:

- **ICON** provides the required initial conditions for a model simulation as concentrations of specified individual chemical species for the complete modeling domain.

- **BCON** provides the needed simulation boundary conditions as concentrations of individual chemical species for the grid cells surrounding the modeling domain.

- **Emissions-Chemistry Interface Processor (ECIP)** transforms emission files produced by the Models-3 Emissions Processing and Projection System (MEPPS) into an hourly, 3-D emissions data file for the CMAQ Chemical Transport Model (CCTM). For a detailed discussion of the MEPPS operations, refer to Chapter 6 of the Models-3 User Manual [6].

- **The Meteorology-Chemistry Interface Processor (MCIP)** interprets the output from a meteorological model, such as the Penn State/NCAR fifth-generation Mesoscale Model (MM5) [8], and prepares the data for use in the CCTM.

- **The Landuse Processor (LUPROC)** provides a high-resolution landuse database for the CMAQ system. For example, MCIP uses the output from LUPROC to obtain surface characteristics for computing dry deposition and other planetary boundary layer parameters.

- **The photolysis model (JPROC)** computes photolytic rate constants (j-values) for the gas-phase chemistry used in the CCTM.

- **The Plume Dynamics Model (PDM)** generates plume dimensions and positions along with other related data for use in applying the plume-in-grid module in a CCTM simulation.

- **The CMAQ Chemical Transport Model (CCTM)** simulates the chemical and physical processes affecting tropospheric pollutants and estimates pollutant concentrations (e.g., ozone, particulate matter ($PM_{2.5}$ and PM_{10}), and carbon monoxide) and acid deposition.

The reader is referred to the Models-3 User Manual [6] for more details and descriptions of the usage of these models.

Each generic model consists of a complete set of codes that can be compiled and linked into different model executables by means of specific user-selected options with regard to groups of code called modules, the horizontal grid and vertical layer domain, the chemical mechanism, the computer platform, and compiling options.

One of the key issues in modeling is the manner in which developers and users must deal with reading, writing and using model data. Models-3 provides a user-friendly Input/Output Applications Programming Interface (I/O API) library [4] that enables a universal approach to managing data across subroutines, models, platforms, and networks.

The organization of the remainder of this chapter is as follows: We discuss the concepts that lead to and the implementation of classes and modules in the next section (18.2). We describe I/O API usage and function in Section 18.3. Code management, an important consideration for complex models, is discussed in Section 18.4. In Section 18.5 we discuss how a model is organized and constructed through Models-3. Issues in regards to executing a particular model are discussed in Section 18.6. Using the Models-3 framework to construct and execute a model is summarized in Section 18.7. Finally in Section 18.8, we discuss concepts and issues that address Models-3 compliant coding practice and how to ensure code development that is conformant to the Models-3 system.

18.2 Classes and Modules

18.2.1 Operational Design

The design concept of code classes is used to facilitate the plug&play capability in the Models-3/CMAQ system. Science process modules are grouped into classes that are primarily based on the time-splitting paradigm used in a CMAQ Chemical Transport Model (CCTM). Generally each class is associated with a particular science process. A module consists of a complete set of subroutines capable of modifying the concentration field related to the science process associated with the module's class. The CCTM is designed so that each module computes the changes in the concentration field (CGRID) specific to that particular science process. A science module operates on the entire three-dimensional gridded concentration field for a period of time called the synchronization time step interval. The module performs whatever looping over that grid and over whatever internal time steps that are necessary to complete its function.

Table 18-1 lists the classes and associated processes and modules currently available for the CCTM. In building a particular CCTM, one module is selected from each class.

The other CMAQ models (i.e., those other than the CCTM) do not generally follow the time-splitting science process paradigm. However, for convenience and consistency in model building, and for code organization, we have extended the concept of classes and modules to apply to the other CMAQ models. Class and module organizations similar to those in Table 18-1 exist for the other models but are less extensive.

Table 18-1. The Classes, Processes, and Modules Available in CCTM

CLASS	(PROCESS)	MODULES
driver	(Control Model Execution)	ctm
init	(Initialize Model)	init
hadv	(Horizontal Advection)	hbot (Bott’s scheme)
		hppm (Piecewise Parabolic Method)
		hadv_noop (no operation)
vadv	(Vertical Advection)	vbot (Bott’s scheme)
		vppm (Piecewise Parabolic Method)
		vadv_noop (no operation)
hdiff	(Horizontal Diffusion)	unif (Uniform eddy diffusion)
		hdiff_noop (no operation)
vdiff	(Vertical Diffusion)	eddy (eddy diffusion)
		vdiff_noop (no operation)
adjcon	(Advected Mass Adjustment)	denrate
		adjcon_noop (no operation)
phot	(Photolytic Rate Constants)	phot
		phot_noop (no operation)
chem	(Gas Chemistry Solver)	smvgear (SMVGEAR solver)
		qssa (Vectorized QSSA solver)
		chem_noop (no operation)
aero	(Aerosol Solver)	aero
		aero_noop (no operation)
aero_depv	(Aerosol Dry Deposition)	aero_depv
		aero_depv_noop (no operation)
cloud	(Cloud and Aqueous Chemistry)	cloud_radm (RADM cloud scheme)
		cloud_noop (no operation)
ping	(Plume-in-Grid)	ping_smvgear (uses SMVGEAR solver)
		ping_qssa (uses QSSA solver)
		ping_noop (no operation)
procan	(Process Analysis)	pa
couple	(Couple for Transport)	gencoor
util	(Utility Processing)	util

18.2.2 CGRID

To facilitate the science processing for the different chemistries involved in CMAQ, the concentration field array CGRID, is partitioned into four species classes: gas chemistry, aerosols, non-reactive, and tracer species. This order is mandatory and maintained throughout the processing. A subroutine, CGRID_MAP provides the means to correctly index into the CGRID array for each science process that deals with individual classes of species. The tracer species group is user-determined and may be empty in CGRID. Aqueous chemistry is dealt with solely in the cloud processing, and currently aqueous species are not transported outside of the cloud processing, therefore do not appear explicitly in CGRID. A subroutine call to CGRID_MAP provides pointers to the locations of the species classes within CGRID. A process that deals only with one of the classes (e.g. aerosols) can thus determine which part of CGRID it needs to access.

18.2.3 Class Driver

Generally, each module within a class requires a “class-driver,” which is the top level subroutine within the module - it is at the top of the call chain for that module. Exceptions for this paradigm exist for the util, procan, phot, and aero_depv classes. The class-driver presents a fixed name and calling interface to the driver class subroutine SCIPROC that calls all the science process modules. The calling interface consists of a subroutine argument list containing CGRID, the current scenario data and time, and a time step vector. Each class also contains a “no-operation” module, if appropriate. The no-op module has the fixed name and call interface for that class, but performs no calculations. When called, it merely returns control to SCIPROC.

The following algorithm illustrates the processing and call sequence used in a CMAQ model for fractional time steps processed non-symmetrically with respect to the chemistry processes:

```
Load CGRID from initial data; set initial Date/Time
Get the TimeStep vector:   TimeStep[1] = output time interval
                           TimeStep[2] = synchronization time interval
                           TimeStep[3] = horizontal advection time interval
Foreach output_time_step

      Call Couple ( CGRID, Date, Time, TimeStep )*

      Foreach sync_step  [in output time interval]
            Call Horizontal_Advection ( CGRID, Date, Time, TimeStep )
            Call Vertical_Advection   ( CGRID, Date, Time, TimeStep )
            Call Mass_Adjustment      ( CGRID, Date, Time, TimeStep )
            Call Horizontal_diffusion ( CGRID, Date, Time, TimeStep )
            Call De-Couple            ( CGRID, Date, Time, TimeStep )*
            Call Vertical_diffusion   ( CGRID, Date, Time, TimeStep )
            Call Plume_in_Grid        ( CGRID, Date, Time, TimeStep )
            Call Gas_Chemistry        ( CGRID, Date, Time, TimeStep )
            Call Aerosol              ( CGRID, Date, Time, TimeStep )
            Call Cloud                ( CGRID, Date, Time, TimeStep )
            Advance Date/Time by synchronization time interval
            Call Couple               ( CGRID, Date, Time, TimeStep )*
      End Foreach sync_step

      Call De-Couple ( CGRID, Date, Time, TimeStep )*
      Write Concentration File

End Foreach output_time_step
```

* Couple CGRID concentrations with, or de-couple CGRID from the air density × the Jacobian of the computational grid. See Chapter 6 for a complete discussion on the concept.

Not shown in the algorithm are processes that deal with source emissions (in either Vertical_diffusion or Gas_chemistry), dry deposition (in Vertical_diffusion), and wet deposition (in Cloud). Also not shown are calls to process analysis routines after each science process, which compute the integrated process rates for that particular process. Such calls are also embedded in science process modules that affect CGRID, such as emissions injection in either the vertical diffusion process or gas chemistry. The Science Process Code Template in Section 18.8.3 below illustrates how these calls can be integrated within a module.

In order to maintain the Models-3/CMAQ modularity and data-independence, the science process class-drivers must adhere to coding standards with regard to the calling interface, file data I/O, and standardized, uniform domain and mechanism data within global include files. See Section 18.5, below, and Chapter 15 for a more complete discussion on the use of global include files. These coding standards have not been propagated to the level below the class-driver. At the sub-module level there are no strict I/O or coding standards, however we offer suggestions to facilitate the potential incorporation of a module into the Models-3 system in Section 18.8 below.

18.2.4 Synchronization Time Step

There are many different time scales that are important in modeling. The CCTM deals with four levels of time stepping:

1. **Output time step** - the time interval for which output data is written to disk.

2. **Synchronization time step** - the time interval for which the science processes, represented by science modules, are considered to run independently of the other processes. This is the time interval during which no interaction with the other time-split science processes needs to take place.

3. **Advection time step** - the time interval over which horizontal advection occurs. It may be less than or the same as the synchronization time interval.

4. **Local or internal time looping** - a possibly variable time interval that subdivides the synchronization time step and is dependent on the user's implementation of a particular algorithm.

There may also be some science processes that don't fit into this type of time scale hierarchy. The sub-grid cloud time step in CCTM for example, has a fixed lifetime that may span the synchronization time interval.

In the CCTM the synchronization time step is essentially the time interval over which the chemistry processes are considered to be time-split and independent of the other processes.

The proper relationship between the first three time steps listed above is determined by the "ADVSTEP" algorithm. Based on a user-supplied output time step interval, and optionally an upper and lower limit for the synchronization time step interval,[1] the algorithm:

- Ensures the synchronization time interval evenly divides the output time interval.

- Ensures the synchronization interval to not be greater than the upper limit.

- Determines a horizontal advection time interval that ensures all horizontal advection calculations satisfy a Courant condition with respect to the horizontal winds for each output time step. See Chapter 7 for details about the Courant condition requirement.

[1]If these limits are not specified, the algorithm uses defaults of 900 and 300 seconds, respectively.

- Attempts to establish a horizontal advection time interval as close to the synchronization interval as possible, but evenly divides the synchronization interval.
- In case the Courant condition restriction forces an advection time interval to be less than the lower synchronization interval limit, sets advection time intervals to be as close to the lower limit as possible but still evenly divide[2] the lower limit.
- If none of these criteria can be satisfied, reports the issue and aborts the execution.

18.3 Input/Output Applications Programming Interface

A model needs access to external data. Data access, including output to external files, is done at the module level. The Models-3 I/O API [4] provides a standardized interface to external data that enables a user to follow the object oriented design concept of *encapsulation*. For example, all basic and some derived meteorological data are calculated outside the CCTM (see MCIP documentation in Chapter 12, e.g.) and available to a module through calls to the Models-3 I/O API library [4]. Thus the user can avoid the re-calculation of various meteorological variables within subroutines using different algorithms and parameterizations to re-compute essentially the same quantities, which may give rise to errors and modeling inconsistencies. It is expected that the modules treat these I/O API data completely independently, not knowing or caring if files have been opened or read by any other module. In particular, any subroutine within a module can open an I/O API file. It is recommended that at least one routine within a module, preferably the class-driver, should open any I/O API files that are required in the module.

In general, the I/O API provides the Models-3 user with a library containing both Fortran and C routines, which manage all the necessary file manipulations for data storage and access. The main requirement is that the user follows the I/O API conventions for data structures, naming conventions, and representations of scenario date and time. The I/O API routines manage access to the files in such a manner that data can be read or written flexibly, virtually in any order, and freeing the user from being concerned with low level file manipulation details, such as the order or format of file variables.

In addition, the I/O API routines can be used for both file storage and cross-media model coupling using Parallel Virtual Machine (PVM) mailboxes.

For file storage and access, the I/O API, which is the standard data access library for Models-3, is built on top of NCAR's netCDF files [5]. NetCDF (network Common Data Form) is a library that provides an interface for array-oriented data access. The netCDF library also defines a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific data. The netCDF software was developed at the Unidata Program Center in Boulder, Colorado. The freely available source can be obtained by anonymous FTP [5].

The I/O API provides a variety of data structure types for organizing the data, and a set of access routines that offer selective direct access to the data in terms meaningful to the user.

[2]Strictly speaking, this means divides without remainder, not by a factor of two.

Since they are netCDF files, I/O API files share the following characteristics:

1. They are machine-independent and network-transparent. Files created on a Cray Supercomputer can be read on a desktop workstation (or vice versa) either via NFS mounting or FTP, with no data format translation necessary;

2. They are self-describing. That is, they contain headers that provide a complete set of information necessary to use and interpret the data they contain;

3. They are direct access. A small subset of a large dataset may be accessed efficiently without first reading through all the preceding data. This feature is mandatory in visualization requiring rapid access of large datasets;

4. They are appendable. Data can be appended to a netCDF dataset along one dimension without copying the dataset or redefining its structure. For example, the I/O API can add more time step data to a previously created file; and

5. They are sharable. One writer and multiple readers may simultaneously access the same netCDF file. This means, for example, that visualization tools can access the file and display data while the file is being created by the model.

The I/O API has been designed to support a variety of data types used in the environmental sciences, among them:

- Gridded data (e.g., concentration or meteorological fields);
- Grid-boundary data (for model boundary conditions);
- Scattered data (e.g., meteorology observations or source level emissions); and
- Sparse matrix data (a specialized data type used in an emissions model).

I/O API files support three different time step structures:

1. Time-stepped with regular time steps (e.g. hourly model output concentration fields or twice-daily upper air meteorology observation profiles);

2. Time-independent (e.g. terrain height); and

3. Restart, which always maintains the last two time step records of output from a running process as “even” and “odd.” The “odd” is available for restart in case of a crash while the “even” step is being written, and vice versa. Restart data does not consume an inordinate amount of disk space with only two time steps worth of data at all times.

The I/O API provides automated built-in mechanisms to support production and application requirements for dataset histories and audit trails:

- Identifiers of the program execution that produced the file.

- Description of the study scenario in which the file was generated.

The I/O API also contains an extensive set of utility routines for manipulating dates and times, performing coordinate conversions, storing and recalling grid definitions, sparse matrix arithmetic, etc. There are a variety of related programs that perform various analysis or data-manipulation tasks, including statistical analysis, file comparison, and data extraction. Section 18.8.3 below illustrates the use of some of these utilities in coding practice, and the web site [5] provides background materials, a user manual, and a tutorial.

18.4 Code Configuration Management

18.4.1 The Need

Faced with a large and growing community that uses and develops a wide variety of programs, modules and codes, it is imperative to systematically manage the cross-community access to this software. Typically, successful management of software involves:

- A repository - a place where all of the public code resides;

- The concept of archived code - codes that have been deposited into the repository in such a manner that anyone can extract the exact code at a later time. This involves some kind of transformation program to maintain master copies of the codes with embedded change tables;

- The concept of revision control - archiving codes results in modifying the tags or unique revision identifiers in the change tables in the master copies in order to recover the exact code at a later date; and

- The concept of released code - codes that have reached some state of maturity and have been designated with some kind of “released” status. They can be used with reasonable expectation of reliability.

The paradigm used employs the following scenario.

1. A user modifies or develops code. The code may be one subroutine or many, possibly comprising whole science models. The code may originate from “scratch,” or be extracted from the repository and modified.

2. After testing or reaching a point of being satisfied with his/her results, he/she decides to save it in the repository so that others can have access to it.

3. Some archived codes may still be in an experimental, or development, state while others may be reasonably stable and more completely tested. The latter may be designated as “released.” There is no enforceable means to control access based on an experimental or released state. The community will have and should have access indiscriminately, well aware that using development state code is risky.

4. As the user continues to work with the codes, he/she may make enhancements or discover and fix errors. The upgrades are then installed in the repository, which automatically assigns unique revision identifiers.

5. The repository is located where it is conveniently accessible to all pertinent users, and it is maintained by an administrator who sets and enforces general access rules.

18.4.2 The Tool

There are many configuration management tools both free and commercially available. We chose The Concurrent Versions System (CVS) [1] mainly because of its versatility. CVS controls the concurrent editing of sources by several users working on releases built from a hierarchical set of directories. CVS uses the Revision Control System (RCS) [2] as the base system. Other reasons that CVS was an attractive choice include:

- It works on virtually all UNIX platforms and many PCs;
- It is publicly available and free;
- CVS is a state-of-the-art system, constantly being improved; and
- MM5 codes are managed by CVS, and MM5 is a primary meteorology model for the CMAQ system.

From the UNIX man pages (online manual):

> CVS is a front end to the Revision Control System (RCS) that extends the notion of revision control from a collection of files in a single directory to a hierarchical collection of directories consisting of revision controlled files. These directories and files can be combined together to form a software release. CVS provides the functions necessary to manage these software releases and to control the *concurrent* editing of source files among multiple software developers [emphasis added].
>
> The Revision Control System (RCS) manages multiple revisions of files. RCS automates the storing, retrieval, logging, identification, and merging of revisions.

Another widely used source code management system is the Source Code Control System (SCCS), which is usually distributed with UNIX systems. The main note is that RCS and SCCS act on files only, while CVS operates on projects. Working with entire projects works better and easier with CVS. Multi-developer project development works better with CVS as well. CVS supports a client/server mode of operation that can be very useful. CVS also allows customization by adding hooks so that local scripts or programs can be called when executing various CVS commands. This can be useful to force naming conventions of tags, or reference to bug-tracking software, etc. Thus CVS adds power and features that are attractive for the ModelsS-3/CMAQ system.

18.4.3 The Repository

The repository structure, that is, the UNIX directory hierarchy, follows the class/module organization discussed above in Section 18.2. The repository is actually divided into many repositories, one for each generic model. This division makes it easier to maintain the class/module organization that is important for the model building operation described below in Section 18.5.

CVS allows for the use of a “modules” file[3], which enables a user to easily check out or extract a complete CMAQ module. For example, a module might be checked out by a user to make code modifications, and complete modules are checked out during the model building operation.

The following shows a symbolic CVS UNIX directory tree that represents the current structure for the CCTM:

[3]The terminology is unfortunate. The CVS *modules* file has no intrinsic relationship with the CMAQ classes/module design implementation.

```
...CCTM -+-> CVSROOT ---+-> CVS administrative files
         |              |
         \-> src -------+-> driver --+-------> ctm ------+-> RCS files
                        |
                        +-> hadv ---+---> hadv_noop ---+-> RCS files
                        |           +---> hbot --------+-> RCS files
                        |           +---> hppm --------+-> RCS files
                        |
                        +-> vadv ---+---> vadv_noop ---+-> RCS files
                        |           +---> vbot --------+-> RCS files
                        |           +---> vppm --------+-> RCS files
                        |
                        +-> vdiff --+---> vdiff_noop --+-> RCS files
                        |           +---> eddy --------+-> RCS files
                        |
                        +-> chem ---+---> chem_noop ---+-> RCS files
                        |           +---> smvgear -----+-> RCS files
                        |           +---> qssa --------+-> RCS files
                        |
                        +-> phot ---+---> phot_noop ---+-> RCS files
                        |           +---> phot --------+-> RCS files
                        |
                        +-> aero ---+---> aero_noop ---+-> RCS files
                        |           +---> aero --------+-> RCS files
                        |
                        +-> couple -+---> gencoor -----+-> RCS files
                        |
                        +-> cloud --+---> cloud_noop --+-> RCS files
                        |           +---> cloud_radm --+-> RCS files
                        |
                        +-> procan -+---> pa ----------+-> RCS files
                        |
                        +-> hdiff --+---> hdiff_noop --+-> RCS files
                        |           +---> unif --------+-> RCS files
                        |
                        +-> init ---+---> init --------+-> RCS files
                        |
                        +-> util ---+---> util --------+-> RCS files
                        |
                        +-> aero_depv +-> aero_depv_noop +-> RCS files
                        |             +-> aero_depv -+-> RCS files
                        |
                        +-> adjcon -+---> adjcon_noop -+-> RCS files
                        |           \---> denrate -----+-> RCS files
                        |
                        \-> ping ---+---> ping_noop ---+-> RCS files
                                    +---> ping_qssa ---+-> RCS files
                                    \---> ping_smvgear +-> RCS files
```

The symbolic tree is shown relative to the subdirectory in the repository named for the CCTM model. Similar trees exist for each of the generic models. The RCS files are the revision control history files that contain the change tables to reconstruct the actual source code according to a specific revision identifier. Also note that the tree closely follows the organization of classes and modules for the CCTM described in Table 18-1, and contains alternate modules within the classes. In particular, most classes contain a “no-operation” (_noop) module that allows a user to essentially turn off that particular science process modeling. This is useful, for example in debugging, where rapid turn-around is important, and a computationally demanding module that is not needed can be bypassed.

18.5 How a Model is Constructed

18.5.1 Object Oriented Concepts

To make the Models-3/CMAQ system robust and flexible, object oriented concepts were incorporated into the design of the CMAQ system. Incorporating these ideas into the design helps avoid introducing errors when code modifications are necessary. Additionally, the system is capable of easy and efficient modification, allowing the user to quickly make models for different applications.

The implementation language for CMAQ is Fortran 77, which imposes limits on how far one can go in terms of object oriented design. In particular, since Fortran is a static language, objects cannot be instantiated dynamically; they must be declared explicitly in the source code to be created at compile time. However, to encourage a user community that will be contributing code for future enhancements, every attempt has been made to adhere to the Fortran 77 standard. In the future, the use of other implementation languages such as Fortran 90 will be considered.

18.5.2 Global Name Table Data

In order to implement modularity and data-independence, we have employed design ideas that draw heavily from the object-oriented concept of *inheritance* and code re-use. The data structures in the codes that deal with the domain sizes, chemical mechanism, I/O API, logical file names, general constants, and CGRID pointers are determined by Fortran declarations in data and parameter statements that are created through the Models-3 system. These data structures pertain to a particular application (domain, mechanism, etc.) and are meant to apply globally, not only to one particular CCTM through all its subroutines, but also to all the models that supply data to the CCTM for that application. These data structures are contained in Fortran INCLUDE files, which are essentially header files, included in the declaration sections near the top of the Fortran code source files. The inclusion of these source files is made automatic by using a generic string that represents the include file and which is parsed and expanded to the actual include file during a pre-processing stage in the compilation. The Fortran global include files contain name tables that define:

1. The computational grid domain;

2. The chemical mechanism;

3. The I/O API interface, including logical file names;

4. The global modeling constants; and

5. Other constants or parameters that apply across the model.

In order to effect the implementation of the include files into the code, a special compiling system, m3bld, has been developed [3], which reads a configuration file that, based on the application, completely determines the model executable to be built. The ASCII configuration file can be generated either by the Models-3 system or by the user following a few, simple syntactical rules illustrated below. For additional details, the reader is referred to Chapter 15.

In addition to the global include files, the configuration file contains module commands that tell m3bld to extract the codes for that module from the model code repository for compilation.

18.5.3 Build Template

The following exhibit is an example of a configuration file that m3bld would use to build a model executable: (The numerals at the left-hand margin are labels for the legend and are not part of the configuration file. The "//" represents a non-parsed, comment line.)

Example Configuration File

```
(1)  model TUTO_r4y;

(2)  cpp_flags    "-Demis_vdif  ";
(2)  f77_flags    "-e -fast -O4 -xtarget=ultra2 -xcache=16/32/1:1024/64/1";
(2)  link_flags   "-e -fast -O4 -xtarget=ultra2";

(3)  libraries    "-L/home/models3/tools/IOAPI/release/m3io/lib/SunOS5 -lm3io
                  -L/home/models3/tools/netCDF/SunOS5 -lnetcdf";

(4)  global verbose;

(5)  include   SUBST_BLKPRM      /work/rep/include/release/BLKPRM_500.EXT;
(5)  include   SUBST_CONST       /work/rep/include/release/CONST3_RADM.EXT;
(5)  include   SUBST_FILES_ID    /work/rep/include/release/FILES_CTM.EXT;
(5)  include   SUBST_EMPR_VD     /work/rep/include/release/EMISPRM.vdif.EXT;
(5)  include   SUBST_EMPR_CH     /work/rep/include/release/EMISPRM.chem.EXT;
(5)  include   SUBST_IOPARMS     /work/rep/include/release/PARMS3.EXT;
(5)  include   SUBST_IOFDESC     /work/rep/include/release/FDESC3.EXT;
(5)  include   SUBST_IODECL      /work/rep/include/release/IODECL3.EXT;
(5)  include   SUBST_XSTAT       /work/rep/include/release/XSTAT3.EXT;
(6)  include   SUBST_COORD_ID    /work/yoj/tgt/BLD/COORD.EXT;
(6)  include   SUBST_HGRD_ID     /work/yoj/tgt/BLD/HGRD.EXT;
(6)  include   SUBST_VGRD_ID     /work/yoj/tgt/BLD/VGRD.EXT;
(6)  include   SUBST_RXCMMN      /work/yoj/tgt/BLD/RXCM.EXT;
(6)  include   SUBST_RXDATA      /work/yoj/tgt/BLD/RXDT.EXT;
(6)  include   SUBST_GC_SPC      /work/yoj/tgt/BLD/GC_SPC.EXT;
(6)  include   SUBST_GC_EMIS     /work/yoj/tgt/BLD/GC_EMIS.EXT;
(6)  include   SUBST_GC_ICBC     /work/yoj/tgt/BLD/GC_ICBC.EXT;
(6)  include   SUBST_GC_DIFF     /work/yoj/tgt/BLD/GC_DIFF.EXT;
(6)  include   SUBST_GC_DDEP     /work/yoj/tgt/BLD/GC_DDEP.EXT;
(6)  include   SUBST_GC_DEPV     /work/yoj/tgt/BLD/GC_DEPV.EXT;
(6)  include   SUBST_GC_ADV      /work/yoj/tgt/BLD/GC_ADV.EXT;
(6)  include   SUBST_GC_CONC     /work/yoj/tgt/BLD/GC_CONC.EXT;
(6)  include   SUBST_GC_G2AE     /work/yoj/tgt/BLD/GC_G2AE.EXT;
(6)  include   SUBST_GC_G2AQ     /work/yoj/tgt/BLD/GC_G2AQ.EXT;
(6)  include   SUBST_GC_SCAV     /work/yoj/tgt/BLD/GC_SCAV.EXT;
(6)  include   SUBST_GC_WDEP     /work/yoj/tgt/BLD/GC_WDEP.EXT;
(6)  include   SUBST_AE_SPC      /work/yoj/tgt/BLD/AE_SPC.EXT;
(6)  include   SUBST_AE_EMIS     /work/yoj/tgt/BLD/AE_EMIS.EXT;
(6)  include   SUBST_AE_ICBC     /work/yoj/tgt/BLD/AE_ICBC.EXT;
(6)  include   SUBST_AE_DIFF     /work/yoj/tgt/BLD/AE_DIFF.EXT;
(6)  include   SUBST_AE_DDEP     /work/yoj/tgt/BLD/AE_DDEP.EXT;
(6)  include   SUBST_AE_DEPV     /work/yoj/tgt/BLD/AE_DEPV.EXT;
(6)  include   SUBST_AE_ADV      /work/yoj/tgt/BLD/AE_ADV.EXT;
(6)  include   SUBST_AE_CONC     /work/yoj/tgt/BLD/AE_CONC.EXT;
```

Example Configuration File

```
(6)  include   SUBST_AE_A2AQ     /work/yoj/tgt/BLD/AE_A2AQ.EXT;
(6)  include   SUBST_AE_SCAV     /work/yoj/tgt/BLD/AE_SCAV.EXT;
(6)  include   SUBST_AE_WDEP     /work/yoj/tgt/BLD/AE_WDEP.EXT;
(6)  include   SUBST_NR_SPC      /work/yoj/tgt/BLD/NR_SPC.EXT;
(6)  include   SUBST_NR_EMIS     /work/yoj/tgt/BLD/NR_EMIS.EXT;
(6)  include   SUBST_NR_ICBC     /work/yoj/tgt/BLD/NR_ICBC.EXT;
(6)  include   SUBST_NR_DIFF     /work/yoj/tgt/BLD/NR_DIFF.EXT;
(6)  include   SUBST_NR_DDEP     /work/yoj/tgt/BLD/NR_DDEP.EXT;
(6)  include   SUBST_NR_DEPV     /work/yoj/tgt/BLD/NR_DEPV.EXT;
(6)  include   SUBST_NR_ADV      /work/yoj/tgt/BLD/NR_ADV.EXT;
(6)  include   SUBST_NR_N2AE     /work/yoj/tgt/BLD/NR_N2AE.EXT;
(6)  include   SUBST_NR_N2AQ     /work/yoj/tgt/BLD/NR_N2AQ.EXT;
(6)  include   SUBST_NR_SCAV     /work/yoj/tgt/BLD/NR_SCAV.EXT;
(6)  include   SUBST_NR_WDEP     /work/yoj/tgt/BLD/NR_WDEP.EXT;
(6)  include   SUBST_TR_SPC      /work/yoj/tgt/BLD/TR_SPC.EXT;
(6)  include   SUBST_TR_EMIS     /work/yoj/tgt/BLD/TR_EMIS.EXT;
(6)  include   SUBST_TR_ICBC     /work/yoj/tgt/BLD/TR_ICBC.EXT;
(6)  include   SUBST_TR_DIFF     /work/yoj/tgt/BLD/TR_DIFF.EXT;
(6)  include   SUBST_TR_DDEP     /work/yoj/tgt/BLD/TR_DDEP.EXT;
(6)  include   SUBST_TR_DEPV     /work/yoj/tgt/BLD/TR_DEPV.EXT;
(6)  include   SUBST_TR_ADV      /work/yoj/tgt/BLD/TR_ADV.EXT;
(6)  include   SUBST_TR_T2AQ     /work/yoj/tgt/BLD/TR_T2AQ.EXT;
(6)  include   SUBST_TR_SCAV     /work/yoj/tgt/BLD/TR_SCAV.EXT;
(6)  include   SUBST_TR_WDEP     /work/yoj/tgt/BLD/TR_WDEP.EXT;

     // Process Analysis / Integrated Reaction Rates processing
(6)  include   SUBST_PACTL_ID    /work/yoj/tgt/BLD/PA_CTL.EXT;
(6)  include   SUBST_PACMN_ID    /work/yoj/tgt/BLD/PA_CMN.EXT;
(6)  include   SUBST_PADAT_ID    /work/yoj/tgt/BLD/PA_DAT.EXT;

(7)  module ctm release ;

(7)  module init release ;

     // options are denrate and adjcon_noop
(7)  module denrate release ;

(7)  module gencoor release ;
     // options are hbot and hadv_noop
(7)   module hppm release ;

    // options are vbot and vadv_noop
(7)   module vppm release ;

    // options are phot and phot_noop

(7)   module phot release ;

    // options are ping_qssa, ping_smvgear and ping_noop
 (7)  module ping_qssa release ;

    // options are qssa, smvgear and chem_noop
(7)  module qssa release ;

    // aerosols
```

Example Configuration File

```
(7)  module aero release ;
     // aerosol dep vel
(7)  module aero_depv release ;

    // options are eddy and vdiff_noop
(7)  module eddy release ;

    // options are const and hdiff_noop
(7)  module unif release ;

    // options are cloud_radm and cloud_noop
(7)  module cloud_radm release ;

    // options are pa and pa_noop, which requires the replacment of the three
    // global include files with their pa_noop counterparts
(7)  module pa release ;

(7)  module util release ;
```

Legend:
(1) - Model name definition. This string is what the model executable will be named.
(2) - The C pre-processor, Fortran compiler and link flags. The flags specified indicate that the model is to be compiled with the option to input emissions in the vertical diffusion processing, and to compile and link on a Sun Sparc, Ultra-30 workstation.
(3) - I/O API and netCDF object libraries to be linked. The format is virtually identical to that of a UNIX make command.
(4) - m3bld flag: one of many options. verbose indicates report all actions.
Other options are:

- compile_all -   force compile, even if object files are current
- clean_up   -   remove all source files upon successful completion
- no_compile -   do everything except compile
- no_link   -   do everything except link
- one_step   -   compile and link in one step
- parse_only -   checks config file syntax
- show_only   -   show requested commands but doesn’t execute them

(5), (6) - global include files. The syntax is:
 include **internal-string** **full-path-name.**
Every routine that requires a specific include file must contain a Fortran include statement using the **internal-string** (see Section 18.3 for examples).
 (5) - “fixed” global include files. The fixed include files have been constructed outside of the Models-3 framework and contain global data not directly related to the domain or chemical mechanism such as modeling constants and file logical names.
(6) - The Models-3 framework automatically generates all the include files labeled (6). The user supplies information through the Models-3 Science Manager that determines the data in these include files. These data define the complete problem domain for a particular CMAQ application. See Chapter 15 for a complete discussion.
(7) - The module name to extract from the model code repository and an optional revision flag to select a particular module version. The syntax is:
 module **module** name **revision** flag.

Example Configuration File

For additional information on this implementation, the reader is referred to the Model Building Tool described in Fine et al. [3].

18.6 How a Model is Executed

In order to run a model executable, various UNIX environment variables must be set in the shell that invokes the execute command. Generally, these involve the modeling scenario start date and time, the run duration, the output time step interval, various internal code flags that differ among the models, and all the input and output logical, or symbolic file names (see Section 15.4.3 in Chapter 15). There are various ways by which external file names can be referenced in the code, but the most general across all UNIX platforms is to link them by means of environment variables. There are I/O API utility functions that allow a user easy access to these variables within the code, making such accesses generic and portable.

An additional feature that is provided through the I/O API is to declare a file "volatile" by appending a -v flag in the shell's declaration for the environment variable. By doing this, the I/O API will cause the netCDF file to update (sync) its disk copy after every write and thereby update the netCDF header. Otherwise, netCDF file headers are not updated until the files are closed. This is useful, for example, to allow a user to analyze an open netCDF file using visualization tools while the model is executing. It is also useful in case of a system crash. A CCTM model can be restarted at the scenario time step after the last successful write using the aborted output file as the input initial data.

18.7 Using the Models-3 Framework

The Models-3 framework simplifies the model building and model execution tasks, especially if key objects have been pre-defined. Thus if a user is conducting a study that uses the same domain or chemical mechanism, but uses different code options, e.g., then rebuilding and re-executing using the framework makes conducting the study very straightforward. Other examples where the framework greatly simplifies operational tasks would be where a model is to be applied on different domains, or for control strategies in which the only thing that changes are the emissions input files. A knowledgeable user is still able to build and execute model applications outside the framework. In fact, one can easily run model applications using executables and UNIX scripts generated by the framework.

In general, in order to build a model within the framework, the coordinate system, horizontal grid and vertical layer definitions, and chemical mechanism must be specified. This is done within the Models-3 Science Manager component. If objects required for a particular application, such as a coordinate system specification, are already defined, then the user needs only to select those objects. The Configuration File Manager, an additional subcomponent within the Science Manager, allows a user to control the module definitions and fixed [4] include files that go into the

[4]not generated by the framework

configuration file used to build the model. Also the user can specify compiling and linking flags related to a specific platform on which the model is to be executed.

With all required Models-3 objects defined, the task of building a model is made simple using the Models-3 component, Model Builder. It's important to note that as users develop models, the set of objects will grow to become a rich pool from which to select for a particular application. The Models-3 framework was designed to allow extensive re-use of objects. Model Builder generates all the "non-fixed" include files, then builds a model, linking in the include files including the user-specified fixed ones. See the section above for the include file definitions. Perhaps one of the most useful features of the Models-3 framework is the ability to graphically set up and execute single or multiple model executions with automatic data registration. This allows a user to carefully control the order of model executions and the management of all the input and output data. The Models-3 component that performs this function is the Study Planner. Study Planner allows a user to specify input data sets, executables, and UNIX environment variables, and to initiate (possibly multiple) model runs.

18.8 Conformant Code

18.8.1 Thin Interface

As mentioned above in section 18.5.1 , the Models-3/CMAQ system was designed to be robust and flexible with respect to the interchange of modules and the elimination of cross-module data dependencies. Consequently, the concept of a "thin interface" has been employed in the design, which applies principally to the class-drivers. At the least, the thin interface implementation implies the following requirements:

- Eliminate global memory references (across modules). This implies no common blocks across modules, no hidden data paths, no "back doors";
- Each module reads and interpolates its required data independently. The I/O API helps to ensure this kind of data independence; and
- Standardized argument list (CGRID, Date, Time, TimeStep) for the class-driver, as described in the section above.

These requirements attempt to incorporate the object-oriented idea of encapsulation in the Models-3/CMAQ design. The following quotation is from Rumbaugh et al. [7]:

> Encapsulation (also information hiding) consists of separating the external aspects of an object, which are accessible to other objects, from the internal implementation details of the object, which are hidden from other objects. Encapsulation prevents a program from becoming so interdependent that a small change has massive ripple effects. The *implementation* of an object can be changed without affecting the applications that use it [emphasis added].

The encapsulation design makes the CMAQ system safer and enables the transaction processing, plug&play capability. This design also makes it easier for a user to trace data and usage within a module, particularly at the class-driver level.

18.8.2 Coding Rules

In order to maintain the object oriented concepts implemented in the CMAQ system design, we have established a small set of coding rules that apply for those that develop CMAQ science and affect the low-level design of the models. We have developed standards to control data dependencies at the class-driver level, but we have not propagated these coding standards to the sub-module level.

1. The models are generally coded in Fortran (Fortran-77 conventions are used). It is possible to link in subroutines written in the C language, although this has not been done within the current CMAQ implementation.

2. The modules must be controlled by a top-level class-driver routine, whose calling arguments must be the computational concentration grid array, CGRID, the current scenario data and time, and the controlling time step vector. See the section above.

3. The class-driver is also responsible for any temporal integration required within the module. (The time steps for process integration at the module level are usually shorter than those of the CCTM synchronization time step.)

4. Any reads and writes for the module should be done at the level of the class-driver routine. Although not absolutely necessary, this is strongly suggested because it is usually much easier to control the timing of the data accesses at the highest level of the module where the current scenario date and time are known.

5. Use the IMPLICIT NONE Fortran declaration to maintain some control on typographic errors and undefined variables. Although not standard Fortran-77, the use of IMPLICIT NONE forces the developer to declare all internal variables.

6. Use the global include files for domain definitions, chemical mechanism data, and other data where available.

7. Use the I/O API for external data references where appropriate. For an illustration of these rules, the reader is referred to the code template below.

At the sub-module level there are no strict I/O or coding standards. Here it is envisioned that individual researchers/programmers use their own coding styles for their algorithms. However the following suggestions are offered to facilitate the potential incorporation of a module into the CMAQ system:

- It is expected that MKS units are used for input and output variables, as these units have been standardized throughout the CMAQ system. Within a sub-module subroutine whatever units are most convenient can be used. However, the developer must be responsible for any unit conversions to MKS for input and output, and thus avoid any potential errors.

- For efficiency and performance considerations, operations may need to be done on groups of grid cells (a block of cells) at a time. If there are *N* cells in the block and the entire domain contains M cells, then the entire domain can be decomposed into M/N blocks. We have used N=500. For operations in the horizontal (x,y), the cell constraint becomes $X \times Y \leq$ N, where X= number of cells in the x-direction, and Y= number of cells in the y-direction. For operations in both the horizontal and vertical, the constraint becomes $X \times Y \times Z \leq$ N, where Z= number of cells in the z-direction. There may be some operations, such as for some horizontal advection schemes, where this decomposition into blocks becomes more difficult or impossible.

18.8.3 Science Process Code Template

The following demonstrates what a science process class-driver Fortran 77 subroutine might look like. We recommend that a code developer follows this template, where appropriate, to get maximum benefit from the design concepts implemented in the Models-3/CMAQ system. This template is generic and attempts to show most, if not all the features available. Some class-drivers and most other sub-programs within a module may not have, nor require, most or any of these features. (The numerals at the left-hand margin are for the legend and are not part of the text, and the text within “< >“ indicates code not included.)

Example of Science Process Class-driver

```
( 1)       SUBROUTINE VDIFF SUBST_GRID_ID ( CGRID, JDATE, JTIME, TSTEP )
( 2) C-------------------------------------
( 2) C Function:
( 2)
( 2) C Preconditions:
( 2)
( 2) C Subroutines and Functions Called:
( 2)
( 2) C Revision History:
( 2) C-------------------------------------
( 3)       IMPLICIT NONE

( 4)       INCLUDE SUBST_HGRD_ID   ! horizontal dimensioning parameters
( 4)       INCLUDE SUBST_VGRD_ID   ! vertical dimensioning parameters

( 5)       INCLUDE SUBST_GC_SPC    ! gas chemistry species table
( 5)       INCLUDE SUBST_GC_EMIS   ! gas chem emis surrogate names and map table
( 5)       INCLUDE SUBST_GC_DEPV   ! gas chem dep vel surrogate names and map table
( 5)       INCLUDE SUBST_GC_DDEP   ! gas chem dry dep species and map table
( 5)       INCLUDE SUBST_GC_DIFF   ! gas chem diffusion species and map table

( 5)       INCLUDE SUBST_AE_SPC    ! aerosol species table
( 5)       INCLUDE SUBST_AE_DEPV   ! aerosol dep vel surrogate names and map table
( 5)       INCLUDE SUBST_AE_DDEP   ! aerosol dry dep species and map table
( 5)       INCLUDE SUBST_AE_DIFF   ! aerosol diffusion species and map table

( 5)       INCLUDE SUBST_NR_SPC    ! non-reactive species table
( 5)       INCLUDE SUBST_NR_EMIS   ! non-react emis surrogate names and map table
( 5)       INCLUDE SUBST_NR_DEPV   ! non-react dep vel surrogate names and map table
( 5)       INCLUDE SUBST_NR_DDEP   ! non-react dry dep species and map table
( 5)       INCLUDE SUBST_NR_DIFF   ! non-react diffusion species and map table

( 5)       INCLUDE SUBST_TR_SPC    ! tracer species table
( 5)       INCLUDE SUBST_TR_EMIS   ! tracer emis surrogate names and map table
( 5)       INCLUDE SUBST_TR_DEPV   ! tracer dep vel surrogate names and map table
( 5)       INCLUDE SUBST_TR_DDEP   ! tracer dry dep species and map table
( 5)       INCLUDE SUBST_TR_DIFF   ! tracer diffusion species and map table

( 6) #ifdef emis_vdif
( 6)       INCLUDE SUBST_EMPR_VD   ! emissions processing in vdif
( 6) #else
( 6)       INCLUDE SUBST_EMPR_CH   ! emissions processing in chem
```

Example of Science Process Class-driver

```
( 6) #endif
( 7)       INCLUDE SUBST_PACTL_ID  ! PA control parameters
( 7)       INCLUDE SUBST_CONST     ! constants
( 7)       INCLUDE SUBST_FILES_ID  ! file name parameters
( 7)       INCLUDE SUBST_IOPARMS   ! I/O parameters definitions
( 7)       INCLUDE SUBST_IOFDESC   ! file header data structure
( 7)       INCLUDE SUBST_IODECL    ! I/O definitions and declarations
( 7)       INCLUDE SUBST_XSTAT     ! M3EXIT status codes
( 4)       INCLUDE SUBST_COORD_ID  ! coordinate and domain definitions (req IOPARMS)

( 8) C Arguments:
( 8)
( 8)       REAL         CGRID( NCOLS,NROWS,NLAYS,* )  !  concentrations
( 8)       INTEGER      JDATE        ! current model date, coded YYYYDDD
( 8)       INTEGER      JTIME        ! current model time, coded HHMMSS
( 8)       INTEGER      TSTEP( 3 )   ! time step vector (HHMMSS)
( 8)                                 ! TSTEP(1) = local output step
( 8)                                 ! TSTEP(2) = sciproc sync.  step (chem)
( 8)                                 ! TSTEP(3) = advection time step

( 9) C Parameters:
( 9)       <   >
(10) C External Functions not previously declared in IODECL3.EXT:
(10)
(10)       INTEGER      SECSDIFF, SEC2TIME, TIME2SEC
(10)       EXTERNAL     SECSDIFF, SEC2TIME, TIME2SEC

(11) C File variables:
(11)       <   >
(12) C Local variables:
(12)       <   >
(13)       IF ( FIRSTIME ) THEN
(13)
(13)          FIRSTIME = .FALSE.
(13)
(13)          LOGDEV = INIT3()
(13)
(13) C Open the met files:
(13)
(13)          IF ( .NOT.  OPEN3( MET_CRO_3D, FSREAD3, PNAME ) ) THEN
(13)             XMSG = 'Could not open '// MET_CRO_3D // ' file'
(13)             CALL M3EXIT( PNAME, JDATE, JTIME, XMSG, XSTAT1 )
(13)             END IF
(13)
(13)          < open other met files >
(13)
(13) C Open Emissions files:
(13)
(13)          < do other intialization or operations that need be done only once >
(13)
(13)          END IF          !  if firstime

(14) C set file interpolation to middle of time step
(14)
(14)       MDATE = JDATE
(14)       MTIME = JTIME
(14)       MSTEP = TIME2SEC( TSTEP( 2 ) )
(14)       CALL NEXTIME ( MDATE, MTIME, SEC2TIME( MSTEP / 2 ) )

(15) C read&interpolate met data
(15)
(15)      VNAME = 'DENSA_J'
(15)      IF ( .NOT.  INTERP3( MET_CRO_3D, VNAME, PNAME,
(15)     &                     MDATE, MTIME, NCOLS*NROWS*NLAYS,
(15)     &                     RRHOJ ) ) THEN
(15)         XMSG = 'Could not interpolate DENSA_J from ' // MET_CRO_3D
(15)         CALL M3EXIT( PNAME, MDATE, MTIME, XMSG, XSTAT1 )
(15)         END IF
(15)
```

Example of Science Process Class-driver

```
(15)       < do other reads >
(15)
(15) C read&interpolate deposition velocities
(15)
(15)       < do operations >
(15)
(15) C read&interpolate emissions
(15)
(15)       CALL RDEMIS ( MDATE, MTIME, NCOLS, NROWS, EMISLYRS, NEMIS, EM_TRAC,
(15)     &               VDEMIS )
(15)
(16)       IF (LIPR) CALL PA_UPDATE_EMIS ( ‘VDIF’, VDEMIS, JDATE, JTIME, TSTEP )

           < do other operations >

(17)       CALL EDYINTB SUBST_GRID_ID ( EDDYV, DT, JDATE, JTIME, TSTEP( 2 ) )

(18)       DO 345 R = 1, NROWS
(18)          DO 344 C = 1, NCOLS

                 < do operations >

(19)             DO 301 N = 1, NSTEPS( C,R )

                    < do operations >

(19) 301         CONTINUE      !  end time steps loop
(18) 344      CONTINUE         !  end loop on col C
(18) 345   CONTINUE            !  end loop on row R

(20) C If last call this hour:  write accumulated depositions:
(20)
(20)       WSTEP = WSTEP + TIME2SEC( TSTEP( 2 ) )
(20)       IF ( WSTEP .GE.  TIME2SEC( TSTEP( 1 ) ) ) THEN
(20)          MDATE = JDATE
(20)          MTIME = JTIME
(20)          CALL NEXTIME( MDATE, MTIME, TSTEP( 2 ) )
(20)          WSTEP = 0
(20)
(20)          IF ( .NOT.  WRITE3( CTM_DRY_DEP_1,
(20)     &                        ALLVAR3, MDATE, MTIME, DDEP ) ) THEN
(20)          XMSG = ‘Could not write ‘ // CTM_DRY_DEP_1 // ‘ file’
(20)          CALL M3EXIT( PNAME, MDATE, MTIME, XMSG, XSTAT1 )
(20)          END IF
(20)
(20)          WRITE( LOGDEV, ‘( /5X, 3( A, :, 1X ), I8, “:”, I6.6 )’ )
(20)      &          ’Timestep written to’, CTM_DRY_DEP_1,
(20)      &          ’for date and time’, MDATE, MTIME
(20)
(16)          IF (LIPR) CALL PA_UPDATE_DDEP ( ‘VDIF’, DDEP, JDATE, JTIME, TSTEP )
(20)
(20)       < do other operations >
(20)
(20)       END IF

(21)       RETURN
(21)       END
```

Legend:

(1) - Class-driver subroutine internal name. Note the calling argument list. It is fixed for class-drivers. The string “SUBST_GRID_ID” is used for developing run-time nesting applications, which has not been fully implemented in the current version. By specifying subroutine names associated with a particular nest and using corresponding domain include files for that subroutine, it is possible to set up a multi-nest application. The details have not been discussed because this kind of nesting has not been implemented in the current release of the Models-3/CMAQ system.
(2) - Header comments. Highly recommended for internal documentation.
(3) - Highly recommended for Fortran. IMPLICIT NONE will catch typo’s and other undeclared

Example of Science Process Class-driver

```
variable errors at compile time.
( 4) - Domain array dimensioning and looping global variables, e.g.  NCOLS, NROWS, NLAYS.
( 5) - Chemical mechanism array dimensioning and looping global variables.
( 6) - C Preprocessor flags that determine which emissions control dimensioning and looping
variables are compiled.
( 7) - Other global array dimensioning and looping global variables including those for the I/O
API.  The logical variable LIPR is defined in the SUBST_PACTL_ID include file for use at lines
labeled (16).
( 8) - Declarations for the argument list (standardized).
( 9) - Declarations and PARAMETER statements for local Fortran parameters.
(10) - Declarations for external functions not previously declared.
(11) - Declarations for arrays to hold external file data.
(12) - Declarations for local variables.
(13) - Code section for subroutine initialization and for any local data that need not be set at
every entry into the subroutine.  Such data would require a SAVE statement in the declarations.
For example FIRSTIME is initialized to .TRUE. in the local variables section (12).
(14) - Illustrates using an I/O API function to set file interpolation time.
(15) - Read accesses from I/O API files using the I/O API time-interpolation function.
(16) - Call to process analysis routine to obtain data for the optional integrated process rates
function.
(17) - Illustrates call to another science process within the module.
(18) - Main computational loop over the horizontal grid.
(19) - Time step loop over sub-synchronization time step intervals.
(20) - Illustrates writing to an I/O API file within a module.
(21) - Subroutine end
```

18.8.4 Robustness and Computational Efficiency

Scientists, while working in their discipline, are generally not interested in the physical situations that don't significantly affect the phenomena they are studying and modeling. Typically, the scientist works with box models, and uses specialized data to test hypotheses, parameterizations and numerical schemes. The focus is to generate new results based on output from these relatively simple models. In trying to integrate these codes into a more general grid model, problems may arise when modeling simulations carry their formulations into regimes not explored in initial studies, therefore not properly accounted for in the grid model. The scientist who is developing codes that may go into a grid model should be cognizant of these issues and should make every effort to make the codes robust.

Another issue is code efficiency. In order to handle the pathological situations, codes like these may become inefficient, spending many cycles dealing with the "off-problem" cases. A good example is solving for the roots of a cubic polynomial whose coefficients, depending closely on the locally modeled physics, may range over broad values. So, in addition to robustness, attention should be paid to code performance. As the science improves in environmental modeling, this usually translates into increased computational complexity. Although hardware performance is continually improving, it cannot keep pace with the computational demands of new science developments. Not typically the main interest of the science developer, software performance is nevertheless a critical issue and must be addressed. Otherwise scientifically sound models will not be of much general use if they take an inordinate amount of time to execute.

18.9 Conclusion

The CMAQ system has been designed and implemented in such a way that its integration into Models-3 allows access to the extensive functionality of the Models-3 framework. Users can easily select important model options and can readily develop and execute models that apply to their requirements.

We have integrated the CMAQ codes into the Models-3 system by following the basic set of guidelines and rules presented in this chapter.

18.10 References

[1] http://interactivate.com/public/cvswebsites/cvs_toc.html, http://www.cyclic.com/cvs/, and the UNIX man pages

[2] http://www.cs.cmu.edu/People/vaschelp/Archiving/Rcs/, and the UNIX man pages

[3] Fine, S. S., W. T. Smith, D. Hwang, T. L. Turner, 1998: Improving model development with configuration management, IEEE Computational Science and Engineering, 5(1, Ja-Mr), 56-65. and http://envpro.ncsc.org/pub_files/fine1998b.html

[4] http://sage.mcnc.org/products/ioapi/

[5] ftp://ftp.unidata.ucar.edu/pub/netcdf/

[6] EPA Third-Generation Air Quality Modeling System, Models-3 Volume 9b, User Manual, Appendices, June 1998, EPA-600/R-98/069(b)

[7] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, 1991: Object-Oriented Modeling and Design, Prentice Hall

[8] Grell, G. A., J. Dudhia and D. R. Stauffer, 1994: A description of the fifth-generation Penn State/NCAR mesoscale model (MM5). NCAR Technical Note, NCAR/TN-398+STR, 122 pp.

This chapter is taken from *Science Algorithms of the EPA Models-3 Community Multiscale Air Quality (CMAQ) Modeling System*, edited by D. W. Byun and J. K. S. Ching, 1999.