



3 4456 0369182 3

OR  
**oml**

ORNL/TM-12267  
Dist. Category UC-530

**OAK RIDGE  
NATIONAL  
LABORATORY**

**MARTIN MARIETTA**

*G O O S E 1.4*  
**Generalized Object-Oriented  
Simulation Environment**

**User's Manual**

D. J. Nypaver  
M. A. Abdalla  
L. Guimaraes

OAK RIDGE NATIONAL LABORATORY  
CENTRAL RESEARCH LIBRARY  
CIRCULATION SECTION  
4590N ROOM 175  
**LIBRARY LOAN COPY**  
DO NOT TRANSFER TO ANOTHER PERSON  
If you wish someone else to see this  
report, send in name with report and  
the library will arrange a loan.  
NOV 20 1982

MANAGED BY  
MARTIN MARIETTA ENERGY SYSTEMS, INC.  
FOR THE UNITED STATES  
DEPARTMENT OF ENERGY

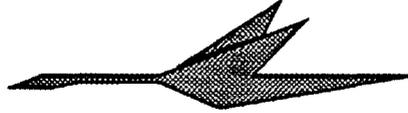
This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831; prices available from (615) 576-8401, FTS 626-8401.

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

INSTRUMENTATION AND CONTROLS DIVISION



*G O O S E 1.4*

Generalized Object-Oriented  
Simulation Environment

**USER'S MANUAL**

**D. J. Nypaver**

Oak Ridge National Laboratory  
Oak Ridge, Tennessee 37831-6285

**M. A. Abdalla**

Oak Ridge Associated Universities\*  
Post Office Box 117  
Oak Ridge, Tennessee 37831-0117

**L. Guimaraes**

Centro Tecnico Aeroespacial  
Instituto de Estudios Avanzados/ENU  
Caixa Postal 60444  
Sao Jose Campos, SP 12231, Brazil

Date Published—November 1992

Prepared by the  
**OAK RIDGE NATIONAL LABORATORY**  
Oak Ridge, Tennessee 37831-6285  
managed by  
**MARTIN MARIETTA ENERGY SYSTEMS, INC.**  
for the  
**U.S. DEPARTMENT OF ENERGY**  
under contract DE-AC05-84OR21400



3 4456 0369182 3

---

\*This research was supported in part by an appointment to the U.S. Department of Energy Laboratory Cooperative Postgraduate Research Training Program administered by the Oak Ridge Institute for Science and Education.



# PREFACE

## ABOUT THIS MANUAL

This manual describes the Generalized Object-Oriented Simulation Environment (GOOSE) and some of its basic concepts. The three basic tools of GOOSE and the various commands available in each tool are explained. This manual also describes how to model dynamic systems in GOOSE and presents a model of a Westinghouse pressurized water reactor as an example.

## SUMMARY OF CONTENTS

This manual introduces GOOSE as a new and innovative simulation tool. A brief introduction is given about object-oriented programming and Objective-C, the language in which GOOSE is written. Some tips are given on how to approach modeling in GOOSE. The basic tools of GOOSE (Class Developer, Environment Builder, and Runtime Environment) are described. An explanation on how to access the simulation environment on the Advanced Controls Program's SUN network is discussed, as well as the installation of GOOSE on a personal computer. All commands available in each GOOSE tool are explained in detail, and, finally, an example of a Westinghouse pressurized water reactor model is given. A glossary with a summary of GOOSE-related terms is given in Appendix C.

## HOW TO USE THIS MANUAL

It is advisable that the user read Chaps. 1 and 2 before starting to develop models in GOOSE. Chapter 1, *Introduction*, introduces GOOSE and its fundamental concepts. Chapter 2, *Basic Steps for Creating a Model with GOOSE*, gives pointers that will simplify the process of modeling in GOOSE. Chapters 3 to 6 (*Class Developer Commands*, *Environment Builder Commands*, *Runtime Simulation Environment Commands*, and *Global Variables*) should familiarize the user with commands that are available, many of which are used in the example program given in Chap. 8. Chapter 7, *Additional Features Found in GOOSE*, gives a detailed explanation of some advanced features of GOOSE.

## EXAMPLE PROGRAM

The example program presented in Chap. 8 is a model of the Westinghouse pressurized water reactor. This example and an example using the Volterra equations are available to users in both the PC and SUN versions of GOOSE. Their class definition and command files can be found in the GOOSE home directory under the *examples* subdirectory.

## SYNTAX

The syntax used in Chaps. 3 to 5 (*Class Developer Commands*, *Environment Builder Commands*, and *Runtime Simulation Environment Commands*) and the various options associated with each command are explained in Appendix A.

## ASSUMPTIONS

Readers are assumed to be proficient in the C programming language. The syntax of commands unique to Objective-C used in GOOSE are explained in this manual. An understanding of X Windows concepts would also be helpful when accessing the plotting and graphics objects provided with the SUN version of GOOSE.

## SOFTWARE USED IN GOOSE

GOOSE requires the installation of Version 4.3 of Stepstone's Objective-C compiler and ICpak 101 on both the DOS and UNIX platforms. The Microsoft C and FORTRAN compilers are required for the PC version of GOOSE, and SUN FORTRAN is required for the SUN version.

If the graphics provided in the SUN version of GOOSE are to be used, X Windows must be installed. Some of the dynamic graphics use VI Corporation's DataViews, but *neither of these packages is required* to run GOOSE. They are only necessary if you use some of the graphics objects described in Chap. 7.

## ACKNOWLEDGMENTS

The authors would like to acknowledge the Advanced Controls Program, Office of Reactor Technologies Development, U.S. Department of Energy, under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc., for funding this endeavor.

The authors would also like to acknowledge C. E. Ford for developing the GOOSE methodology and for his initial program development and C. March-Leuba for contributing his simulation expertise and other development efforts to GOOSE.

# CONTENTS

LIST OF FIGURES .....	ix
ABSTRACT .....	x
1. INTRODUCTION .....	1
1.1 WHAT IS GOOSE? .....	1
1.2 THE GOOSE DIFFERENCE .....	1
1.3 BRIEF INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING ..	2
1.4 BUILDING CLASS DEFINITIONS .....	3
1.5 BUILDING A SIMULATION ENVIRONMENT .....	4
1.6 CREATING A SIMULATION MODEL .....	7
1.7 GOOSE LIBRARY .....	7
1.8 ACCESSING THE GOOSE SOFTWARE ON THE ACP SUN NETWORK ..	7
1.9 INSTALLING AND USING THE GOOSE SOFTWARE ON A PC .....	8
2. BASIC STEPS FOR CREATING A MODEL WITH GOOSE .....	9
2.1 STEP 1 — DEFINE THE COMPONENTS OF THE MODE .....	9
2.2 STEP 2 — DEFINE THE NECESSARY CLASSES FOR THE COMPONENTS .....	10
2.3 STEP 3 — COMPILE THE CLASS DEFINITIONS .....	10
2.4 STEP 4 — CREATE THE SIMULATION ENVIRONMENT .....	10
2.5 STEP 5 — WRITE A COMMAND FILE TO GENERATE THE MODEL ..	10
3. CLASS DEVELOPER COMMANDS .....	11
3.1 chdir .....	11
3.2 CLASS .....	11
3.3 comment(#) .....	12
3.4 compile .....	12
3.5 DERIVMETHOD .....	12
3.6 DESCRIBE .....	14
3.7 DIGIMETHOD .....	14
3.8 DYNAMETHOD .....	16
3.9 edit .....	18
3.10 exit .....	18
3.11 flags .....	18
3.12 HEADER .....	19
3.13 help .....	19
3.14 INITMETHOD .....	20
3.15 METHOD .....	21
3.16 read .....	21
3.17 reset .....	21
3.18 REFERENCES .....	22
3.19 save .....	22
3.20 shell(!) .....	22
3.21 show .....	23
3.22 showcwd .....	23
3.23 showfs .....	23

3.24	SLOTS	24
3.25	VALIDATE	24
3.26	WHENDO	25
4.	ENVIRONMENT BUILDER COMMANDS	27
4.1	build	27
4.2	buildsm	27
4.3	chdir	28
4.4	comment(#)	28
4.5	describe	28
4.6	exit	29
4.7	flags	29
4.8	help	29
4.9	include	30
4.10	read	30
4.11	reset	30
4.12	shell(!)	31
4.13	show	31
4.14	showcwd	31
4.15	showfs	32
5.	RUNTIME SIMULATION ENVIRONMENT COMMANDS	33
5.1	assign(=)	33
5.2	call	33
5.3	chdir	34
5.4	comment(#)	34
5.5	connect	34
5.6	continue	35
5.7	create	35
5.8	createT1	35
5.9	DEFINE	36
5.10	delete	36
5.11	describe	37
5.12	edit	37
5.13	exit	37
5.14	flags	38
5.15	help	38
5.16	hold	39
5.17	initialize	39
5.18	load	39
5.19	output	40
5.20	panelobjs	40
5.21	plot	41
5.22	range	41
5.23	read	42
5.24	reset	42
5.25	run	42
5.26	save	42
5.27	send	43
5.28	shell(!)	43
5.29	show	43

5.30	showcwd	44
5.31	syntax	44
5.32	validate	44
5.33	waterp [ <machine-name> ]	45
5.34	writeto	45
6.	GLOBAL VARIABLES	46
6.1	abstol	46
6.2	derivSrtFlg	46
6.3	dynaSrtFlg	46
6.4	circularFlg	46
6.5	dt	46
6.6	euler	46
6.7	h0	47
6.8	hcur	47
6.9	hmax	47
6.10	hmin	47
6.11	hu	47
6.12	imxer	47
6.13	initSrtFlg	47
6.14	iopt	47
6.15	istate	48
6.16	itask	49
6.17	itol	49
6.18	ixpr	49
6.19	mcurl	50
6.20	mf	50
6.21	mused	50
6.22	mxordn	50
6.23	mxords	50
6.24	mxstep	50
6.25	neqns	50
6.26	nfe	51
6.27	nhold	51
6.28	nje	51
6.29	noutput	51
6.30	nperiods	51
6.31	nplot	51
6.32	nqcur	51
6.33	nqu	51
6.34	nst	51
6.35	orgSort	52
6.36	plotmax	52
6.37	ppcpu	52
6.38	reitol	52
6.39	showop	52
6.40	t	52
6.41	tcur	52
6.42	tend	53
6.43	tolsf	53
6.44	trace	53

6.45	tstart	53
6.46	tsw	53
6.47	t0	53
7.	ADDITIONAL FEATURES FOUND IN GOOSE	54
7.1	DIFFERENTIAL EQUATION SOLVERS	54
7.2	DYNAMIC PLOTS	54
7.3	REAL-TIME SIMULATOR	56
7.4	EIGENVALUE CALCULATIONS	56
7.5	INTERACTIVE EDITING	57
7.6	VECTORS	57
8.	GOOSE SAMPLE PROGRAM	60
8.1	THE PROBLEM	60
8.2	THE CLASS DEFINITION	60
8.3	THE CLASS DEVELOPER	64
8.4	THE ENVIRONMENT BUILDER	64
8.5	THE RUNTIME ENVIRONMENT	65
	REFERENCES	70
	APPENDIX A. COMMAND SYNTAX	71
	APPENDIX B. ERROR MESSAGES	72
	APPENDIX C. GLOSSARY	77

# LIST OF FIGURES

Figure 1.	Relationship between the three GOOSE tools. . . . .	2
Figure 2.	Relationships between a class, an object, methods, messages, and slots. . .	4
Figure 3.	Dynamic plots created in the SUN version of GOOSE. . . . .	5
Figure 4.	Nondynamic plots created in the SUN version of GOOSE. . . . .	6
Figure 5.	Nondynamic plots created in the PC version of GOOSE. . . . .	6
Figure 6.	A diagram of a low-order model of a Westinghouse pressurized water reactor. . . . .	9
Figure 7.	An example of a graphical user interface created to interact with a GOOSE simulation model. . . . .	17
Figure 8.	The Plot Manager Window is used for setting up dynamic plots in the SUN version of GOOSE. . . . .	55
Figure 9.	Interactive edit panels created in the SUN version of GOOSE. . . . .	58



# ABSTRACT

The Generalized Object-Oriented Simulation Environment (GOOSE) is a new and innovative simulation tool that is being developed by the Simulation Group of the Advanced Controls Program at Oak Ridge National Laboratory. GOOSE is a fully interactive prototype software package that provides users with the capability of creating sophisticated mathematical models of physical systems. GOOSE uses an object-oriented approach to modeling and combines the concept of modularity (building a complex model easily from a collection of previously written components) with the additional features of allowing precompilation, optimization, and testing and validation of individual modules. Once a library of components has been defined and compiled, models can be built and modified without recompilation. This user's manual provides detailed descriptions of the structure and component features of GOOSE, along with a comprehensive example using a simplified model of a pressurized water reactor.



# 1. INTRODUCTION

## 1.1 WHAT IS GOOSE?

The Generalized Object-Oriented Simulation Environment (GOOSE)<sup>1,2</sup> is a fully interactive prototype software package that is being developed by the Simulation Group of the Advanced Controls Program at Oak Ridge National Laboratory. GOOSE is a software package that provides users with the capability of creating sophisticated mathematical models of physical systems. These models may involve discrete-time behavior as well as behavior governed by ordinary differential equations. GOOSE provides access to powerful tools such as numerical integration packages, eigenvalue and Jacobian matrix calculations, graphical displays, and on-line help. In GOOSE, portability has been achieved by creating the environment in Objective-C,<sup>3</sup> which is supported by a variety of platforms, including UNIX and DOS. GOOSE Version 1.4 is primarily command-line driven.

The GOOSE software package consists of three basic tools: a Class Developer, an Environment Builder, and a Runtime Environment. These tools are independent of each other and are to be used in sequence. Figure 1 illustrates the relationship between the three GOOSE tools. The Class Developer allows the user to create classes that model the system components. The Environment Builder creates an executable simulation environment that includes the classes specified by the user that are needed for the simulation. The Runtime Environment allows the user to build, modify, execute, and test the model through the dynamic creation, connection, modification, and deletion of model objects from the classes included in the environment. On-line help is available in all three GOOSE tools.

## 1.2 THE GOOSE DIFFERENCE

GOOSE uses an object-oriented approach to modeling. GOOSE combines the concept of modularity (building a complex model easily from a collection of previously written components) with the additional features of allowing precompilation, optimization, and testing and validation of individual modules. Once a library of components has been defined and compiled, models can be built and modified without recompilation. Dynamic models are easily constructed and tested. The fully interactive capabilities of GOOSE allow the user to alter model parameters and complexity without recompilation. Currently, two differential equation solvers, the complex Livermore solver for ordinary differential equations with root-finding (Isodar) and the simpler Euler method, are available in GOOSE.

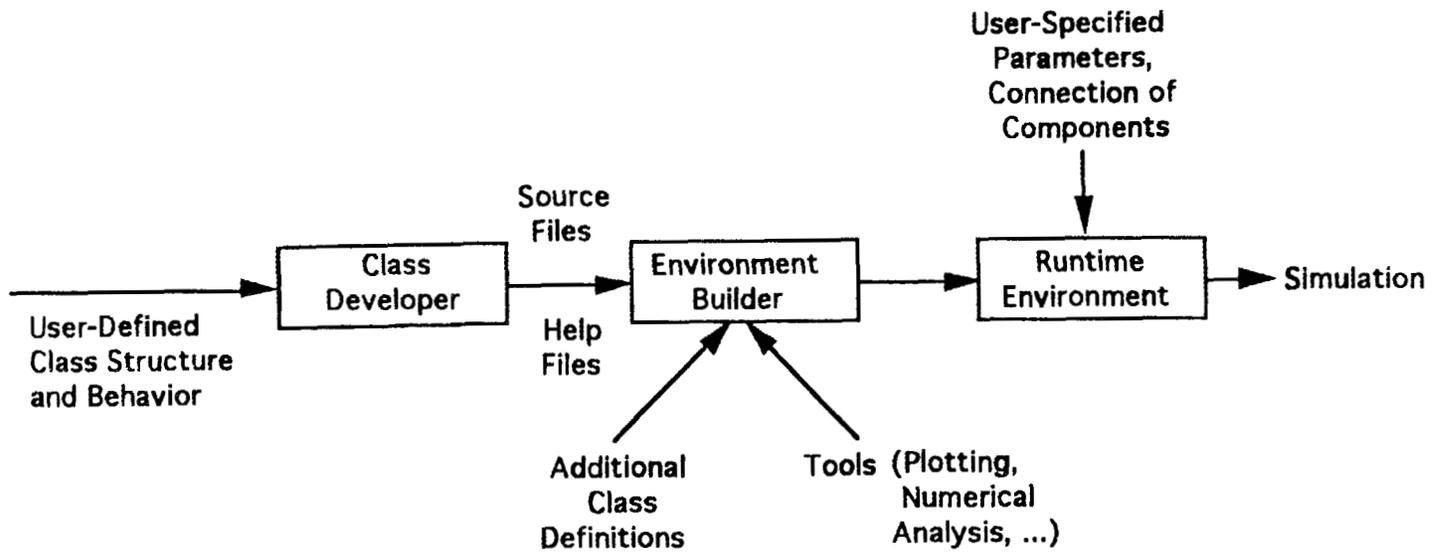


Fig. 1. Relationship between the three GOOSE tools.

## 1.3 BRIEF INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING

To use GOOSE, one needs a brief introduction to object-oriented programming terminology.<sup>4</sup> In object-oriented programming systems, the processing of information is done through *objects*. An object is a particular instance of a *class*. All objects within a class share a common *definition*, which is the *data structure* and *behavior* of the class. Objects are manipulated by sending them *messages*. The class definition is a generic description of the object and how it behaves. In the class definition, the functions that implement the behavior of the various messages are called *methods*, and the variables that implement the structure of the class are called *slots*. All classes belong to a hierarchy. Through the mechanism of *inheritance*, objects belonging to subclasses may automatically *inherit* the data structures, messages, and methods of their *superclass*, or parent class. GOOSE allows a user to create a model, which is just a combination of defined objects. A user can dynamically *create*, *delete*, *edit*, and *connect* objects in the model without recompilation.

Any C language syntax is valid in Objective-C. A different construct in the Objective-C language that a user may notice in GOOSE is the *[object message]* syntax. This is the code for sending messages to objects. For example, the code *[valve1 trip]* sends the object *valve1* the message *trip*. This executes the method, *trip*, defined for that object.

In the equation,  $\text{flow\_rate} = C_v * [\text{valve1 valve\_pos}] * \text{sqrt}(\text{delta\_P})$ , the *[valve1 valve\_pos]* value is produced by sending the object, *valve1*, the message, *valve\_pos*, which returns the value of the slot *valve\_pos* in the object *valve1*. This syntax construct is used to communicate between objects. Figure 2 illustrates the relationship between a class, an object, methods, messages, and slots.

Another Objective-C syntax used in GOOSE commands is *object.slot*, where the object name is followed by a period and then a defined slot name for that object. This is a naming convention for the slot.

## 1.4 BUILDING CLASS DEFINITIONS

GOOSE provides a Class Developer to ease the task of developing class definitions in an object-oriented language. The Class Developer enables the user to specify only a minimal description of the model components, their structure and behavior. The Class Developer automatically produces the necessary class definition files, source code files, and on-line help files for each class definition it *reads* and *saves*. These files are available to the user and to other GOOSE tools. The purpose of the class definition is to define the structure and behavior of objects that belong to the class.

The user may create the class definition in a text file, using a text editor, or through a series of commands entered on the command line while inside the Class Developer. If a class is defined in a file using a text editor, it must be read into the Class Developer using the *read* command. Once a class has been defined in the Class Developer, the *save* command saves the class definition in the Objective-C files needed by the other GOOSE tools. The class is then compiled using the *compile* command, after which the user can create a customized simulation environment using the GOOSE Environment Builder.

Recompilation of a class is necessary only if its class definition is changed. Once a class definition is compiled, it can be used to build any model. Inside the Runtime Environment, objects can be dynamically added and deleted, and parameter values can interactively be changed without recompilation.

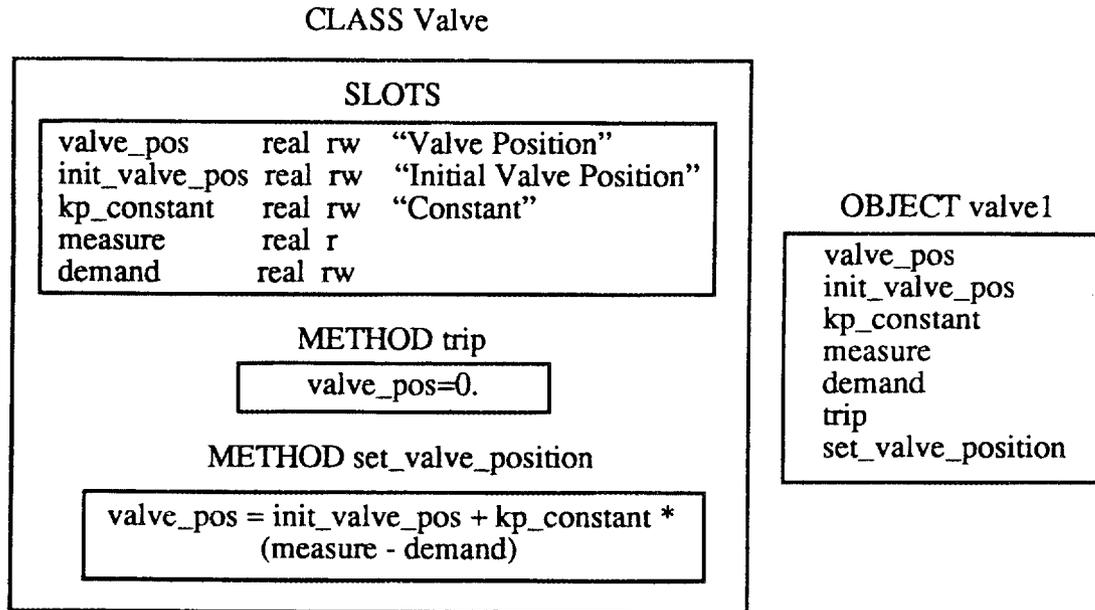


Fig. 2. Relationships between a class, an object, methods, messages, and slots.

## 1.5 BUILDING A SIMULATION ENVIRONMENT

The Environment Builder is used to create a customized simulation environment to build models from the classes defined with the Class Developer. The *include* command specifies the class definitions that need to be available in the simulation environment. All other necessary files and class definitions will be loaded automatically by GOOSE.

In the SUN version, the *build* or *buildsm* (build small model) commands create the environment in the executable file specified. The *build* command includes all the plotting libraries required for the dynamic (on-line) DataViews plots. The dynamic or real-time plotting features of GOOSE enable users to observe plots of selected model parameters as the simulation runs. The executable files made with the *build* command must run on machines with DataViews, X Windows, and Objective-C installed. The *buildsm* command does not include the DataViews plotting package, but nondynamic (off-line) plots can still be done using the plotting package TempleGraph. Nondynamic plots are displayed only after the simulation is paused or finished. The executables made with the *buildsm* command only require the installation of Objective-C, but in order to have the nondynamic plotting capabilities, the installation of TempleGraph is also required. Figure 3 illustrates the dynamic plots available in the SUN version of GOOSE. Figures 4 and 5 depict the nondynamic plots available in the SUN and PC versions of GOOSE, respectively.

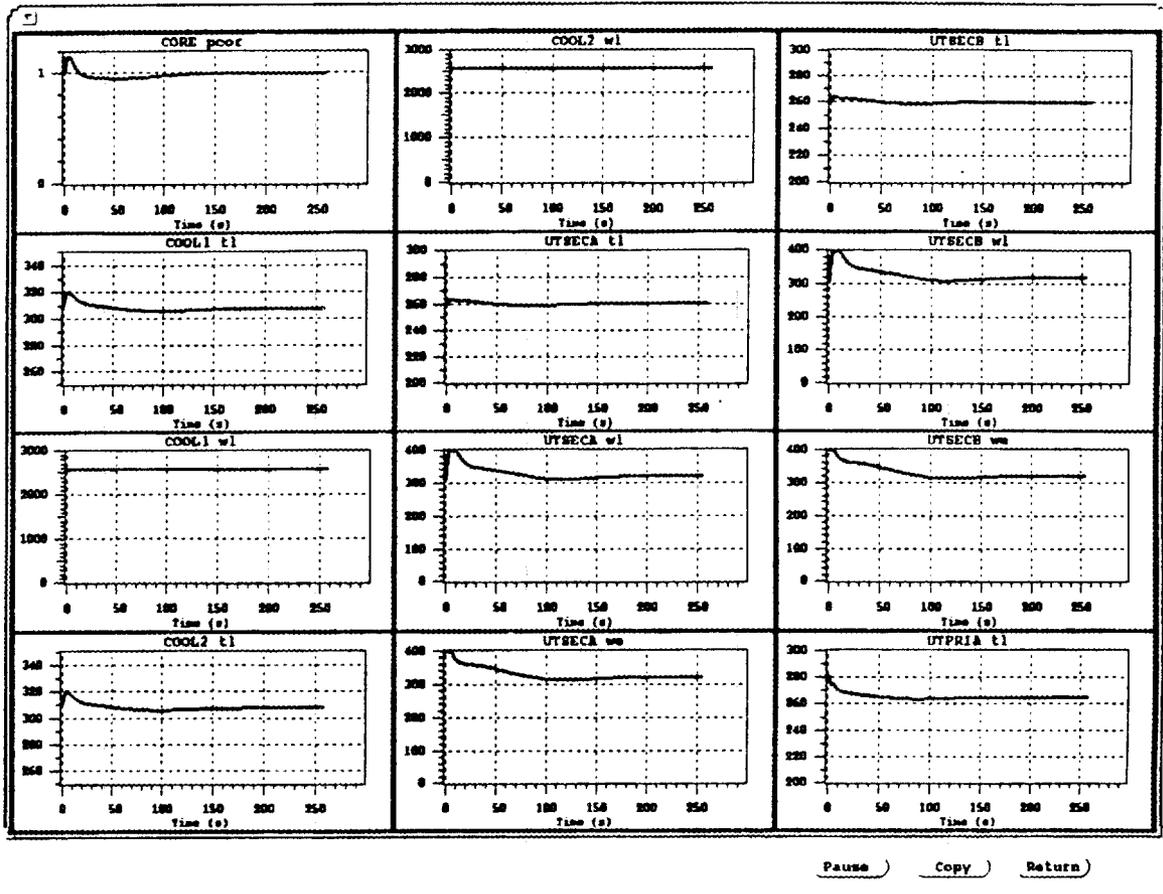


Fig. 3. Dynamic plots created in the SUN version of GOOSE.

### Simulation Data

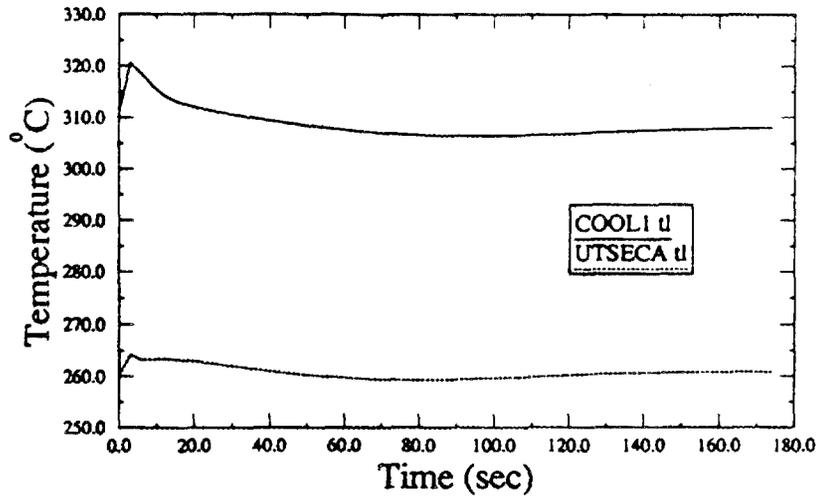


Fig. 4. Nondynamic plots created in the SUN version of GOOSE.

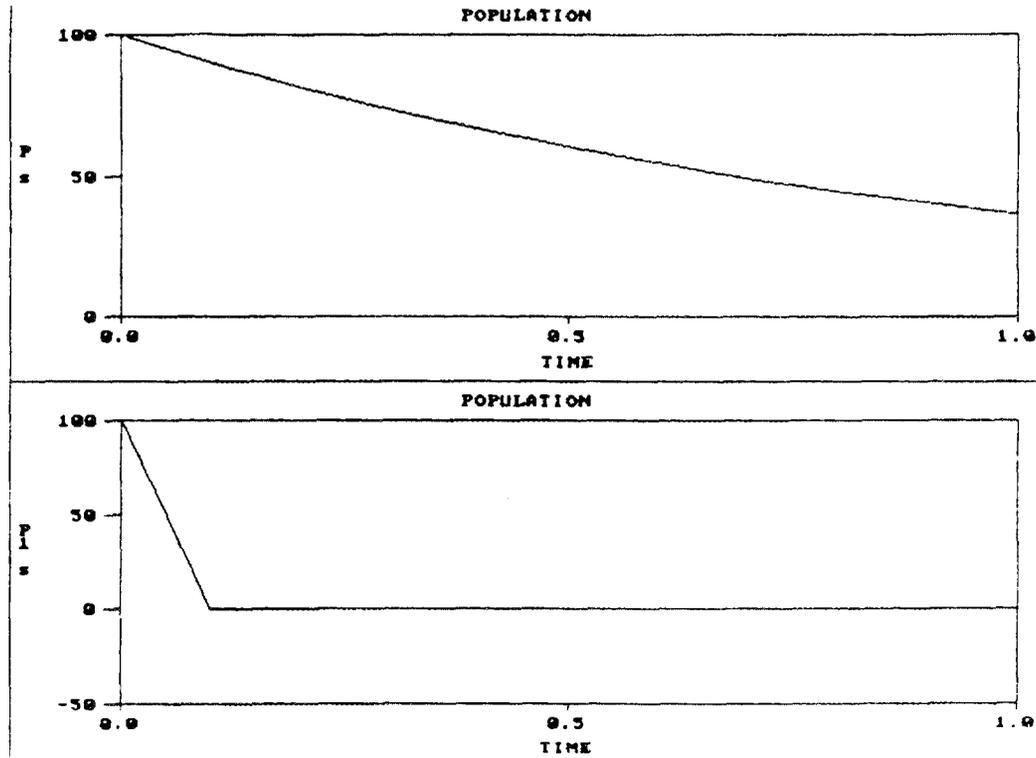


Fig. 5. Nondynamic plots created in the PC version of GOOSE.

Since dynamic plots are currently not available in the PC version, the PC *build* and *buildsm* commands perform the same function.

## 1.6 CREATING A SIMULATION MODEL

In GOOSE, a model is a combination of defined objects. These objects are created from the classes included in the environment. For example, if you have a class Valve (Fig. 2) included in the environment, one could create as many different valve objects from that class as are needed in the simulation. A user can *create*, *delete*, *edit*, and *connect* objects and can *save*, *load*, and *reset* the model in the Runtime Environment. The Runtime Environment can *read* commands from the command line or from a command file.

## 1.7 GOOSE LIBRARY

GOOSE provides the user with a library of classes that are accessible in the Runtime Environment. The classes that are currently available in the GOOSE library are listed below.

<u>Class Name</u>	<u>Description</u>
1. DspPanel	Interactive Edit Panel available in the SUN version of GOOSE
2. LinearA	Enables Jacobian matrix and eigenvalue calculations
3. PlotDV	Dynamic plots available in the SUN version of GOOSE
4. RealTime	Real-time simulator

The graphical classes, DspPanel and PlotDV, are only available if the user builds his Runtime Environment with the *build* command in the SUN version of GOOSE. The LinearA and RealTime classes are always available to the user in the Runtime Environment in both the SUN and PC versions of GOOSE. For more information on these classes, refer to Chap. 7.

## 1.8 ACCESSING THE GOOSE SOFTWARE ON THE ACP SUN NETWORK

To have access to the GOOSE software on the ORNL Advanced Controls Program's (ACP's) SUN network, add the following lines to your *.cshrc* file:

```
setenv GOOSEHOME /usr6/users/ACP/goose  
set path=($GOOSEHOME/bin $path )
```

To use the *build* command in the Environment Builder, add

```
setenv DVHOME /usr/local/dvhome
```

to your *.cshrc* file.

In order to enable these changes, either log off and back on or type

```
source .cshrc
```

from your home directory.

## **1.9 INSTALLING AND USING THE GOOSE SOFTWARE ON A PC**

To install the GOOSE software on a PC, create a GOOSE home directory on a hard drive. Then copy the files from the GOOSE floppies into the home GOOSE directory.

To have access to the GOOSE software, include the *bin* directory under the GOOSE home directory in your path. Make sure that your path is set up for the Objective-C, Microsoft C, and FORTRAN compilers and libraries.

In order for the GOOSE software to run, you must set the environment variable GOOSEHOME to the appropriate directory in your *autoexec.bat* file. In our case, we have

```
set GOOSEHOME=d:\goose
```

A GOOSE PC version that uses PharLap's DOS extender software is also available. This version enables the user to compile and run large models. To run this version of GOOSE, set the environment variable DOSXTNDR to run286 in your *autoexec.bat* file as follows,

```
set DOSXTNDR=run286
```

## 2. BASIC STEPS FOR CREATING A MODEL WITH GOOSE

### 2.1 STEP 1 — DEFINE THE COMPONENTS OF THE MODEL

The first step in creating a simulation is to clearly define the model components and their interconnections. Constructing a flow diagram is a good way to plan the arrangements and connections of all of the objects in the model. As mentioned in Sect. 1.3, each object belongs to a “class.” All objects within a class share a common “data structure” and “behavior.” In the definition of a class, the data structure includes the variables on which the class depends; these variables are called “slots.”

Figure 6 illustrates a diagram of a low-order model of a Westinghouse pressurized water reactor. Each box in the figure represents an object in the model and belongs to one of the five classes described in Sect. 8.1. The arrows represent the connections between the objects. The objects in the diagram are created and connected in the Runtime Environment.

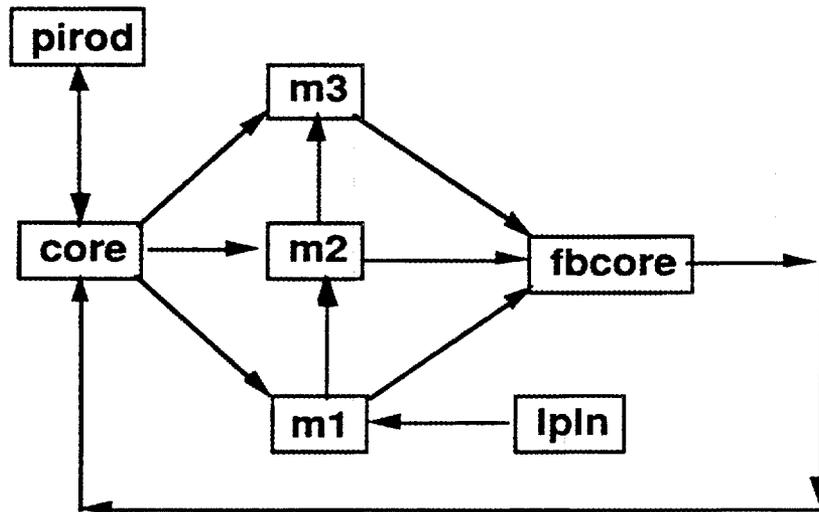


Fig. 6. A diagram of a low-order model of a Westinghouse pressurized water reactor.

## **2.2 STEP 2 — DEFINE THE NECESSARY CLASSES FOR THE COMPONENTS**

Write class definitions for each of the different types of components needed in the model. The class definition is used to describe the structure and behavior of a component. The interdependence between components should be kept in mind when identifying those objects that need information to be “received from” and/or “sent to” other objects in the model. For more information on the class definition, see Sect. 8.2.

## **2.3 STEP 3 — COMPILE THE CLASS DEFINITIONS**

Use the Class Developer to read the class definition files, save them into their corresponding Objective-C files, and compile the definitions so that they may be used in any Runtime Environment. Section 8.3 explains how to use the Class Developer and gives an example.

## **2.4 STEP 4 — CREATE THE SIMULATION ENVIRONMENT**

Use the Environment Builder to build an executable simulation environment from the classes defined and compiled with the Class Developer. Section 8.4 explains how to use the Environment Builder.

## **2.5 STEP 5 — WRITE A COMMAND FILE TO GENERATE THE MODEL**

After an executable simulation environment is created, a command file is written for the Runtime Environment. This file is used to create, initialize, and connect objects into a model. Section 8.5 contains an example of a Runtime Environment command file.

## 3. CLASS DEVELOPER COMMANDS

The Class Developer allows a user to define Objective-C classes by entering a set of Class Developer commands. These commands can be entered on the command line or *read* from a command file. The Class Developer informs a user of any syntax errors that may occur. To help locate syntax errors inside a command file, the flag *echo* has to be turned *on* before reading the file. For best results, after the errors have been located and corrected, *reset* the Class Developer before reading the command file again. Then the user can *save* the newly defined class into its corresponding Objective-C and GOOSE files and can compile the class for use in a Runtime Environment. It is advisable to use a command file when defining a class. An example of a Class Developer command file can be found in Sect. 8.2. Appendix A explains the syntax used in the command descriptions that follow. Note that all the commands are case sensitive; the way the commands appear in the manual (upper or lower case) is the only way that they are recognizable to GOOSE.

---

### 3.1 **chdir**

Function: Changes the working directory.

Syntax: [ *chdir* / *cd* ] <dir-spec>

To change the current working directory while in the Class Developer, type

**chdir** <a directory name or path>

or

**cd** <a directory name or path>

---

### 3.2 **CLASS**

Function: Begins a new class definition.

Syntax: **CLASS** <name> [ **OF** <superclass> ]

To start a new class definition, type

**CLASS** <the new class name>

To create a new class and associate it with a superclass, type

**CLASS** <the new class name> **OF** <the name of the superclass>

---

### 3.3 comment(#)

Function: Allows documentation inside the Class Developer.

Syntax: #

To use the comment indicator, type

# comment

The comment indicator is the first character on each comment line.

---

### 3.4 compile

Function: Compiles the (most recently *saved*) class definition.

Syntax: *compile*

To compile the most recently *saved* class definition, type

**compile**

---

### 3.5 DERIVMETHOD

Function: Defines the source code for derivatives of the state variables.

Syntax: **DERIVMETHOD** <name>  
<Objective-C code>  
**INTEGRATE**  
<v> <dv> <v0>  
...  
**INPUTS**  
<i1> <i2> ...  
...  
**OUTPUTS**  
<o1> <o2> ...  
...  
**END**

The **DERIVMETHOD** section is used to define the model equations, the state variables to be integrated, and the input and output variables. Each **DERIVMETHOD** requires the user to identify it with a unique name. A class definition can have more than one **DERIVMETHOD**.

To define a derivative method, type

**DERIVMETHOD** <name>

The name of the derivative method is followed by the Objective-C code that needs to be executed at each integration step.

The optional **INTEGRATE** subsection provides information to the differential equation solver. To define an **INTEGRATE** subsection, type

**INTEGRATE**  
v dv v0

where *v*, *dv*, and *v0* are the scalar or array variable, its derivative, and its initial condition, respectively. To add another equation to be integrated in a **DERIVMETHOD**, insert a line containing the variable name, derivative, and initial condition in the **INTEGRATE** subsection.

To define inputs, type

**INPUTS**  
object1.x object2.z

where *x* and *z* are input variables from *object1* and *object2*, respectively. The **INPUTS** subsection is optional.

To define outputs, type

**OUTPUTS**  
w1 w2

where *w1* and *w2* are variables that are output to other objects in the Runtime Environment. The **OUTPUTS** subsection is optional.

The **INPUTS** and **OUTPUTS** subsections are used when sorting the derivative list. GOOSE cannot sort the derivative list when output variables are referenced before they are defined as inputs. In this case, a warning message indicating a circularity error is displayed. The circularity check is by default turned on; therefore, all the methods that are defined with inputs and outputs will be checked for circularity. To turn the circularity check off, set the global variable *circularFlg=0* in the Runtime Environment.

To end the derivative method definition, type

**END**

Section 8.2 gives two illustrations of **DERIVMETHOD** definitions.

### 3.6 DESCRIBE

Function: Creates descriptive text for the current class.

Syntax: **DESCRIBE**  
    *<Descriptive Text>*  
    **END**

The **DESCRIBE** section is where general information about the current class is provided. This information is displayed in the simulation environment when a user asks for a description of the class.

To use the **DESCRIBE** command, type

**DESCRIBE**

followed by any number of text lines describing the current class. To end the description type

**END**

This **DESCRIBE** text is included in the on-line help file automatically generated for the class by the Class Developer.

---

### 3.7 DIGIMETHOD

Function: Defines a digital method to be executed at a specified sampling time.

Syntax: **DIGIMETHOD** *<name>*  
    *<Objective-C code>*  
    **SAMPLING**  
    *<sl>*  
    **DTIME**  
    *<dtimel>*  
    **INPUTS**  
    *<i1> <i2> ...*  
    ...  
    **OUTPUTS**  
    *<o1> <o2> ...*  
    ...  
    **END**

Digital methods are commonly used as a communication device that needs responses at a given sampling time, not necessarily each computation step. Each **DIGIMETHOD** requires the user to identify it with a unique name. A class definition can have more than one **DIGIMETHOD**.

To define a digital method for a class, type

**DIGIMETHOD** <name>

The name of the digital method is followed by the Objective-C code that needs to be executed at each sampling time.

To define the **SAMPLING** variable, type

**SAMPLING**  
*s1*

where *s1* is the sampling time variable. The sampling time variable is initialized in either the **INITMETHOD** of the current class or with the *edit* command in the Runtime Environment. The Runtime Environment needs to associate a variable with each **DIGIMETHOD** to keep track of the last sampling time. This variable is defined in the **DTIME** subsection. To do this, type

**DTIME**  
*dtimel*

where *dtimel* is the variable that contains the digital method's last sampling time.

To define inputs, type

**INPUTS**  
*object1.x object2.z*

where *x* and *z* are the names of the input variables for the digital method from *object1* and *object2*, respectively. The **INPUTS** subsection is optional.

To define outputs, type

**OUTPUTS**  
*w1 w2*

where *w1* and *w2* are the names of the variables needed by other objects in the simulation environment. The **OUTPUTS** subsection is optional.

To end the digital method definition, type

**END**

Section 8.2 gives an illustration of a **DIGIMETHOD** definition.

---

### 3.8 DYNAMETHOD

Function: Defines a dynamic method to be executed after each time step, *dt*, which is specified by the user.

Syntax: **DYNAMETHOD** <name>  
<Objective-C code>  
**INPUTS**  
<i1> <i2> ...  
...  
**OUTPUTS**  
<o1> <o2> ...  
...  
**END**

Dynamic methods are commonly used as a communication device that needs responses at each time step, such as graphical user interfaces. Figure 7 presents a graphical user interface created to interact with a GOOSE simulation model. Each **DYNAMETHOD** requires the user to identify it with a unique name. A class definition can have more than one **DYNAMETHOD**.

To define a dynamic method, type

**DYNAMETHOD** <name>

The name of the dynamic method is followed by the Objective-C code to be executed after each time step.

To define inputs, type

**INPUTS**  
object1.x object2.z

where *x* and *z* are the names of the input variables for the dynamic method. The **INPUTS** subsection is optional.

To define outputs, type

**OUTPUTS**  
w1 w2

where *w1* and *w2* are the names of the variables needed by other objects in the simulation environment. The **OUTPUTS** subsection is optional.

To end the dynamic method definition, type

**END**

Section 8.2 gives an illustration of a **DYNAMETHOD** definition.

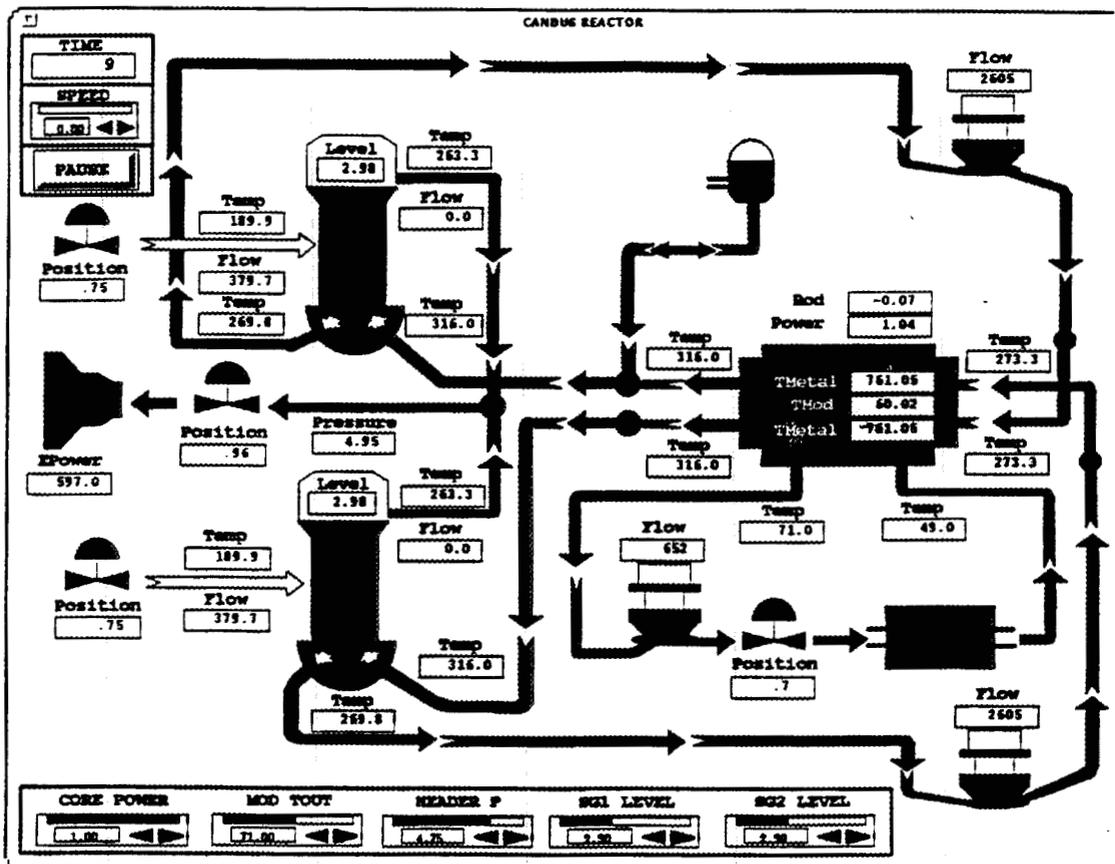


Fig. 7. An example of a graphical user interface created to interact with a GOOSE simulation model.

### 3.9 edit

Function: Edits [ a file ] (currently uses the EMACS Editor).

Syntax: *edit* [ <file> ]

To edit a file while in the Class Developer, type

**edit** <a filename>

The EMACS editor (ed editor on the PC) will be invoked to edit the file. If a filename is not specified on the command line, the editor will be invoked, but a file will not be included for editing. GNU EMACS is a public domain editor, which can be ported to SUNs.

---

### 3.10 exit

Function: Exits the Class Developer.

Syntax: [ *exit* / *halt* / *quit* / *stop* ]

To leave the Class Developer, type

**exit**

The commands *halt*, *quit*, and *stop* also exit the Class Developer and all are equivalent to the *exit* command.

---

### 3.11 flags

Function: Sets the debug or echo flag.

Syntax: [ *debug* / *echo* ] [ *on* / *off* ]

To set the debug flag on, type

**debug on**

To turn the debug flag off, type

**debug off**

Use the same syntax to turn the *echo* flag on or off. These verbose flags are off by default. If the debug flag is on, GOOSE prints out messages informing the user what routine is being executed and the current values of some variables. This flag is used to help the user debug their class definition. If the echo flag is on, each line of the command file is echoed.

---

### 3.12 HEADER

Function: Provides comments, whether Objective-C or C *include* files, and other header information for the class definition. This information will be included in all the files the Class Developer creates for the current class.

Syntax: **HEADER**  
          <Header information text>  
          **END**

To use the **HEADER** command, type

**HEADER**

followed by Objective-C comment code containing the needed header information. To end the **HEADER** section, type

**END**

---

### 3.13 help

Function: Gets on-line help for one or more commands in the Class Developer.

Syntax: [ **help** | ? ] { <command> }+

To get the available help, type

**help**  
or  
?

To get help on specific commands, type **help** or ? followed by the command name. For example,

**help validate read**

gives help on the commands *validate* and *read*.

---

### 3.14 INITMETHOD

Function: Defines a method to initialize variables for the current class.

Syntax: **INITMETHOD** <name>  
          <Objective-C code>  
          **INPUTS**  
          <i1> <i2> ...  
          ...  
          **OUTPUTS**  
          <o1> <o2> ...  
          ...  
          **END**

To define an initialization method, type

**INITMETHOD** <name>

The name of the initial method is followed by the Objective-C code that needs to be executed initially for the class being defined.

To define inputs, type

**INPUTS**  
object1.x object2.z

where *x* and *z* are the names of the input variables for the initial method. The **INPUTS** subsection is optional.

To define outputs, type

**OUTPUTS**  
w1 w2

where *w1* and *w2* are the names of the variables needed by other objects in the simulation environment. The **OUTPUTS** subsection is optional.

To end the initial method definition, type

**END**

More than one **INITMETHOD** may be defined for a class. The **INPUTS** and **OUTPUTS** are used by the Runtime Environment to determine the order of execution of the **INITMETHODS** in a model. In most cases, all **INPUTS** should be defined before they are referenced as **OUTPUTS**.

### 3.15 METHOD

Function: Defines a method or function for the current class.

Syntax: **METHOD** <name>  
          <Objective-C code>  
          **END**

To define a method for the current class, type

**METHOD** <name>

The name of the method is followed by the Objective-C code that needs to be executed. To end the method definition, type

**END**

These methods are executed when the object for which the method was defined is sent a message that represents the **METHOD**'s name. For example, *send valve1 trip* (Fig. 2), would execute the predefined method *trip* for the object *valve1*.

---

### 3.16 read

Function: Reads a file containing Class Developer commands.

Syntax: **read** <command file>

To read a command file into the Class Developer, type

**read** <a filename>

---

### 3.17 reset

Function: Clears all the previous commands issued in the Class Developer.

Syntax: **reset**

To clear the Class Developer, type

**reset**

The Class Developer needs to be reset before a new class definition is read.

---

### 3.18 REFERENCES

Function: Lists other classes referred to by the current class.

Syntax: **REFERENCES**  
    <class1> <class2> ...  
    ...  
    **END**

To reference other classes, type

**REFERENCES**

followed by the names of the referred classes. As many classes as necessary can be referenced.

To end the **REFERENCES** section, type

**END**

---

### 3.19 save

Function: Creates the help and the Objective-C source files (.h, .m, and .def) for the current class.

Syntax: **save**

To save the currently defined class in Objective-C source files, type

**save**

---

### 3.20 shell(!)

Function: Executes a system shell command and returns to the Class Developer.

Syntax: **! <shell command>**

To execute a UNIX or DOS shell command while in the Class Developer, precede the shell command string with the character **!**. For example, to get a listing of the current UNIX directory, type

**!ls**

---

### 3.21 show

Function: Displays Class Developer information.

Syntax: **show** [ *class* / *derivlist* / *describe* / *flags* / *initlist* / *dynamiclist* / *header* / *integrate* / *methods* / *references* / *digitallist* / *slots* / *whenlist* / *validate* ]+

To see current information in the Class Developer, use the *show* command. For example, to see the class and integrate list, type

**show class integrate**

---

### 3.22 showcwd

Function: Displays the current working directory.

Syntax: [ **showcwd** / **pwd** ]

Type

**showcwd**

or

**pwd**

to see the current working directory.

---

### 3.23 showfs

Function: Displays the suffixes of Class Developer files.

Syntax: **showfs** [ *def* / *des* / *int* / *imp* ]+

To see the suffixes on the definition file (*def*), description file (*des*), implementation file (*imp*), and/or interface file (*int*), use the *showfs* command. For example, to see the suffix of the implementation file, type

**showfs imp**

---

## 3.24 SLOTS

Function: Provides the class structure, which is a definition of all the variables on which the current class depends.

Syntax: **SLOTS**  
    <name> <data type> <access> [ <prompt> [ <units> [ <comment> ] ] ]  
    ...  
    **EXTERNAL**  
    <name> <data type>  
    ...  
    **INTERNAL** <name>  
    <code>  
    **END**

To define the slots for a class, type

### **SLOTS**

followed by the variable name; data type (object, real, int, char, bool, string, or vector); access (\*, rw, r, or w) and three optional fields: the prompt, the variable's units, and a comment. Note that the access character, \*, means that no object has access to the slot; the slot is *local* to the class definition. If an optional field contains a string with blanks, the string must be enclosed in quotes. To skip one optional field, use "" to indicate nothing goes in the skipped field. For example, if the units are needed, but a prompt is not, put a "" in the prompt field. This ensures that the proper information is put in the proper field; otherwise, the needed unit information will be given to the prompt field. For more information on vector types, see Sect. 7.6, *Vectors*.

The optional **EXTERNAL** section is used to reference slots and their data types that are declared in other objects but needed by this class. To define an external slot reference, type

### **EXTERNAL**

followed by the external slot's name and type.

The optional **INTERNAL** subsection declares Objective-C code to be used by this class only. To declare internal portions of code, type

### **INTERNAL** <name>

followed by lines of Objective-C or C code.

To end the slot definition section, type

### **END**

Section 8.2 gives an example of a **SLOTS** section.

### 3.25 VALIDATE

Function: Defines the list of validation rules for the current class.

Syntax: **VALIDATE**  
    <slot> { *notnil* / [ { *respondsto* / *class* / *subclass* } <arg> ] }  
    ...  
    **END**

To define a list of validation rules for a class, type

**VALIDATE**

followed by a slot name and the rule. The rules are *notnil*, *respondsto* <slot name>, *class* <name>, or *subclass* <name>. Multiple validation rules can be defined for a class.

To end the **VALIDATE** section, type

**END**

The optional **VALIDATE** section specifies a list of validation rules that applies to objects of the defined class during runtime. The validation rules ensure that objects in the model are connected properly. The *subclass* rule (*object-name subclass class-name*) specifies that the named object must be connected to another object that belongs to the specified subclass. The rule *notnil* (*object-name notnil*) means that the object must be connected. The *class* (*object-name class class-name*) rule indicates that the connected object must be a member of the named class. The rule *respondsto* (*object-name respondsto slot-or-method-name*) means that the connected object must be able to respond to or return a value for the given slot or method.

Section 8.2 gives an example of a **VALIDATE** section.

---

### 3.26 WHENDO

Function: Defines a method that tells the differential equation solver what code to execute to find a root of a constraint equation (when code) and what code to execute when the root is found (do code).

Syntax: **WHENDO** <name>  
    **WHEN**  
    <Objective-C code>  
    **DO**  
    <Objective-C code>  
    **END**

To define an optional **WHENDO** method for the current class, type

**WHENDO** <name>

Then, to define the Objective-C code that needs to be executed to find the root of a constraint equation, type

**WHEN**  
<Objective-C code>

To define the code to be executed when a root is found, type

**DO**  
<Objective-C code>

To define inputs, type

**INPUTS**  
object1.x object2.z

where *x* and *z* are the names of the input variables for the **whendo** method. The **INPUTS** subsection is optional.

To define outputs, type

**OUTPUTS**  
*w1 w2*

where *w1* and *w2* are the names of the variables needed by other objects in the simulation environment. The **OUTPUTS** subsection is optional.

To end the **WHENDO** method definition, type

**END**

More than one **WHENDO** method may be defined for a class. The **INPUTS** and **OUTPUTS** are used by the Runtime Environment to determine the order of execution of the list of **WHENDO** methods for all the objects in a model. In most cases, all **INPUTS** should be defined before they are referenced as **OUTPUTS**. A class definition can have more than one **WHENDO** method.

---

## 4. ENVIRONMENT BUILDER COMMANDS

The Environment Builder aides the user in bringing the classes compiled with the Class Developer into a simulation environment. Commands can be entered on the command line or *read* from a command file. Appendix A explains the syntax used in the command descriptions that follow. Note that all the commands are case sensitive; the way the commands appear in the manual (upper or lower case) is the only way that they are recognizable to GOOSE.

---

### 4.1 build

**Function:** Builds (creates) a runtime simulation environment on the specified file. This environment will include the DataViews and X Windows libraries, as well as the real-time simulator class and the interactive edit display class. This environment must be built on machines with DataViews, X Windows, and Objective-C installed.

**Syntax:** *build* <env-name>

To build a runtime simulation environment on a specified file, type

**build** <a filename>

---

### 4.2 buildsm

**Function:** Builds (creates) a small runtime simulation environment on the specified file. This environment will not include the DataViews libraries, the real-time simulator class, or the interactive edit display class. This environment can be built on any machine with the Objective-C compiler installed. To include any class that the *buildsm* command leaves out, just reference it with the *include* command.

**Syntax:** *buildsm* <env-name>

To build a small runtime simulation environment on a specified file, type

**buildsm** <a filename>

---

### 4.3 **chdir**

Function: Changes the working directory.

Syntax: [ *chdir* / *cd* ] <dir-spec>

To change the current working directory while in the Environment Builder, type

**chdir** <a directory name or path>

or

**cd** <a directory name or path>

---

### 4.4 **comment(#)**

Function: Allows documentation inside the Environment Builder.

Syntax: #

To use the comment indicator, type

# comment

The comment indicator is the first character on each comment line.

---

### 4.5 **describe**

Function: Accesses on-line help for classes.

Syntax: *describe* [ <class> ]+

To get a description of two classes, type

**describe** <a class name> <another class name>

---

## 4.6 exit

Function: Exits from the Environment Builder.

Syntax: [ *exit* / *halt* / *quit* / *stop* ]

To leave the Environment Builder, type

**exit**

The commands *halt*, *quit*, and *stop* will also exit the Environment Builder, and all are equivalent to the *exit* command.

---

## 4.7 flags

Function: Sets the debug or echo flag.

Syntax: [ *debug* / *echo* ] [ *on* / *off* ]

To set the debug flag on, type

**debug on**

To turn the debug flag off, type

**debug off**

Use the same syntax to turn the *echo* flag on or off. These verbose flags are off by default. If the debug flag is on, GOOSE prints out messages informing the user what routine is being executed and the current values of some variables. This flag is used to help the user debug while building his Runtime Environment. If the echo flag is on, each line of the command file is echoed.

---

## 4.8 help

Function: Gets on-line help for one or more commands in the Environment Builder.

Syntax: [ *help* / ? ] { <command> }+

To get the available help, type

**help**

or

**?**

To get help on specific commands, type **help** or **?** followed by the command name, for example

**help include read**

gives help on the commands *include* and *read*.

---

## 4.9 include

Function: Specifies class(es) that have been compiled with the Class Developer to be loaded into the simulation environment.

Syntax: **include** [ <class> ]+

To include or load compiled class(es) into the simulation environment, type

**include** <one or more class names>

---

## 4.10 read

Function: Reads a file containing Environment Builder commands.

Syntax: **read** <command file>

To read a command file into the Environment Builder, type

**read** <a filename>

---

## 4.11 reset

Function: Clears all the previous commands issued in the Environment Builder.

Syntax: **reset**

To clear the Environment Builder, type

**reset**

The Environment Builder needs to be reset every time a new simulation environment is built.

---

## 4.12 shell(!)

Function: Executes a system shell command and returns to the Environment Builder.

Syntax: *! <shell command>*

To execute a UNIX or DOS shell command while in the Environment Builder, precede the shell command string with the character **!**. For example, to get a listing of the current UNIX directory, type

**!ls**

---

## 4.13 show

Function: Displays Environment Builder information.

Syntax: *show { classes / flags }+*

To view the current information on classes and/or system flags in the Environment Builder, use the show command. For example, to see the classes and system flags, type

**show classes flags**

---

## 4.14 showcwd

Function: Displays the current working directory

Syntax: *[ showcwd / pwd ]*

Type

**showcwd**

or

**pwd**

to see the current working directory.

---

## 4.15 showfs

Function: Displays the suffix of the Objective-C definition files.

Syntax: *showfs*

To see the suffix of the Objective-C definition files, type

**showfs**

---

## 5. RUNTIME SIMULATION ENVIRONMENT COMMANDS

The Runtime Simulation Environment is the executable file created with the Environment Builder which includes those classes specified by the user. These classes were previously compiled with the Class Developer. The Runtime Simulation Environment is used to build, test, modify, and execute models. The following commands can be entered on the command line or *read* from a command file. An example of a Runtime Environment command file can be found in Sect. 8.5. Appendix A explains the syntax used in the command descriptions that follow. Note that all the commands are case sensitive; the way the commands appear in the manual (upper or lower case) is the only way that they are recognizable to GOOSE.

---

### 5.1 assign(=)

Function: Assigns a value to a global variable.

Syntax: `<variable> = <value>`

To assign a value to a global variable, use the = sign. For example, to change *dt*, type

```
dt = 2
```

---

### 5.2 call

Function: Calls a macro.

Syntax: `call <name> [<p1> <p2> ... ]`

Macros are commonly defined to prevent repetitive typing of commands or series of commands. See the *DEFINE* command, Sect. 5.9, for information on defining macros. To call a previously defined macro, type

```
call <macro name>
```

If the defined macro has optional parameters, the values to be substituted in the macro for the parameters are included on the command line. For example, typing

```
call macro1 p1 b
```

would invoke the macro *macro1* and substitute *p1* for all the references to the first parameter in the macro and *b* for all the references to the second parameter in the macro.

---

### 5.3 **chdir**

Function: Changes the current working directory.

Syntax: [ *chdir* / *cd* ] <*dir-spec*>

To change the current working directory while in the Runtime Environment, type

**chdir** <a directory name or path>

or

**cd** <a directory name or path>

---

### 5.4 **comment(#)**

Function: Allows documentation inside the Runtime Environment.

Syntax: #

To use the comment indicator, type

# comment

The comment indicator is the first character on each comment line.

---

### 5.5 **connect**

Function: Connects a slot of one object to another object.

Syntax: **connect** <*name1*> <*slot*> = <*name2*>

Assuming *object1* has a slot *other*, connecting *object1* to *object2* requires typing

**connect** object1 other=object2

---

## 5.6 continue

Function: Continues running the paused simulation.

Syntax: *continue*

To continue running a paused simulation, type

```
continue
```

---

## 5.7 create

Function: Creates an object and adds it to the current model.

Syntax: *create* <name> <Class> { <slot> = <dvalue> }+

For example, to create an object *valve1* of the class *Valve* in Fig. 2, type

```
create valve1 Valve
```

Any number of slot values can be initialized on the *create* command line. Since the *Valve* class has a slot *valve\_pos*, *valve\_pos* could be initialized by typing

```
create valve1 Valve valve_pos=.001
```

---

## 5.8 createT1

Function: Creates an interpolation table object and adds it to the current model.

Syntax: *createT1* <name> *Table1* <n> <x1>...<xn> <y1>...<yn>

For example, to create an interpolation table of size 2 x 2 with x values 5 and 6 and y values 3 and 4, type

```
createT1 intab1 Table1 2 5 6 3 4
```

Note that the table must be a square matrix, nxn, where *n* is the dimension of the x and y arrays.

---

## 5.9 DEFINE

Function: Defines a macro.

Syntax: **DEFINE** *<name>* [ *<p1>* *<p2>* ... ]  
*<code>*  
**END**

Macros are commonly defined to prevent repetitive typing of commands or series of commands. For example, the following macro creates a pipe and connects it to a pump.

```
DEFINE mymacro  
create pipe2 Pipe  
connect pipe2 ce=pump1  
END
```

Macro parameters can be any character set separated by blanks. To include parameters in the macro, add them to the first line of the definition as in the following example.

```
DEFINE mymacro $1 P2  
create pipe$1 Pipe  
connect pipe$1 ce=pumpP2  
END
```

To mimic the first example, call the above macro with parameters of 2 and 1. See the *call* command, Sect. 5.2, for more information on calling macros.

---

## 5.10 delete

Function: Deletes one or more objects from the model.

Syntax: **delete** { *<name>* } +

For example, to delete the object *pipe1*, type

```
delete pipe1
```

---

## 5.11 describe

Function: Accesses on-line help for available classes.

Syntax: *describe* { <class> } +

To get a description of two classes, type

```
describe <a class name> <another class name>
```

---

## 5.12 edit

Function: Assigns a value to a slot of an existing object.

Syntax: *edit* <name> { <slot> = <dvalue> }+

For example, to edit the slot *rho* of the object *pipe1*, type

```
edit pipe1 rho = .01
```

More than one slot of the same object can be edited with the same *edit* command.

---

## 5.13 exit

Function: Exits the Runtime Environment.

Syntax: [ *exit* | *halt* | *quit* | *stop* ]

To leave the Runtime Environment, type

```
exit
```

The commands *halt*, *quit*, and *stop* will also exit the Runtime Environment, and all are equivalent to the *exit* command.

---

## 5.14 flags

Function: Sets the debug, echo, or extraderiv flag.

Syntax: [ *debug / echo / extraderiv* ] [ *on / off* ]

To set the debug flag on, type

**debug on**

To turn the debug flag off, type

**debug off**

Use the same syntax to turn the *echo* or *extraderiv* flags on or off. These verbose flags are off by default. If the debug flag is on, GOOSE prints out messages informing the user what routine is being executed and the current values of some variables. This flag is used to help the user debug his Runtime Environment. If the echo flag is on, each line of the command file is echoed. The extraderiv flag is used to force the differential equation solver, lsodar, to recheck the need to resolve the derivative at the current time step.

---

## 5.15 help

Function: Provides on-line help for one or more commands in the Runtime Environment.

Syntax: [ *help / ?* ] { <topic> }+

To get the available help, type

**help**

or

**?**

To get help on a specific topic, type **help** or **?** followed by the topic help is needed on, for example,

**help Pipe**

gives information on the class *Pipe*.

---

## 5.16 hold

Function: Adds slot(s) to the list of slots held (for plots, etc.). The list can also be cleared and displayed with the hold command.

Syntax: **hold** [ *clear* / *list* / { *from* <name> <slot> ... <slotn>}+ ]

To clear the hold list, type

**hold clear**

To view the hold list, type

**hold list**

To add the slot *rho* from the object *pipe1* to the hold list, type

**hold from pipe1 rho**

---

## 5.17 initialize

Function: Initializes objects of the model by performing the INITMETHODs defined for each class.

Syntax: **initialize** [ <name> ]+ / *model*

To initialize the model, type

**initialize model**

To initialize the objects *pipe1* and *pipe2*, type

**initialize pipe1 pipe2**

---

## 5.18 load

Function: Loads a previously *saved* model from a disk file.

Syntax: **load** <model file>

To load a model file, type

**load <model file>**

---

## 5.19 output

Function: Adds slot(s) to the list of slots output at each time step of the simulation. The list can also be cleared and displayed with the output command.

Syntax: **output** [ *clear* / *list* / { *from* <name> <slot> }+ ]

To clear the output list, type

**output clear**

To view the output list, type

**output list**

To add the slots *rho* and *w* from the object *pipe1* to the output list, type

**output from pipe1 rho w**

---

## 5.20 panelobjs

Function: Adds slot(s) to the list of objects that are to be displayed in the interactive edit panel. The list can also be cleared and displayed with the panelobjs command.

Syntax: **panelobjs** [ *clear* / *list* / { <name> }+ ]

To clear the edit panel list, type

**panelobjs clear**

To view the edit panel list, type

**panelobjs list**

To add the objects *pipe1* and *pump* to the edit panel list, type

**panelobjs pipe1 pump**

Note that this command is available only in the SUN version.

---

## 5.21 plot

Function: Specifies one or more slots (or time *t*) to be plotted after the simulation run or while the simulation is paused. The first value is plotted on the horizontal axis. The plot command can also show or set the current value of the plot title.

Syntax: **plot { from <name> <slot> }+ or plot title [ <title> ]**

To plot the slots *rho* and *w* from the object *pipe1*, type

```
plot from pipe1 rho w
```

To set the plot title, type

```
plot title <title name>
```

To plot time *t* and the slot *rho* from the object *pipe1*, type

```
plot from t from pipe1 rho
```

---

## 5.22 range

Function: Adds slot(s) to the range list and specifies the minimum and/or maximum of one or more slots to be plotted. The list can also be cleared and displayed with the range command.

Syntax: **range { [ clear / list / { from <name> <slot> [ min x ] [ max y ] }+ }**

To clear the range list, type

```
range clear
```

To view the range list, type

```
range list
```

To specify the range of the slots *rho* and *w* from the object *pipe1*, type

```
range from pipe1 rho min 0.0001 max 1.0 w min 0.0
```

---

### 5.23 read

Function: Reads a file containing Runtime Environment commands.

Syntax: *read* <command file>

To read a command file into the Runtime Environment, type

**read** <a filename>

---

### 5.24 reset

Function: Clears all the previous commands issued in the Runtime Environment.

Syntax: *reset*

To clear the Runtime Environment, type

**reset**

---

### 5.25 run

Function: Runs the current model from  $t=tstart$  to  $t=tend$  in steps of  $dt$ .

Syntax: *run*

To begin the simulation run, type

**run**

---

### 5.26 save

Function: Saves the current model on a disk file.

Syntax: *save* <model file>

To save the current model to a disk file, type

**save**

---

## 5.27 send

Function: Sends a message to an existing named object.

Syntax: *send* <object-name> <message>

The command *send* is used to execute a defined operation in a class definition. For example, in the class *LinearA*, there is a method *roots* that computes the eigenvalues. Assuming there is a defined object *jac* of the class *LinearA*, to execute the method *roots*, send the object *jac* the message *roots* as follows

```
send jac roots
```

---

## 5.28 shell(!)

Function: Executes a system shell command and returns to the Runtime Environment.

Syntax: *!* <shell command>

To execute a UNIX or DOS shell command while in the Runtime Environment, precede the shell command string with the character *!*. For example, to get a listing of the current UNIX directory, type

```
!ls
```

---

## 5.29 show

Function: Displays Runtime Environment information.

Syntax: *show* { *classes* / *connections* / *derivlist* / *flags* / *dynamiclist* / *initlist* / *digitallist* / *model* / *whenlist* / *definelist* / <obj\_name> / { *from* <name> <slot> ... <slotn> }+ / <variable\_name> / *variables* }+

To see current information in the Runtime Environment, use the *show* command. For example, to see the object *jac* and the derivative list, type

```
show jac derivlist
```

---

### 5.30 showcwd

Function: Displays the current working directory.

Syntax: *[ showcwd / pwd ]*

Type

**showcwd**

or

**pwd**

to see the current working directory.

---

### 5.31 syntax

Function: Shows the syntax of the Runtime Environment commands.

Syntax: *syntax { <command> }+*

For example, to see the syntax of the *show* command, type

**syntax show**

---

### 5.32 validate

Function: Validates object connections by executing the rules defined in the class definition's VALIDATE section.

Syntax: *validate [ <name> ]+ / model*

To validate the model, type

**validate model**

To validate the objects *jac* and *pipe*, type

**validate jac pipe**

---

### 5.33 **waterp** [ <machine-name> ]

Function: Calculates water properties.

Syntax: **waterp**

To calculate the water properties, type

**waterp**

A screen is displayed to allow the user to input pressure, temperature, and enthalpy values for water properties calculations. Water properties for subcooled, saturated, and superheated conditions are available.

The default display is on the machine currently logged into. To display the water properties screen on another machine, include the name of the machine to be used for the display on the **waterp** command line.

---

### 5.34 **writeto**

Function: Writes specified values in the output list to a file.

Syntax: **writeto** [ <data-file> [ *all* / { *from* <name> <slot> ... <slotn>}+ ] ]

To write all of the output values to a file, type

**writeto all**

For example, to write the values of the slots *valve\_pos* and *init\_valve\_pos* (Fig. 2) to the output file *data.dat*, type

**writeto data.dat from valve1 valve\_pos init\_valve\_pos**

If slots are not requested, all data are assumed by default. If a filename is not specified, the default file is *writeto.dat*.

---

## 6. GLOBAL VARIABLES

Global variables are variables that can be changed interactively in the Runtime Simulation Environment with the assignment command (=). Note that all the commands are case sensitive; the way the commands appear in the manual (upper or lower case) is the only way that they are recognizable to GOOSE. Section 8.5 contains many examples on how to use global variables in the Runtime Environment.

Variables denoted by an asterick (\*) are used by the differential equation solver, Isodar.

### 6.1 abstol

\*Purpose: Sets the absolute error tolerance for the differential equation solver. The default is .0001.

### 6.2 derivSrtFlg

Purpose: Whether or not the derivative list is sorted. 1 = sorted, 0 = not sorted.

### 6.3 dynaSrtFlg

Purpose: Whether or not the dynamic list is sorted. 1 = sorted, 0 = not sorted.

### 6.4 circularFlg

Purpose: Whether or not to check for circularity in the sorted lists. 1 = check for circularity, 0 = do not check for circularity. The default is to check for circularity (1).

### 6.5 dt

Purpose: Time step variable.

### 6.6 euler

Purpose: Whether or not to use the Euler method to solve the differential equations. 1 = use the Euler method, 0 = do not use the Euler method (use Isodar). The default is not to use the Euler method (0).

## **6.7 h0**

\*Purpose: Step size to be attempted on the first step of solving the differential equation. The default value is determined by lsodar.

## **6.8 hcur**

\*Purpose: Step size to be attempted in the next step.

## **6.9 hmax**

\*Purpose: Maximum absolute step size allowed. The default value is infinity.

## **6.10 hmin**

\*Purpose: Minimum absolute step size allowed. The default value is 0. This lower bound is not enforced on the final step before reaching time critical (tcrit) when itask = 4 or 5 (see Sect. 6.16 for more information on itask).

## **6.11 hu**

\*Purpose: Last time step size used successfully.

## **6.12 imxer**

\*Purpose: Index of the component of largest magnitude in the weighted local error vector  $[e(i)/ewt(i)]$ , on an error return with istrate = -4 or -5 (see Sect. 6.15 for more information on istrate).

## **6.13 initSrtFlg**

Purpose: Whether or not the list of INITMETHODs is sorted. 1 = sorted, 0 = not sorted.

## **6.14 iopt**

\*Purpose: Whether or not any optional inputs are being used on this call to lsodar. 0 = no optional inputs are being used, default values will be used in all cases; 1 = one or more optional inputs are being used.

## 6.15 istate

\*Purpose: Index used for input and output to specify the state of the Isodar calculation.

On input :

- 1 = This is the first call for the problem (initializations will be done).
- 2 = This is not the first call, and the calculation is to continue normally with no change in any input parameters except possibly *tout* and *itask* (if *itol*, *rtol*, and/or *atol* are changed between calls with *istate* = 2, the new values will be used but not tested for legality).
- 3 = This is not the first call, and the calculation is to continue normally but with a change in input parameters other than *tout* and *itask*.

On output :

- 1 = Nothing was done.
- 2 = The integration was performed successfully and no roots were found.
- 3 = The integration was successful, and one or more roots were found before satisfying the stop condition specified by *itask*.
- 1 = An excessive amount of work (more than *mxstep* steps) was done on this call before completing the requested task, but the integration was otherwise successful as far as *t* (to continue reset *istate* to a value >1 and reset *mxstep* to avoid the error again).
- 2 = Too much accuracy was requested for the precision of the machine being used. This was detected before completing the requested task, but the integration was successful as far as *t* (to continue, reset the tolerance parameters and set *istate* to 3).
- 3 = Illegal input was detected before taking any integration steps.
- 4 = There were repeated error test failures on one attempted step before completing the requested task, but the integration was successful as far as *t* (the problem may have a singularity or the input may be inappropriate).
- 5 = There were repeated convergence test failures on one attempted step before completing the requested task, but the integration was successful as far as *t* (this may be caused by an inaccurate Jacobian matrix, if one is being used).

- 6 = *ewt(i)* became 0 for some *i* during the integration. Pure relative error control [*atol(i)=0.0*] was requested on a variable which has now vanished; the integration was successful as far as *t*.
- 7 = The length of *rwork* and/or *iwork* was too small to proceed, but the integration was successful as far as *t* (this happens when *lsodar* chooses to switch methods but *lrw* and/or *liw* is too small for the new method).

### 6.16 itask

\*Purpose: Index specifying the task to be performed.

- 1 = Normal computation of output values of *y(t)* at *t=tout* (by overshooting and interpolating).
- 2 = Take one step only and return.
- 3 = Stop at the first internal mesh point at or beyond *t=tout* and return.
- 4 = Normal computation of output values of *y(t)* at *t=tout* but without overshooting *t=tcrit*; *tcrit* may be = or beyond *tout* but not behind in the direction of integration (this option is useful if the problem has a singularity at or beyond *t=tcrit*).
- 5 = Take one step without passing *tcrit* and return.

### 6.17 itol

\*Purpose: Indicator for the type of error control

<u>itol</u>	<u>rtol</u>	<u>atol</u>	<u>ewt(i)</u>
1	scalar	scalar	$rtol * abs(y(i)) + atol$
2	scalar	array	$rtol * abs(y(i)) + atol(i)$
3	array	scalar	$rtol(i) * abs(y(i)) + atol$
4	array	array	$rtol(i) * abs(y(i)) + atol(i)$

### 6.18 ixpr

\*Purpose: Flag to generate extra printing at method switches. 0 = no extra print (the default), 1 = print data on each switch.

## 6.19 mcur

\*Purpose: Current method indicator. 1 = Adams (nonstiff), 2 = bdf (stiff). This is the method to be attempted on the next step; thus, it differs from *mused* (Sect. 6.21) only if a method switch has just been made.

## 6.20 mf

\*Purpose: Jacobian type indicator (*jt* in *lsodar*). Specifies how the Jacobian matrix  $df/dy$  will be treated, if and when *lsodar* requires this matrix. 1 = user-supplied full (*neqns* by *neqns*) Jacobian, 2 = an internally generated (difference quotient) full Jacobian, 4 = a user-supplied banded Jacobian, 5 = an internally generated banded Jacobian.

## 6.21 mused

\*Purpose: Method indicator for the last successful step. 1 = Adams (nonstiff), 2 = bdf (stiff).

## 6.22 mxordn

\*Purpose: Maximum order to be allowed for the nonstiff (Adams) method. The default value is 12. If *mxordn* exceeds 12, it will be reduced to 12. *mxordn* is held constant during the problem.

## 6.23 mxords

\*Purpose: Maximum order to be allowed for the stiff (bdf) method. The default value is 5. If *mxords* exceeds 5, it will be reduced to 5. *mxords* is held constant during the problem.

## 6.24 mxstep

\*Purpose: Maximum number of steps (internally defined) allowed during one call to *lsodar*. The default value is 500.

## 6.25 neqns

\*Purpose: Size of the ode system (number of first-order ordinary differential equations). *neqns* may be decreased, but not increased, during the problem.

### **6.26 nfe**

\*Purpose: Number of times the user-supplied function  $f$  has been evaluated for the problem so far.

### **6.27 nhold**

Purpose: Number of elements in the hold list.

### **6.28 nje**

\*Purpose: Number of Jacobian evaluations (and of matrix  $lu$  decompositions) for the problem so far.

### **6.29 noutput**

Purpose: Number of elements in the output list.

### **6.30 nperiods**

Purpose: Number of steps executed in the last simulation run.

### **6.31 nplot**

Purpose: Number of points to plot.

### **6.32 nqcur**

\*Purpose: Order to be attempted on the next step.

### **6.33 nqu**

\*Purpose: Method order last used successfully.

### **6.34 nst**

\*Purpose: Number of steps taken for the problem so far.

### 6.35 orgSort

Purpose: Whether to use the sequential or heap sort. 1 = sequential sort, 0 = heap sort. It depends on how the user defines his inputs and outputs as to which sort is better to use. In the worst-case scenario (reverse order), the heap sort is better to use, but, in general, the user will notice little difference between the sorts. The default is the sequential sort (1).

### 6.36 plotmax

Purpose: Maximum number of points to plot.

### 6.37 ppcpu

Purpose: CPU name on which to display the TempleGraph plot (used only in the SUN version).

### 6.38 reltol

\*Purpose: Set the relative error tolerance for the differential equation solver. The default is .0001.

### 6.39 showop

Purpose: Whether or not to print the simulation run output to the screen. 1 = print output to the screen, 0 = do not print output. The default is to print output to the screen (1).

### 6.40 t

Purpose: Independent variable. On the first call,  $t$  is the initial point of the integration. After each call,  $t$  is the value at which a computed solution  $y$  is evaluated. If a root was found,  $t$  is the computed location of the root reached first. On error,  $t$  is the farthest point reached.

### 6.41 tcur

\*Purpose: Current value of the independent variable which the solver has actually reached, that is the current internal mesh point in  $t$ . If interpolation was done,  $tcur$  may be farther than  $t$ .

## 6.42 tend

Purpose: Time for the simulation run to end.

## 6.43 tolsf

\*Purpose: Tolerance scale factor, >1.0, computed when a request for too much accuracy was detected. If *itol* is left unaltered but *rtol* and *atol* are uniformly scaled up by a factor of *tolsf* for the next call, then the solver is deemed likely to succeed.

## 6.44 trace

Purpose: Whether or not to turn the verbose trace flag on (usually used for debugging). 1 = on, 0 = off. The default is off.

## 6.45 tstart

Purpose: Start time for the simulation run to begin execution.

## 6.46 tsw

\*Purpose: Value of *t* at the time of the last method switch, if any.

## 6.47 t0

Purpose: Time for the simulation run to start execution. The same as *tstart*.

## 7. ADDITIONAL FEATURES FOUND IN GOOSE

### 7.1 DIFFERENTIAL EQUATION SOLVERS

Two differential equation solvers, the complex Livermore solver for ordinary differential equations with root-finding (Isodar) and the simpler Euler method, are available in GOOSE. *Isodar* provides automatic method switching for stiff and nonstiff problems. GOOSE uses *Isodar* by default. The user can choose between *Isodar* and the Euler method by changing the value of the flag, *euler*, in the Runtime Environment. The flag is off by default; to turn it on, type

```
euler=1
```

to go back to *Isodar*, type

```
euler=0
```

### 7.2 DYNAMIC PLOTS

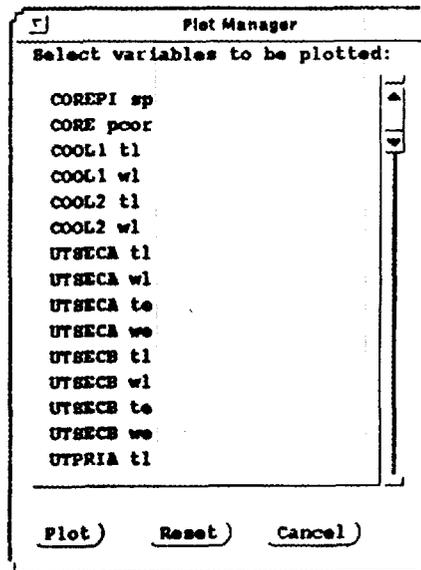
Dynamic plots are available in the SUN version of GOOSE and require the installation of the software packages DataViews and X Windows. To have access to dynamic plots, create the Runtime Environment with the *build* command inside the Environment Builder (Sect. 8.4).

The variables specified in the *output* command of the Runtime Environment are available for dynamic plotting. To create a dynamic plot object, use the *create* command to create an object of the class PlotDV as follows

```
create plot PlotDV pltdsply=mname tplot=300 tscroll=60 vgrid=1 hgrid=1
```

where *mname* is the name of the machine on which the plots are to be displayed. If no machine name is given, the plots will be displayed on the machine the user is logged into. If the plots cannot be displayed, an error message is printed, and the Runtime Environment is exited. The slot *tplot* sets the number of data points to plot in each graph; the default is 100 points. The slot *tscroll* sets the number of points to scroll; the default is no scrolling. The *vgrid* and *hgrid* slots turn the vertical and horizontal grids on (=1) and off (=0), respectively; the default is off.

The dynamic plot object is initialized when the model is initialized (*initialize model*) or the object itself is initialized (*initialize plot* or *send plot init*). While the object is being initialized, a Plot Manager window is displayed (Fig. 8) containing all of the variables in the output list from which the variables to be plotted against time can be chosen. Three buttons are associated with the Plot Manager window; the Plot button plots the selected variables, the Reset button unselects all the choices on the list, and the Cancel button cancels the dynamic plot and deletes the object from the model. The Runtime Environment prompt will not be displayed until either the Plot or Cancel button is pressed.



**Fig. 8. The Plot Manager Window is used for setting up dynamic plots in the SUN version of GOOSE.**

Once the initialization process is complete and the *run* command is executed, the plots will be displayed. If the total number of data points (number of graphs and number of points in each graph) is too large for the screen to accommodate, DataViews errors will be displayed. To resolve this problem, reduce the number of graphs or decrease the number of points on each graph.

Once the plots are active (Fig. 3), three buttons are displayed; the Pause button pauses the simulation, the Copy button prints the displayed plots, and the Return button returns control back to the Plot Manager window.

To prevent variables in the output list and their values from being displayed on the screen after each time step, turn the *showop* flag off by typing

**showop=0**

The output will be displayed again when you type

**showop=1**

### 7.3 REAL-TIME SIMULATOR

To have access to the real-time simulator, create a Runtime Environment with the *build* command inside the Environment Builder (Sect. 8.4). To create a real time simulator object of the class RealTime, use the *create* command as shown below

```
create RealTime RT scale=1
```

where *scale* is the slot that determines the speed of the simulation. When *scale* is equal to 1, the simulator runs in real time (this assumes that time is in units of seconds and the machine the simulation is running on is fast enough). When *scale* is set to 0, the speed of the simulation is maximum, determined by the machine it is running on. The slot *scale* can be set to any number between and including 0.0 and 1.0 and can be changed with the *edit* command in the Runtime Environment.

The real-time simulator is initialized as mentioned in Sect. 7.2.

### 7.4 EIGENVALUE CALCULATIONS

Jacobian matrix and eigenvalue calculations are available in GOOSE. These linear computations are made available when the Runtime Environment is created. The eigenvalues are computed using the public domain subroutine *rg*. This subroutine calls the recommended sequence of subroutines from the eigensystem subroutine package (*eispack*) to find the eigenvalues and eigenvectors (if desired) of a real general matrix.

To have access to the Jacobian matrix and eigenvalue solver, an object of the class LinearA has to be created by typing

```
create LinearA jac
```

The LinearA class has two methods; the Jacobian method computes the Jacobian matrix for the model equations, whereas the roots method computes the Jacobian matrix and the model eigenvalues. To activate these methods, type

```
send jac jacobian
```

or

```
send jac roots
```

GOOSE prints the eigenvalues to the screen and to a file called *roots.dat*. The Jacobian matrix is printed to the file *jac.dat*. If either of these files exist, GOOSE informs the user of their existence and asks if they are to be overwritten. If the files are not overwritten, the computation is canceled.

The class LinearA contains the slot *eps<sub>mch</sub>*, which defines the precision of the machine being used. The default value of *eps<sub>mch</sub>* is 1.E-7 and can be changed when the *jac* object is created or with the *edit* command in the Runtime Environment.

## 7.5 INTERACTIVE EDITING

Interactive editing is available in the SUN version of GOOSE and requires the installation of X Windows. To have access to interactive editing, create a Runtime Environment with the *build* command inside the Environment Builder (Sect. 8.4).

To create an edit panel object, use the *create* command to create an object of the class DspPanel, as follows:

```
create dpanel DspPanel fdsply=mymachine flabel="Demo Label"  
panelobjs object1 object2 object3
```

where *mymachine* is the name of the machine on which the edit panel will be displayed. If no machine name is given, the edit panel will be displayed on the machine the user is logged into. If the panel cannot be displayed, an error message is printed and the Runtime Environment is exited. The slot *flabel* is displayed at the top of the interactive edit panel.

Figure 9 presents an interactive edit panel. This panel is initialized as mentioned in Sect. 7.2. *panelobjs* is a list of objects to be displayed on the interactive edit panel. All the slots of these objects with read and write access are displayed in the panel. During initialization, GOOSE waits for the user to press the Resume button on the edit panel before continuing. At this time the user can enter and modify initial slot values before pressing the Resume button. Once the simulation is running, the user can press the Pause button at any time to pause the simulation and edit any slot value displayed on the panel. The Quit button kills the interactive edit panel and deletes the object, *dpanel*, from the model.

## 7.6 VECTORS

In this section a description of how to access and manipulate vectors in GOOSE is presented.

When defining a vector slot in the class definition (for more information on creating slots, see Sect. 3.4), there is an extra field that contains the dimension of the vector. An example slot definition section that contains vectors is as follows:

<b>watt0</b>	<b>real</b>	<b>rw</b>	<b>"Core Power Initial"</b>	<b>W</b>
<b>c</b>	<b>vector 7</b>	<b>rw</b>	<b>"Normalized Precursor"</b>	<b>nondim</b>
<b>c0</b>	<b>vector 7</b>	<b>rw</b>	<b>"Normalized Precursor Initial"</b>	<b>nondim</b>
<b>dc</b>	<b>vector 7</b>	<b>rw</b>	<b>"Normalized Precursor Deriv"</b>	<b>1/s</b>

Currently, the only available vector data type is real. GOOSE requires that vectors involved in the same integration have the same dimension. For example,

```
INTEGRATE  
c      dc      c0
```

*c*, *dc*, and *c0*, defined in the slot definition section above, each have seven elements, and each of the seven elements will be integrated.



GOOSE provides several functions for manipulating vectors. These functions can be accessed in method definitions. For example, to get the current value of a vector element, use the function *getV()*.

```
watt0=getV(c,2);
```

This assigns the value of the third element of the *c* vector to the slot *watt0*. Note vector indices start at 0, not 1.

To change the value of a vector element, use the *editV()* function.

```
editV(c,0,.1);
```

This statement assigns .1 to the first element of the *c* vector.

GOOSE also provides users the ability to access the address of vector elements. For example,

```
aptr=[dc getA:5];
```

where *aptr* is a pointer to a real number (declared as *real \*aptr*) and is assigned the address of the sixth element of the *dc* vector.

## 8. GOOSE SAMPLE PROGRAM

### 8.1 THE PROBLEM

In this section, an illustration is presented that uses GOOSE to produce a simple model of a Westinghouse pressurized water reactor (PWR). This model consists of the point kinetics equations with an averaged delayed neutron group. Reactivity changes in this model are due to external input from control rod motion and to internal effects caused by changes in the temperature of both the fuel and the moderator/coolant. The primary system is represented by state equations describing the dynamics of the fuel and moderator temperatures. The fuel and the moderator are each modeled in three nodes. The heat capacity of the clad is neglected, and an average heat transfer coefficient between the fuel and moderator is used. This model does not include the pressurizer, steam generator, or primary pumps effects.

The GOOSE model consists of five classes:

1. **ReactCore** represents the point kinetics equations.
2. **ReactFeedBack** represents the reactivity feedback due to changes in the fuel and moderator temperatures.
3. **ReactMode** represents one node for the fuel and one node for the moderator temperatures.
4. **HPRTW** represents the moderator (coolant) inlet parameters.
5. **ReactPi** represents a proportional-integral (PI) controller.

A diagram of the PWR model explained above can be found in Sect. 2.1. As explained in Sect. 2.1, each box in the figure represents an object in the model and belongs to one of the five classes described above. Three objects of the **ReactMode** class represent the fuel and moderator nodes. The arrows represent the connections between the objects. The objects in the diagram are created and connected in the Runtime Environment.

### 8.2 THE CLASS DEFINITION

The following is the class definition file for the class **ReactCore**.

---

#### HEADER

```
// ---- ReactCore ACP LIBRARY -----  
// ---- M. A. Abdalla, L. Guimaraes, D.J. Nypaver --  
// ---- ORNL P.O.Box 2008 MS 6010 Oak Ridge TN 37831-6010 -----
```

#### END

#### DESCRIBE

```
Reactor Core Model. Point Kinetics with one Delay Group.  
END
```

## CLASS ReactCore

### SLOTS

beta1	real	rw	"Delay Neutron fract 1"	nondim
beta2	real	rw	"Delay Neutron fract 2"	nondim
beta3	real	rw	"Delay Neutron fract 3"	nondim
beta4	real	rw	"Delay Neutron fract 4"	nondim
beta5	real	rw	"Delay Neutron fract 5"	nondim
beta6	real	rw	"Delay Neutron fract 6"	nondim
betat	real	rw	"Delay Neutron fract T"	nondim
lamda1	real	rw	"Delay Neutron Decay 1"	1/s
lamda2	real	rw	"Delay Neutron Decay 2"	1/s
lamda3	real	rw	"Delay Neutron Decay 3"	1/s
lamda4	real	rw	"Delay Neutron Decay 4"	1/s
lamda5	real	rw	"Delay Neutron Decay 5"	1/s
lamda6	real	rw	"Delay Neutron Decay 6"	1/s
lamda	real	rw	"Delay Neutron Decay T"	1/s
ngt	real	rw	"Neutron Generation Time"	sec
pcor	real	rw	"Normalized Power"	nondim
ppcor	real	rw	"Percentage of Power"	nondim
pcor0	real	rw	"Pcor Initial"	nondim
dpcor	real	rw	"Pcor Deriv"	1/s
watt	real	rw	"Core Power"	W
watt0	real	rw	"Core Power Initial"	W
c	real	rw	"Normalized Precursor"	nondim
c0	real	rw	"Normalized Precursor Initial"	nondim
dc	real	rw	"Normalized Precursor Deriv"	1/s
urho	object	rw		
urod	object	rw		
<b>EXTERNAL</b>				
rod	real	rw	"Rod Reactivity"	nondim
rho	real	rw	"Reactivity"	nondim
<b>END</b>				

### INITMETHOD init

```
beta1 = .000209,beta2 = .001414,beta3 = .001309;  
beta4 = .002727,beta5 = .000925,beta6 =.000314;  
betat = beta1+beta2+beta3+beta4+beta5+beta6;  
lamda1 = .0125,lamda2 = .0308,lamda3 = .114;  
lamda4 = .307 ,lamda5 = 1.19 ,lamda6 = 3.19;  
lamda = betat/(beta1/lamda1 + beta2/lamda2 + beta3/lamda3  
+ beta4/lamda4 + beta5/lamda5 + beta6/lamda6);  
ngt = 17.9e-6;  
pcor0=1.,watt0=3436.e6;  
c0 = betat*pcor0/(ngt*lamda);
```

### END

### VALIDATE

```
urho respondsto rho  
urod respondsto rod
```

### END

```
DERIVMETHOD deriv1
```

```
    watt = watt0*pcor;
```

```
OUTPUTS
```

```
    watt
```

```
END
```

```
DERIVMETHOD deriv2
```

```
    dpcor = ([urho rho] + [urod rod] - betat)*pcor/ngt + lamda*c ;
```

```
    dc = betat*pcor/ngt - lamda*c ;
```

```
INTEGRATE
```

```
    pcor      dpcor      pcor0
```

```
    c dc      c0
```

```
INPUTS
```

```
    urho.rho urod.rod
```

```
END
```

```
DYNAMETHOD dynal
```

```
    ppcor=pcor*100.;
```

```
END
```

```
DIGIMETHOD digi1
```

```
    printf("executing digimethod\n");
```

```
SAMPLING
```

```
    s1
```

```
DTIME
```

```
    dtime1
```

```
END
```

```
save
```

---

The optional **HEADER** section consists of text to be included at the beginning of the Objective-C source files created by the Class Developer.

The optional **DESCRIBE** section consists of one or more lines of text describing the class being defined. This text is included in the on-line help file automatically generated for the class by the Class Developer.

The **CLASS** statement identifies the name of the class being defined.

The structure of a class is established through slot definitions found in the **SLOTS** section. Each slot definition includes a slot or variable name, a data type specification (integer, real, string or object), user accessibility ("rw" = read and write, "r" = read, "w" = write, "\*" = none), and two optional fields: a prompt or description field and a units specification field for use in the runtime user interface. The optional **EXTERNAL** subsection declares the data types of slots to be referenced in other objects not (necessarily) in the class being defined. The optional **INTERNAL** subsection provides the user with

the ability to write Objective-C code that declares variables, includes statements, etc., referenced by only the class being defined.

The **INITMETHOD** section defines the source code for initializing objects of the current class. In between the section header and **END** is Objective-C code. More than one **INITMETHOD** can be defined. Optional **INPUTS** and **OUTPUTS** sections for the initial method can also be specified. The **INPUTS** and **OUTPUTS** are used by the GOOSE Runtime Environment to determine the order of execution of the list of **INITMETHODS** for all the objects in a model.

The optional **VALIDATE** section specifies a list of validation rules that apply to objects of the defined class during runtime. The validation rules ensure that objects in the model are interconnected properly. The *subclass* rule specifies that the named object must be connected to another object that belongs to the specified subclass. Other available validation rules include *nomil*, which means that object must be connected to something; *class*, which indicates that the connected object must be a member of the named class; and *respondsto*, which specifies that the connected object must be able to respond to or return a value for the given slot or method. There can be as many validation rules as necessary for a class.

The **DERIVMETHOD** section defines the source code for derivatives of the state variables. The GOOSE Runtime Environment will execute the derivative method section whenever the time derivatives of the components of a model are required. The optional **INPUTS** subsection specifies a list of object slots used as inputs by the method being defined. A list of **OUTPUTS** may also be similarly specified. Any derivative method defined in a model that has slots that are **OUTPUTS** will be executed before the derivative methods in the model that use those slots as **INPUTS**. If GOOSE cannot sort the inputs and outputs so that the inputs are defined before they are referenced by the outputs, GOOSE informs the user that a circularity error exists. The optional **INTEGRATE** subsection provides information for the differential equation solver. It requires the slots that the differential equation solver will integrate, the slots that represent the time derivative, and the slots (or constants) that contain the initial values.

The **DYNAMETHOD** section includes statements that are to be executed at each communication interval and can have optional **INPUTS** and **OUTPUTS** sections. This method is commonly used for interfacing with dynamic plots and graphical user interfaces.

The **DIGIMETHOD** section includes statements that are to be executed at each specified sampling interval and can have optional **INPUTS** and **OUTPUTS** sections. The **DIGIMETHOD** has a required **SAMPLING** and **DTIME** section. The code defined in the **DIGIMETHOD** will be executed at each time interval that the sampling variable is set to. The **DTIME** variable is used by the method to keep track of the last sampling time. A digital method could be used as a communication device that needs responses at a given sampling time, not necessarily each computation step. This method can be used for interprocess communication, scheduling, and interrupting the simulation for other communication needs.

### 8.3 THE CLASS DEVELOPER

To invoke the Class Developer, **cldev9** is entered at the monitor prompt. The *read* command directs the Class Developer to read commands from the file *ReactCore.cd*, shown above. The *save* command saves the class in Objective-C and GOOSE files. The *compile* command compiles the class so that it can be built into a simulation environment. The *exit* command leaves the Class Developer. Following are the results shown on the computer monitor (boldface type indicates user input) of the Class Developer creating the class **ReactCore** using the commands discussed above.

```
[icacp19 pwr] cldev9
```

```
(Type 'help' for help)
```

```
<cldev9> read ReactCore.cd
```

```
Class ReactCore saved (.h, .m, .1 and .def files)
```

```
<cldev9> compile
```

```
objcc -DSUN -postLink -I/usr6/users/ACP/simul/ver1.4 -I${DVHOME}/include  
-I/usr/openwin/include -g -c ReactCore.m
```

```
Objective-C Version 4.3
```

```
Copyright 1988,1989,1990 The Stepstone Corporation. All rights reserved.
```

```
<cldev9> exit
```

### 8.4 THE ENVIRONMENT BUILDER

The Environment Builder is used to generate an executable Runtime Environment. To use the Environment Builder, type **bldenv2** at the monitor prompt, as shown in the following example. The *include* command includes the user-specified classes needed for the PWR model. The *build* command builds the simulation environment with all the classes specified with the *include* command. The *exit* command leaves the Environment Builder. Below are the results of invoking the Environment Builder at the monitor prompt (boldface type indicates user input) and issuing the commands mentioned above which build the Runtime Environment, **pwr**, containing the included classes.

```
[icacp19 pwr] bldenv2
```

```
(Type 'help' for help)
```

```
<bldenv2> include HPRTW ReactFeedBack ReactPi ReactCore ReactMode
```

```
<bldenv2> build pwr
```

```
Building GOOSE Environment on file 'pwr' ...
```

```
/usr6/users/ACP/simul/ver1.4/buildenv -o pwr HPRTW.o ReactFeedBack.o ReactPi.o  
ReactCore.o ReactMode.o
```

```
The attempt to build the GOOSE Environment 'pwr' is complete.
```

```
<bldenv2> exit
```

## 8.5 THE RUNTIME ENVIRONMENT

Following is the command file, *pwr.cmd*, which will be *read* into the Runtime Environment.

---

```
echo on  
  
# model of the Westinghouse Pressurized Water Reactor  
#  
create core      ReactCore  
create pirod     ReactPi  
create fbcore    ReactFeedBack  
create m1        ReactMode   tf0= 809.31 tm0= 286.35 tmout0= 291.73 root=3.e9  
create m2        ReactMode   tf0= 819.09 tm0= 296.62 tmout0= 301.52 root=2.6e9  
create m3        ReactMode   tf0= 828.87 tm0= 306.39 tmout0= 311.39 root=2.2e9  
create lpln      HPRTW      tl= 281.94 wl= 19852.1 rl=761.57 hl=1241.5e3 pl=15.51  
  
connect core     urho=fbcore  
connect core     urod=pirod  
connect pirod    core=core  
connect fbcore   mod1=m1  
connect fbcore   mod2=m2  
connect fbcore   mod3=m3  
connect m1       uwatt=core  
connect m1       ce=lpln  
connect m2       uwatt=core  
connect m2       ce=m1  
connect m3       uwatt=core  
connect m3       ce=m2  
  
initialize model
```

```
edit pirod kc=.01 i0=0. spoint=.9
edit core s1=.5
tstart = 0.
tend = 300.
dt = 1.
```

```
output from core ppcor
hold from core ppcor
show core
```

---

The *create* command creates new objects to be included in the model. The *connect* command establishes a one-way connection between two objects. This command enables one object to be “aware” of another object.

The *initialize model* command causes all the **INITMETHOD**s for all the objects in the model to be executed and sets the state variables to their initial values. Note that if validation rules have been defined for any of the classes of any of the objects in the model, the validation rules are automatically checked before any initialization takes place. If any validation rule fails, an error message is produced, and the initialization process is terminated. Individual initial methods can be executed by sending an object its initialize message, which is the name of its **INITMETHOD**. For example, *send core init* would send the object *core* the message *init* and cause the **INITMETHOD** to be executed. One could also type *initialize core*. More than one object can be initialized with the *initialize* command.

The *edit* command modifies the values of the named slots. The *output* command establishes a list of values that can be displayed after each time step. Similarly, the command *hold* establishes a list of values to be saved or held after each time step. Values are plotted with the *plot* command, but they cannot be plotted if they have not been put in the hold list with the *hold* command. The global variables, *tstart*, *tend*, and *dt* set the values of the starting time, the ending time, and the computation interval. The *show* command displays the values of the requested slots. The *run* command initiates the simulation run. Then for each time step, the underlying system of differential equations is integrated, and values of the slots specified with the *output* command are displayed. Also, values of slots to be held are stored in the hold list.

Turning *echo on* tells the Runtime Environment to echo the command file (the numbered lines being printed). This verbose flag is off by default.

In the simulation environment, **pwr** commands are *read* from the file *pwr.cmd*. The *run* command starts the simulation run. Following are the GOOSE printouts when *demoenv* is invoked at the monitor prompt and the command file is *read*.

[icacp19 pwr] pwr

(Type 'help' for help)

<pwr> read pwr.cmd

Echo is On.

```
2.
3. # model of the Westinghouse Pressurized Water Reactor
4. #
5. create core ReactCore
6. create pirod ReactPi
7. create fbcore ReactFeedBack
8. create m1 ReactMode tf0= 809.31 tm0= 286.35 tmout0= 291.73 root
9. create m2 ReactMode tf0= 819.09 tm0= 296.62 tmout0= 301.52 root=2.
10. create m3 ReactMode tf0= 828.87 tm0= 306.39 tmout0= 311.39 root=2.
11. create lpln HPRTW tl= 281.94 wl= 19852.1 rl=761.57 hl=1241.5e3
12.
13. connect core urho=fbcore
14. connect core urod=pirod
15. connect pirod core=core
16. connect fbcore mod1=m1
17. connect fbcore mod2=m2
18. connect fbcore mod3=m3
19. connect m1 uwatt=core
20. connect m1 ce=lpln
21. connect m2 uwatt=core
22. connect m2 ce=m1
23. connect m3 uwatt=core
24. connect m3 ce=m2
25.
26. initialize model
```

Validating ...

Done

Sorting ...

...

Done.

Initializing ...

Done.

```
27.
28. edit pirod kc=.01 i0=0. spoint=.9
29. edit core s1=.5
30. tstart = 0.
31. tend = 300.
32. dt = 1.
33.
34. output from core ppcor
```

- 35. hold from core ppcor
- 36. show core

\* core

core: CLASS = ReactCore	beta1 = 0.000209	beta2 = 0.001414
beta3 = 0.001309	beta4 = 0.002727	beta5 = 0.000925
beta6 = 0.000314	betat = 0.006898	lamda1 = 0.0125
lamda2 = 0.0308	lamda3 = 0.114	lamda4 = 0.307
lamda5 = 1.19	lamda6 = 3.19	lamda = 0.0822463
ngt = 1.79e-05	pcor = 0	pcor0 = 1
ppcor = 0	dpcor = 0	watt = 0
watt0 = 3.436e+09	c = 0	c0 = 4685.48
dc = 0	urho = fbcore	urod = pirod
rod = 0	rho = 0	s1 = 0.5
dtime1 = 0		

<pwr> run

Sorting ...

..

Done.

Sorting ...

..

Done.

Press <RETURN> to pause ...

executing digimethod

1. T: 0 CORE ppcor: 100

executing digimethod

executing digimethod

2. T: 1 CORE ppcor: 92.77

executing digimethod

executing digimethod

.....

300. T: 299 CORE ppcor: 90.00

executing digimethod

executing digimethod

301. T: 300 CORE ppcor: 90.00

```
<pwr> plot from t from core ppcor  
plotTG: Attempting TempleGraph connection ...  
plotTG: Sending new data ...  
plotTG: 1 of 1 curves successfully sent.  
plotTG: Defining axis labels and title ...  
plotTG: Resetting autoupdate ...  
plotTG: TempleGraph connection closed.  
  
<pwr> exit
```

## REFERENCES

1. D. J. Nypaver, C. E. Ford, C. March-Leuba, M. A. Abdalla, and L. Guimaraes, "GOOSE Version 1.4, A Powerful Object-Oriented Simulation Environment for Developing Reactor Models," pp. 12.01–12.11 in *8th Power Plant Dynamics, Control & Testing Symposium*, May 27–29, 1992, Vol. I, Knoxville, TN.
2. C. E. Ford, C. March-Leuba, L. Guimaraes, and D. Ugolini, "GOOSE, A Generalized Object-Oriented Simulation Environment for Developing and Testing Reactor Models and Control Strategies," pp. 694–703 in *Proceedings of AI91*, Sept. 15–18, 1991, Vol. II, Jackson Lake, WY.
3. *Objective-C Compiler Version 4.3 User Reference Manual*, The Stepstone Corporation, Sandy Hook, CT, 1990.
4. B. J. Cox, *Object Oriented Programming, An Evolutionary Approach*, Addison-Wesley Publishing Company, Inc., MA, 1990.

## APPENDIX A. COMMAND SYNTAX

This appendix describes the various options associated with each command explained in Chaps. 3 to 5 (*Class Developer Commands*, *Environment Builder Commands*, and *Runtime Simulation Environment Commands*).

1. **CLASS**      The bold-faced words are GOOSE keywords required in the command syntax.
2. |              This logical “or” symbol represents an optional command or command argument choices.
3. <>            The words between arrow head symbols represent required syntax for a command.
4. []             The square brackets represent optional command arguments. If these brackets are embedded, it means that any combination of these options is valid.
5. ...            Three sequential dots represent continuation of the same option either on the same line or next line, according to the location of the dots.
6. {}            The braces represent a grouping mechanism for commands or arguments. They associate the group with the preceding or following option.
7. +             The plus symbol means that any number of the preceding options can be specified.

## APPENDIX B. ERROR MESSAGES

This appendix lists the error messages generated by the GOOSE tools.

1. **ERROR: Argument mismatch. '<macro name>' needs # arguments.**

where # is the number of arguments the macro requires. A call was made to a macro that had an incorrect argument list.

2. **ERROR: Attempted connection not valid.**

This error occurs if a connection is made that results in a validation error. The error is followed by a description of the validation rule that was violated.

3. **ERROR: Cannot create file <filename>.**

Failure in creating the file, <filename>.

4. **ERROR: Cannot create new <object name>.**

Unable to create the requested object.

5. **ERROR: CanNOT create new Table1 object.**

Unable to create the requested interpolation table.

6. **ERROR: Cannot open file <filename>. writeto Request canceled.**

The *writeto* command was unable to open a data file.

7. **ERROR: Class <class name> has no <slot name> message.**

This error occurs if an object is being created and an attempt is made to assign a value to a slot that is not defined for the given object.

8. **ERROR: Class definition not saved yet.**

An attempt was made to compile a class definition in the Class Developer without first saving the definition.

9. **ERROR: CmdLoop encountered EndOfFile.**

An end-of-file error was encountered while reading a command file.

10. **ERROR: CmdLoop input error #.**

where # is followed by the error number returned from the C function *feof()*. This error occurs if there is a problem reading a command file.

11. **ERROR: create syntax ... <syntax used>.**

A syntax error occurred when creating an object. This error is followed by the invalid syntax. Usually a problem with a misplaced equal sign.

12. **ERROR: create table SYNTAX; <syntax used>.**

A syntax error occurred when creating an interpolation table. This error is followed by the invalid syntax. Usually a problem with a misplaced equal sign.

13. **ERROR: dt (=#) must be greater than 0.**

where # is the current value of *dt*. The simulation cannot run because the simulation communication time interval, *dt*, is less than or equal to 0.

14. **ERROR: edit syntax ... <syntax used>.**

A syntax error occurred when editing an object. This error is followed by the invalid syntax. Usually a problem with a misplaced equal sign.

15. **ERROR: Failed to read Class Definition file.**

The *describe* command was unable to find the files needed to perform its task.

16. **ERROR: Hold list is EMPTY. Specify 'hold' BEFORE 'run' and 'writeto'.**

There are two reasons for this error to occur. An attempt is made to *run* the simulation without specifying anything to be held in the hold list, or the command *writeto* is used with an empty hold list.

**17. ERROR: Invalid argument for the 'show' command.**

An invalid argument was given with the *show* command inside the Environment Builder.

**18. ERROR: Isodar returned istate=#.**

where # is the value *istate* returns. The differential equation solver, Isodar, has an error. See Chap. 6, *Global Variables*, for an explanation of *istate* values.

**19. ERROR: Macro not defined.**

A call was made to an undefined macro.

**20. ERROR: Model is EMPTY.**

The *save* command is used on a nil or empty model.

**21. ERROR: Model NOT successfully set up.**

The setup procedure for the differential equation solver failed.

**22. ERROR: Model NOT validated.**

This error occurs during initialization if validation rules failed. The validation rules that were violated are printed out.

**23. ERROR: <name > is not numeric.**

An argument was given to the *range* command that is not numeric.

**24. ERROR: No hold list variables specified. writeto Request canceled.**

No objects were specified in the *writeto* command, so the request is canceled.

**25. ERROR: No object named.**

An object was not specified.

26. **ERROR: No such class as <class name>.**  
Unable to create the requested object because the class name was invalid.
27. **ERROR: No WRITE permission for file <filename>.**  
The *writeto* command does not have permission to create its data file.
28. **ERROR: <object name> not in hold list.**  
The object requested in the *writeto* command is not in the hold list.
29. **ERROR: Object <object name> already exists.**  
An attempt was made to create an object that already exists.
30. **ERROR: Object <object name> has no <method name> method.**  
This error occurs if an action is attempted on a method that is not defined for the given object.
31. **ERROR: Object <object name> is not defined.**  
Specified object is undefined.
32. **ERROR: Object <object name> is not recognized in the Hold List.**  
This error occurs if an attempt is made to plot a variable, with the *plot* command, that is not in the hold list.
33. **ERROR: PLOT request canceled. No Plot Function loaded.**  
This error occurs if an attempt is made to use the nondynamic plotting package, but the software needed to perform the plot was not linked when the model was built.
34. **ERROR: Slot <slot name> must belong to class <class name>.**  
The *class* validation rule was violated.
35. **ERROR: Slot <slot name> must belong to subclass <class name>.**  
The *subclass* validation rule was violated.

36. **ERROR: Slot <slot name> must not be nil.**  
The *not nil* validation rule was violated.
37. **ERROR: Slot <slot name> must obey <validation rule> Rule.**  
The specified validation rule was violated.
38. **ERROR: Slot <slot name> must respond to <message name> message.**  
The *respondsto* validation rule was violated.
39. **ERROR: *tstart* (= #) must be less than *tend* (= #).**  
where # is the current value of *tstart* and *tend*, respectively. The simulation cannot run because the simulation start time, *tstart*, is greater than the simulation end time, *tend*.
40. **ERROR: <variable name> not found.**  
An attempt was made to assign (=) a value to an unknown global variable.
41. **\*\*\*\*\* Syntax Error \*\*\*\*\***  
An invalid command syntax was entered on the command line or inside a command file. If a command file is being used, turn the flag *debug on* to pinpoint the error.
42. **SYS\_ERR #: <system error message>.**  
where # is followed by a system error number and then by the error message. This error occurs if there is a problem in changing directories. The current directory is printed following the error message.
43. **WARNING: derivmethod with no code.**  
This warning is printed if a DERIVMETHOD is defined without any code.
44. **WARNING: Invalid integrate data ignored.**  
This warning is printed when an invalid INTEGRATE statement is given inside a DERIVMETHOD.

## APPENDIX C. GLOSSARY

This appendix defines GOOSE terms used throughout this manual.

1. **circularity**

Methods are sorted so that if the outputs of one method are needed as inputs for another, the method doing the outputting is evaluated first. If inputs and outputs are defined in such a way that the methods cannot be sorted, a circularity error occurs.

2. **class**

A general description of characteristics of a prototype to be used for creating objects. These objects have the same structure and behavior but can have different uses and/or values in the model.

3. **class definition**

A description of the structure and behavior of an object, through slot and method definitions, respectively.

4. **definelist**

A list of the macros defined in the current Runtime Environment.

5. **derivlist**

A list of the derivative methods defined in the current Runtime Environment.

6. **DERIVMETHOD**

The section of the class definition in which the derivatives are defined.

7. **DIGIMETHOD**

The section of the class definition in which code is defined to be executed at the specified time sampling.

8. **digitallist**

A list of the digital methods defined in the current Runtime Environment.

9. **DTIME**

A variable needed by the system to perform digital methods. It contains the latest sampling time.

10. **DYNAMETHOD**

The section of the class definition in which code is defined to be executed each time step.

11. **dynamiclist**

A list of the dynamic methods defined in the current Runtime Environment.

12. **inherit**

The process of a subclass acquiring the same structure and behavior of its parent class (superclass).

13. **initlist**

A list of the initial methods defined in the current Runtime Environment.

14. **INITMETHOD**

A section of the class definition in which code is defined to be executed when objects of the associated class are initialized.

15. **INPUTS**

The slots or variables needed as input in the method being defined.

16. **list**

A record of entries of a specified type in the current Runtime Environment.

17. **macro**

User-defined set of commands that are called with or without parameters. Macros are commonly defined to prevent repetitive typing of commands or series of commands.

18. **message**

A character string sent to an object to execute a method. The message is the name of the method.

19. **method**

The section of the class definition in which code is defined to be executed upon request or at certain times during the simulation.

20. **model**

The complete set of objects created and connected in the Runtime Environment for the simulation.

21. **object**  
An image of the object's affiliated class.
22. **OUTPUTS**  
The slots or variables output by a method for use by other methods.
23. **respondsto**  
A validation rule that verifies that a connecting object contains (or respondsto) the specified slot in its definition.
24. **slotlist**  
A list of the slots defined in the current Runtime Environment.
25. **SAMPLING**  
A variable specifying the sampling time in digital methods.
26. **slots**  
The components or variables in a class definition.
27. **subclass**  
A child class of its parent class. A subclass inherits the behavior and structure of its parent class (superclass). Usually occurs used when a class has the characteristics of its parent class, plus more.
28. **validate**  
A procedure that verifies the connections made in the Runtime Environment.
29. **WHENDO**  
The section of the class definition in which code is defined to be executed when the roots of a constraint equation are found.
30. **whenslist**  
A list of the when do methods defined in the current Runtime Environment.

# INDEX

- Objective-C 13, 15-16
- ! 22, 31, 43
- # 12, 28, 34, 65, 67
- = 13, 33, 35, 37, 42, 46, 54-57, 59, 61-62, 65-68, 73-74, 76
- ? 19, 29-30, 38
- [] 3, 59, 62
- abstol 46
- Adams 50
- assign 33, 37, 46, 59, 72, 76
- atol 48-49, 53
- bdf 50
- bldenv2 64-65
- build 4, 7-8, 27, 54, 56-57, 64-65
- buildsm 4, 7, 27
- call 33, 36, 72, 74, 78
- cd 11, 28, 34
- chdir 11, 28, 34
- circularFlg 13, 46
- circularity 13, 46, 63, 77
- CLASS 11, 61-62, 68
- class definition 3-4, 10-12, 14, 16, 19, 21, 24, 26, 43-44, 57, 60, 72-73, 77-79
- Class Developer 1, 3-4, 10-11, 27, 30, 33, 62, 64, 71-72
- class 1, 3-4, 7, 9-11, 14-15, 19-28, 30-31, 33, 35, 37-39, 43, 54, 56-57, 60, 62-64, 66, 72, 75, 77-79
- cldev9 64
- clear 39-41
- command file 7, 10-11, 19, 21, 27, 29-30, 33, 38, 42, 65-66, 73
- command line 1, 3, 7, 11, 18, 27, 33, 35, 45, 76
- comments 12, 19, 24, 28, 34
- compile 1, 3-4, 10-12, 27, 30, 33, 64, 72
- connection 1, 9, 43-44, 60, 66, 72, 79
- connect 3, 7, 10, 25, 34, 36, 60, 63, 65-67, 78-79
- continue 35
- create 1, 3-4, 7, 9-11, 35-36, 54, 56-57, 60, 65-67, 72-73, 75, 78
- createT1 35, 72-73
- data type 24, 57, 62
- DataViews 4, 27, 54-55
- debug 18-19, 29, 38, 76
- DEFINE 33, 36
- definelist 43, 77
- delete 3-4, 7, 36, 54, 57
- derivlist 23, 43, 77
- DERIVMETHOD 12-13, 62-63, 76-77
- derivSrtFlg 46
- DESCRIBE 14, 60, 62
- describe 23, 28, 37, 73
- differential equation solver 1, 13, 25, 38, 46, 52-54, 63, 74
- DIGIMETHOD 14-15, 62-63, 77
- digitallist 23, 43, 77
- DOSXTNDR 8
- DO 25-26
- DTIME 14-15, 62-63, 77
- dt 33, 42, 46, 66-67, 73
- DYNAMETHOD 16, 62-63, 78
- dynamiclist 23, 43, 78
- dynaSrtFlg 46
- echo 11, 18-19, 29, 38, 65-67
- edit 3, 7, 15, 18, 37, 40, 56-57, 59, 66-67, 73
- eigenvalue 1, 7, 43, 56
- EMACS 18
- END 12-16, 19-22, 24-26, 36, 60-63
- Environment Builder 1, 3-4, 8, 10, 27, 33, 54, 56-57, 64, 71, 74
- error tolerance 46, 52
- euler 1, 46, 54
- exit 18, 29, 37, 54, 57, 64-65, 69
- EXTERNAL 24, 61, 62
- extraderiv 38
- flags 18-19, 23, 29, 31, 38, 43
- GOOSEHOME 7-8
- h0 47
- halt 18, 29, 37
- hcur 47
- HEADER 19, 23, 60, 62
- help 1, 3, 14, 19, 22, 28-30, 37-38, 62
- hmax 47
- hmin 47
- hold 39, 51, 66, 68, 73-75
- hu 47
- imxer 47

include 1, 4, 7, 27, 30, 33, 64-65  
 inherit 3, 78-79  
 initialize 10, 15, 20, 35, 39, 54, 56-57,  
     65-67, 78  
 initlist 23, 43, 78  
 INITMETHOD 15, 20, 39, 47, 61, 63,  
     66, 78  
 initSrtFlg 47  
 INPUTS 12-16, 20, 26, 62-63, 78  
 INTEGRATE 12-13, 23, 57, 62-63, 76  
 INTERNAL 24, 62  
 iopt 47  
 istrate 47-48, 74  
 itask 47-49  
 itol 48-49, 53  
 ixpr 49  
 Jacobian matrix 1, 7, 48, 50, 56  
 list 39-41, 78  
 load 7, 39  
 lsodar 1, 38, 46-50, 53-54, 74  
 macro 33, 36, 72, 74, 77-78  
 max 41  
 mcur 50  
 messages 3-4, 21, 43, 66, 72, 76, 78  
 METHOD 21  
 method 3-4, 13-14, 16, 20-21, 23,  
     25-26, 43, 56, 59, 63, 66, 75,  
     77-79  
 mf 50  
 min 41  
 model 1, 3-4, 7, 9-10, 12, 16, 20,  
     25-26, 33, 35-36, 39, 42-44,  
     54, 56-57, 60, 63-67, 74-75,  
     77-78  
 mused 50  
 mxordn 50  
 mxords 50  
 mxstep 48, 50  
 neqns 50  
 nfe 51  
 nhold 51  
 nje 51  
 nonstiff 50, 54  
 notnil 25, 63  
 noutput 51  
 nperiods 51  
 nplot 51  
 nqcur 51  
 nqu 51  
 nst 51  
 object-oriented 1, 3, 70  
 Objective-C 1, 3-4, 8, 10-11, 19-22, 24,  
     26-27, 32, 62-64, 70  
 object 1, 3-4, 7, 9-10, 13, 15-16, 20-21,  
     24-26, 34-37, 39-41, 43-44,  
     54, 56-57, 60-63, 66,  
     72-75, 77-79  
 orgSort 52  
 OUTPUTS 12-16, 20, 26, 62-63, 79  
 output 40, 45, 51-52, 54-55, 66-67  
 panelobjs 40, 57  
 path 7-8  
 plot 4, 7, 39, 41, 51-52, 54-55, 63, 66,  
     69, 75  
 plotmax 52  
 ppcpu 52  
 prompt 24, 62  
 pwd 23, 31, 44  
 quit 18, 29, 37  
 range 41, 74  
 read 3, 7, 10-11, 21, 27, 30, 33, 42,  
     64-67, 73  
 REFERENCES 22-23  
 reitol 52  
 reset 7, 11, 21, 30, 42  
 respondsto 25, 61, 63, 76, 79  
 root 1, 25-26, 48, 52, 54, 56, 79  
 rtol 48-49, 53  
 Runtime Environment 1, 3, 7, 9-11, 13,  
     15, 20, 26-27, 29, 33, 46,  
     54, 56-57, 60, 63-66, 77-79  
 run 41-42, 51-53, 55-57, 66, 68, 73, 76  
 SAMPLING 14-15, 62-63, 79  
 save 3, 7, 10-12, 22, 39, 42, 62, 64,  
     72, 74  
 send 3, 21, 43, 54, 56, 66  
 shell 22, 31, 43  
 show 23, 31, 43, 66, 68, 74  
 showcwd 23, 31, 44  
 showfs 23, 32  
 showop 52, 55  
 slotlist 79  
 SLOTS 24, 61-62  
 slots 3-4, 9, 23-25, 34-35, 37, 39-41,  
     43, 45, 54, 56-57, 62-63,  
     66, 72, 75-79  
 sort 13, 46-47, 52, 63, 67-68, 77  
 stiff 50, 54  
 stop 18, 29, 37  
 subclass 3, 25, 63, 75, 78-79  
 superclass 3, 11, 78-79  
 syntax 3, 11, 27, 33, 44, 71, 73, 76

t 42, 52, 68  
t0 53  
Table1 35, 72  
tcrit 47, 49  
tcur 52  
TempleGraph 4, 52, 69  
tend 42, 53, 66-67, 76  
title 41  
tolsf 53  
trace 53  
tstart 42, 53, 66-67, 76  
tsw 53  
units 24, 62  
VALIDATE 25, 44, 61, 63  
validate 23, 44, 74, 79  
validation rules 25, 63, 66, 72, 74-76,  
79  
vectors 24, 57, 59  
waterp 45  
WHENDO 25-26, 79  
whenlist 23, 43, 79  
writeto 45, 72-75  
X Windows 4, 27, 54, 57

### INTERNAL DISTRIBUTION

- |        |                  |        |                                  |
|--------|------------------|--------|----------------------------------|
| 1-2.   | M. A. Abdalla    | 30.    | O. L. Smith                      |
| 3.     | J. L. Anderson   | 31.    | J. O. Stiegler                   |
| 4.     | S. J. Ball       | 32.    | B. K. Swail                      |
| 5.     | R. E. Battle     | 33.    | B. R. Upadhyaya                  |
| 6.     | E. D. Blakeman   | 34.    | R. E. Uhrig                      |
| 7.     | C. R. Brittain   | 35-40. | J. D. White                      |
| 8.     | N. E. Clapp, Jr. | 41.    | T. L. Wilson                     |
| 9.     | B. G. Eads       | 42.    | R. T. Wood                       |
| 10.    | D. N. Fry        | 43.    | B. Chexal, Advisor               |
| 11.    | H. E. Knee       | 44.    | V. Radeka, Advisor               |
| 12.    | J. E. Jones, Jr. | 45.    | R. M. Taylor, Advisor            |
| 13.    | J. March-Leuba   | 46-47. | Central Research Library         |
| 14.    | J. K. Mattingly  | 48.    | Y-12 Technical Reference Section |
| 15.    | J. K. Munro, Jr. | 49-50. | Laboratory Records               |
| 16-26. | D. J. Nypaver    | 51.    | Laboratory Records-RC            |
| 27.    | R. B. Perez      | 52.    | ORNL Patent Section              |
| 28.    | J. S. Powell     | 53.    | I&C Division Publications Office |
| 29.    | J. C. Schryver   |        |                                  |

### EXTERNAL DISTRIBUTION

54. Assistant Manager for Energy Research and Development, DOE-OR, P.O. Box 2001, Oak Ridge, TN 37831-8600
55. R. J. Neuhold, Director, Division of Advanced Technology Development, NE-462, DOE, Washington, DC 20585
56. B. J. Rock, Director, Office of Technology Support Programs, NE-46, DOE, Washington, DC 20585
57. H. Alter, Office of Technology Support Programs, NE-46, DOE, Washington, DC 20585
58. D. G. Carroll, GE Nuclear Energy, Program Manager, Control Technology Programs, 6835 Via Del Oro, San Jose, CA 95119-1315
59. Y. Dayal, GE Nuclear Energy, Advanced Nuclear Technology, 6835 Via Del Oro, San Jose, CA 95119
60. W. K. Wagner, GE Nuclear Energy, Nuclear Systems Technology Operation, P.O. Box 530954, San Jose, CA 95153-5354
61. H. P. Planchon, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
62. J. I. Sackett, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439
63. J. W. Cooke, Energy Programs Division, DOE-OR, P.O. Box 2001, Oak Ridge, TN 37831-8600
64. C. E. Ford, Liberty University, School of Lifelong Learning, P.O. Box 2000, Lynchburg, VA 24506
65. D. Ugolini, Via Pacinotti 29, Cesena (FO) 47023, Italy
66. L. A. Rovere, Centro Atomico Bariloche, 8400 S. C. de Bariloche, Argentina
- 67-69. T. W. Kerlin, The University of Tennessee, Nuclear Engineering Department, Knoxville, TN 37996-2300

- 70-71. L. Guimaraes, Centro Tecnico Aeroespacial, Instituto de Estudios Avanzados/ENU, Caixa Postal 60444, Sao Jose Campos, SP 12231, Brazil
- 72. C. March-Leuba, INITEC, Padilla 17, Madrid 28006, Spain
- 73-128. Given distribution as shown in DOE/OSTI-4500 under Category UC-530, Liquid Metal Fast Breeder Reactors