gPROMS Introductory User Guide

Process Systems Enterprise Ltd.

Bridge Studios
107a Hammersmith Bridge Road
London W6 9DA
United Kingdom

Tel : +44 (0)20 8563 0888
Fax : +44 (0)20 8563 0999

Release 2.3.1— June 2004

Disclaimer

Code from LAPACK and BLAS is used in gPROMS.
We would like to thank the authors
E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel,
J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling,
A. McKenney, and D. Sorensen
for making the code publicly available.

We would like to thank HQP's author, Ruediger Franke of ABB, for
his help in developing the interface from gPROMS to HQP.

Contents

Chapter 1

Introduction

Contents

1.1 What is gPROMS?

gPROMS is a **g**eneral **PRO**cess **M**odelling **S**ystem with proven capabilities for the simulation, optimisation and parameter estimation (both steady-state and dynamic) of highly complex processes.

1.2 gPROMS advantages

1.2.1 Clear, concise language

gPROMS allows the user to write equations almost as they would appear on paper. The clear, concise language allows the user to concentrate on getting the modelling equations correct while not having to be concerned with the complexity of the solution techniques. In addition, users can easily document comments in a manner that allows models to be passed on to other users transparently, and enables complex models to be quality assured.

1.2.2 Modelling power

gPROMS offers unparalleled modelling power for users. All solvers have been designed specifically for large-scale systems and there are no limits regarding problem size other than those imposed by the available machine memory. Dynamic simulations of models with over 100 000 differential-algebraic equations have been performed.

The generality of gPROMS means that it has been used for a wide variety of applications in petrochemicals, food, pharmaceuticals, speciality chemicals and automation. Furthermore, it has the potential to be used for any process that can be described by a mathematical model.

gPROMS is supplied with libraries of common process models that can be freely extended and customised to ensure applicability to customer's exact requirements.

gPROMS was the first general modelling system to allow the direct mathematical description of distributed unit operations where properties vary in one or more spatial dimensions. This frees the engineer from trying to construct crude approximations of such operations as series of well-mixed volumes and from being involved in complex mathematical manipulations of the process model. Since gPROMS has had these facilities for solving systems of integral, partial and ordinary differential and algebraic equations for many years, it has a long, proven track record in the simulation and optimisation of complex industrial processes including packed absorption/adsorption columns, chromatographic and membrane separators. Moreover, gPROMS has also been used to directly model solid-phase operations involving particle size distributions or distributions with respect to other properties such as molecular weight (*e.g.* batch and continuous crystallisation processes, grinding operations and polymerisation processes).

Finally, because gPROMS represents processes as sets of equations that can be solved in a number of modes – steady state simulation, dynamic simulation, steady-state optimisation, dynamic optimisation, steady-state parameter estimation, dynamic parameter estimation – it allows a single underlying model of a process to be evolved from concept to engineering design and operation. This minimises re-work and gives the possibility of multiple payback from the initial modelling effort.

1.2.3 Modelling of process discontinuities

The physical and chemical behaviour of most processes is inherently discontinuous. Changes take place abruptly and frequently due to phase transitions, flow regime transitions, geometrical limitations, and so on. gPROMS is unique amongst commercial simulators in its facilities for describing processes with discontinuities. Reversible-symmetric, reversible-asymmetric, and irreversible discontinuities can all be routinely handled. This has significant consequences for solution robustness and speed, and allows the simple handling of situations that often present a considerable obstacle to solution in other simulators. The algorithms within gPROMS were developed through years of high-level mathematical research, and automatically detect discontinuities, lock on to them, and then re-initialise rapidly.

1.2.4 Modelling of operating procedures

The detailed modelling of the operating procedures of a process is as important as describing the physics and chemistry of the various unit operations in it. From conception, gPROMS was designed to view processes as a combination of equipment models and their operating procedures, rather than the narrow view of first-generation simulation tools. gPROMS adopts a dual description for processes in terms of `MODEL`s (which describe the physical, chemical and biological behaviour of the process) and `TASK`s (which operate on `MODEL`s and describe the operating procedure that is used to run the process). The gPROMS `TASK` language is general and flexible and allows the description of highly complex operating procedures, each comprising a number of steps to be executed in sequence, in parallel, conditionally or iteratively.

This capability is of crucial importance when dealing with batch processes, where the description of the physical and chemical operations is only half the story (usually the less interesting half!); the other half being the description of the operating policy that is used to run the plant.

It is also extremely helpful for continuous processes that often exhibit transient behaviour either due to a transition between different operating points (*e.g.* grade transitions in polymerisation reactors) or due to abnormal conditions (*e.g.* equipment failures in safety studies).

gPROMS offers a major expansion in the scope of processes that can be modelled and optimised. These include integrated batch/semi-continuous plants, chromatographic processes, and periodic separation and reaction/separation processes.

The `TASK` language also allows the automatic generation of state-space models from nonlinear dynamic models in gPROMS, which is useful, for example, in control design applications.

1.2.5 Hierarchical modelling structure

gPROMS has an "object-oriented" approach to modelling that applies to both process models and operating procedures. In this way, a user can easily construct models of complex flowsheets and procedures by decomposing them into sub-models that call on other sub-models and can even inherit values of parameters. There is no limit on the number of levels in this modelling hierarchy.

1.2.6 Dynamic optimisation

Rigourous optimisation of equipment design, operating procedures, and so on, is the single most important benefit of using process modelling. In tools other than gPROMS this is commonly achieved by the user tweaking parameters and doing numerous, trial-and-error simulations while checking that process constraints are satisfied and measuring some performance measure. With this kind of *ad hoc* approach, there is no guarantee that a user will find any solution that satisfies all the constraints while the probability that he/she will find a mathematically optimal solution is virtually nil.

gPROMS was the first modelling system to have formal, mathematical algorithms for automatically optimising large-scale, dynamic processes (both lumped and distributed). As with its ability to model distributed systems, gPROMS has had this pioneering technology for at least five years longer than competitors and so has a proven track record with the most difficult of industrial problems.

gPROMS optimisation capabilities include taking into account integer or discrete decisions using Mixed Integer Optimisation (MIO). MIO can be applied to both steady-state and dynamic gPROMS models. These may also involve discontinuous equations such as those described by gPROMS IF and CASE equations.

Systems involving 40 000 time-varying quantities have been successfully optimised to date. Examples include optimal start-up and shut-down procedures; optimal design and operation of multi-phase batch/semi-batch reactors; optimal grade switching policies for continuous polymerisation reactors; optimal tuning of PID controllers; and nonlinear model predictive control.

1.2.7 Parameter estimation

In another industry first, gPROMS was the first general process modelling system with facilities for estimating model parameters through optimisation from both steady-state and dynamic experimental data. Parameter estimation is a key tool for model validation and gPROMS has been used for many years for this purpose in a broad variety of industrial applications.

gPROMS has a number of advanced features including the ability to estimate an unlimited number of parameters and to use data from multiple steady-state and dynamic experiments. It also gives the user complete flexibility in that they can specify different variance models for different variables in different experiments. Moreover, it has a built-in interface to MS Excel that allows the user to automatically test the statistical significance of results, generate plots overlaying model data and experimental data, plot confidence ellipsoids, and so on. With this enhanced statistical analysis, gPROMS has a marked technical edge over competitor products.

1.2.8 Project management

The gPROMS ModelBuilder makes it easy for users to construct and manage "projects" involving multiple process models, models of operating procedures, numerical simulation experiments, optimisation information, experimental data and parameter estimation files.

1.2.9 Open architecture

gPROMS has an unrivalled open software architecture that allows users to easily incorporate third-party software components within gPROMS and incorporate gPROMS within third-party applications. Five different categories of interface are currently supported, each via a formally defined and well-tested communication protocol:

- The Foreign Object Interface (FOI). This allows part of the model to be described by external software such as physical properties packages, legacy code for unit operations *etc.*, and CFD tools.
- The Foreign Process Interface (FPI). This allows executing gPROMS simulations to exchange information with external software such as real-time control systems, operator training packages and tailored front-ends for non- expert users.
- The Output Channel Interface (OCI). This allows external software to capture and manipulate all results produced by gPROMS simulations. A good example of this is the built-in interface that gives the user freedom to send and receive data from a gPROMS model to and from MS Excel without having to write any macros.
- The Open Solver Interface (OSI). This allows external mathematical solvers to be interfaced to gPROMS.

All of the above software components may reside and be executed on a different machine (potentially of a different type and/or operating system) to that on which gPROMS is executed. All communication is handled in a manner that is completely transparent to the user via the gNET facility.

Finally, gPROMS is available on a wide range of platforms (including DIGITAL UNIX, AIX, IRIX, Solaris, Linux and Windows NT/2000/XP). This allows users to upgrade and change their hardware and operating systems without having to worry about gPROMS compatibility.

1.3 Outline of this User Guide

This Introductory User Guide is designed to equip new users with all they need to know about how to write models in gPROMS, run simulations and use gPROMS to communicate with packages such as MS Excel, for which interfaces already exist. The use of gPROMS for optimisation and parameter estimation is described in the gPROMS Advanced User Guide which also provides details on how to use gPROMS with other packages such as physical property packages.

Chapter 2 gives an introduction to the gPROMS language and the gPROMS Model-Builder environment by guiding the user through a simple tutorial example. It also explains how to run gPROMS and how to display graphical output.

Chapters 3 to 5 then discuss various features of gPROMS syntax. Chapter 3 deals with arrays and built-in mathematical functions; Chapter 4 details how to model physical discontinuities; while Chapter 5 explains how to set up models of integral, partial and ordinary differential and algebraic equations in gPROMS.

Chapter 6 describes how complex models and flowsheets can be conveniently constructed in gPROMS using a hierarchical modelling approach.

Chapters 7 and 8 discuss how to model simple and complex operating procedures in gPROMS, respectively, again making use of the concept of hierarchical modelling.

Chapter 9 describes how to use gPROMS for conducting stochastic simulations.

Chapter 10 explains how you can control various aspects of model- based activities carried out in gPROMS. This includes not only the simulation activities described in this document, but also the optimisation and parameter estimation activities described in chapters 2 and 3 respectively of the gPROMS Advanced User Guide.

Finally, the Appendices describe the model diagnostic facilities of gPROMS and give more information on the various modes of generating and displaying results in gPROMS.

Chapter 2

An Overview of gPROMS

Contents

This chapter provides an overview of the main features of gPROMS.

We go through a tutorial that examines the process of modelling and performing a dynamic simulation of a very simple unit operation. Our aim in the tutorial is to introduce the reader to the gPROMS ModelBuilder environment and the basic ideas behind the gPROMS language.

The detailed description of more advanced features of the language (*e.g.* the use of arrays, structured and conditional equations, *etc.*) is postponed until later chapters. A more comprehensive introduction to the gPROMS ModelBuilder environment is also given later in this guide.

2.1 Starting gPROMS

2.1.1 Using gPROMS on MS Windows platforms

In order to create new models, run existing ones, and so on, the user must enter the gPROMS ModelBuilder environment. In Windows this is usually done from the `Start` menu: left click on the `Start` menu and select `Programs`, then `Process Systems Enterprise` and, finally, `gPROMS` (figure 2.1)[1]

This will automatically launch the user interface, as shown in figure 2.2. Starting gPROMS ModelBuilder will also start the gRMS application. gRMS stands for gPROMS Results Management System and is discussed in section 2.4.1 when we will look at plotting simulation results.

2.1.2 Using gPROMS on Unix platforms

This section describes how to set up your account to use gPROMS on computer systems running under the UNIX operating system. Platforms that are currently supported include Linux, SUN Solaris, DIGITAL UNIX, IBM AIX and SGI IRIX.

2.1.2.1 Setting up your account to use gPROMS

Before you can run gPROMS for the first time, you need to set up your account appropriately as described below. This procedure needs to be executed **once only**.

(a) If you are using the UNIX C Shell, then execute instruction (a1). If you are using the UNIX Bourne Shell or Korn Shell, then execute instruction (a2).

(a1) UNIX C Shell

Modify your `.cshrc` file[2] to include in your "path"[3] the directory in which gPROMS has been installed in your computer system.

For instance, if your system administrator has installed gPROMS in the default directory, `/usr/local/pse/gPROMS/bin`, add the following lines at the end of your `.cshrc` file:

```
setenv GPROMSHOME /usr/local/pse/gPROMS
if (-d $GPROMSHOME/bin && -x $GPROMSHOME/bin) then
  set path=($path $GPROMSHOME/bin)
endif
```

[1]Alternately, gPROMS can be started by typing `gPROMS` at a command prompt

[2]This normally resides in the home directory of your account.

[3]This is the set of system and other directories that UNIX automatically searches on your behalf when looking for a particular item of software – in this case, gPROMS.

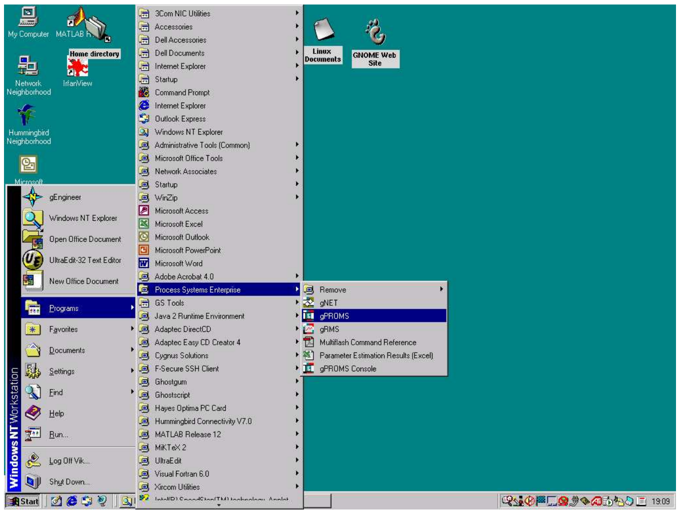

Start
Programs
Process Systems Enterprise
Remove
gNET
gPROMS
gRMS
Multiflash Command Reference
Parameter Estimation Results (Excel)
gPROMS Console

Figure 2.1: Starting gPROMS from the `Start` menu.

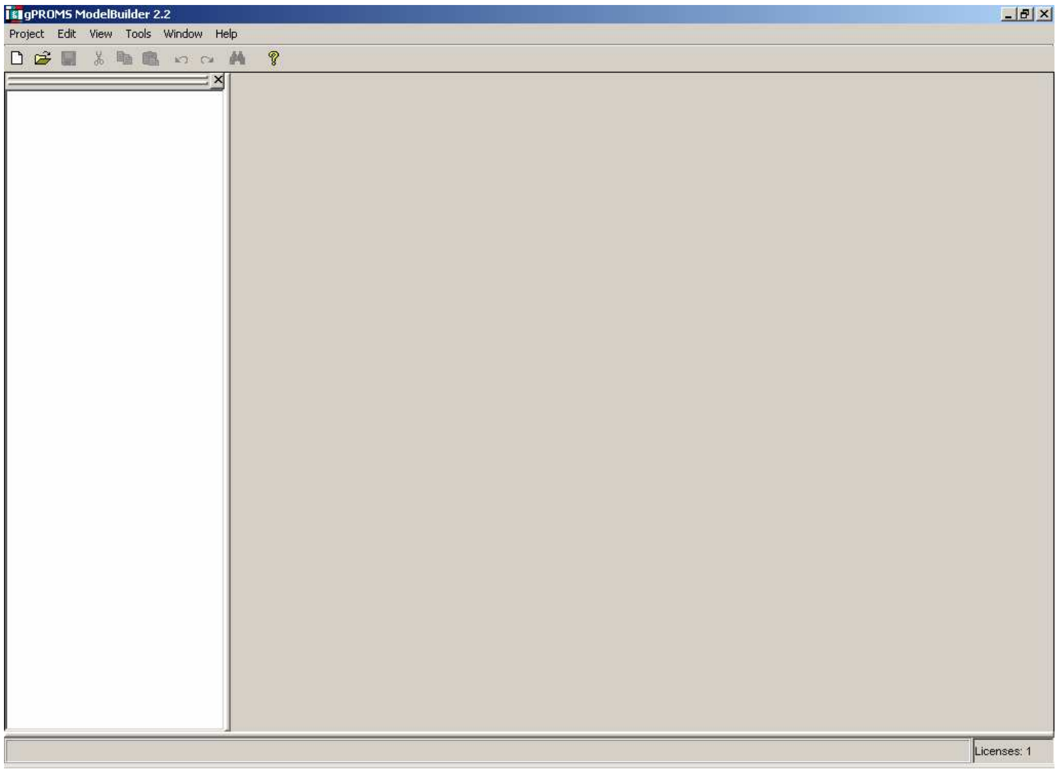


Figure 2.2: The gPROMS ModelBuilder interface.

(a2) UNIX Bourne Shell or Korn Shell

Modify your `.profile` file[4] to include in your "path" the directory in which gPROMS has been installed in your computer system.

For instance, if your system administrator has installed gPROMS in the default directory, `/usr/local/pse/gPROMS/bin`, add the following lines at the end of your `.profile` file:

```
GPROMSHOME=/usr/local/pse/gPROMS
if test -d $GPROMSHOME/bin; then
  PATH=${PATH}:${GPROMSHOME}/bin
fi
export GPROMSHOME PATH
```

This step may be unnecessary if gPROMS has been installed in a directory which is already in your path.

(b) Ensure that any changes made at step (a) above become effective by logging out of your account and logging in again.

2.1.2.2 Entering the gPROMS environment

In order to enter the gPROMS environment[5], simply type:

```
gPROMS
```

If gPROMS is properly installed, the system will respond by bringing up the Model-Builder interface, similar to that shown in figure 2.2. The gRMS ("gPROMS Results Management System") application should also be started (if not already open) - this is discussed further in section 2.4.2.

[4]This normally resides in the home directory of your account.

[5]If you intend to run gPROMS on a *remote* UNIX workstation:

- from the workstation you are working on, type `xhost +yyyyyy`, where `yyyyyy` is the name of the remote workstation; then
- log into the remote machine and issue the command `setenv DISPLAY xxxxxx:0.0` where `xxxxxx` is the name of the workstation on which you are currently working.

Note that you can obtain the name of a workstation by typing `hostname`.

2.2 Developing a simple gPROMS model

2.2.1 Introduction

So far, we have described how to set up your account and how to enter the gPROMS ModelBuilder environment on both UNIX and MS Windows workstations. We will now look at how to create a dynamic model and run a simulation of a simple unit operation using gPROMS.

The example system is a simple buffer tank with gravity-driven outflow (figure 2.3). It is a good choice for illustrating the main features of the gPROMS language because it comprises only one simple unit operation, for which a *primitive* model can be constructed. Primitive models are mathematical models that are completely specified in terms of explicitly declared variables and equations. They usually correspond to simple unit operations or parts thereof. As will be seen in later chapters, they form the building blocks for the construction of higher-level, *composite* models of complex unit operations or entire process flowsheets.

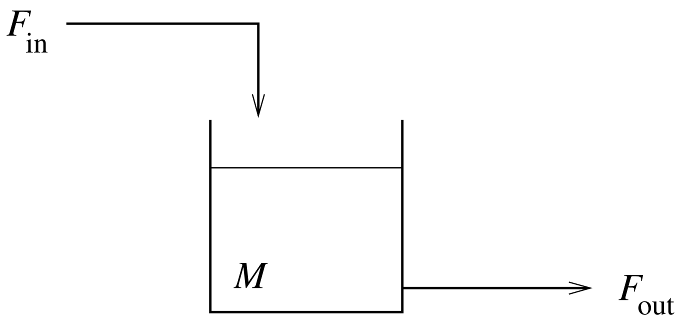


Figure 2.3: Buffer tank with gravity-driven outflow.

The dynamic mathematical model of the buffer tank process takes the following form:

Mass balance

$$\frac{dM}{dt} = F_{\text{in}} - F_{\text{out}} \tag{2.1}$$

Relation between liquid level and holdup

$$\rho A h = M \tag{2.2}$$

Characterisation of the output flowrate

$$F_{\text{out}} = \alpha\sqrt{h} \tag{2.3}$$

Here, M and h are the mass and level of liquid in the tank, and F_{in} and F_{out} are the inlet and outlet flowrates respectively. ρ, A and α denote the density of the liquid material, the cross-sectional area of the tank and the outlet valve constant, respectively. For the purposes of this example, these last three quantities are assumed to be known constants.

2.2.2 New gPROMS Project

The first step that needs to be taken when modelling a new process is to create a new gPROMS "Project". This is achieved by left clicking on the `Project` menu on the task bar of the gPROMS user interface, then left clicking on `New`, as shown in figure 2.4. This will bring up a tree in the left-hand pane containing a number of entries (see figure 2.5):

- `Variable Types`
- `Stream Types`
- `Models`
- `Tasks`
- `Processes`
- `Optimisations`
- `Estimations`
- `Experiments`
- `Saved Variable Sets`
- `Miscellaneous Files`

Each of these entries represents a group of gPROMS *Entities*. We will learn more about these in subsequent sections. In the meantime, you can rename the Project from its default of "`gPROMS_Project_1`" to a name of your choice by left-clicking on `Project`, then `Save As...` and then altering the name in `File Name` (see figures 2.6a, b, c and d). In the example shown, the Project is saved as "`Tank.gPJ`".

Note that two new menus, `Entity` and `Activities`, appeared when you created the Project. These two will only ever be visible when a Project is selected in the tree.

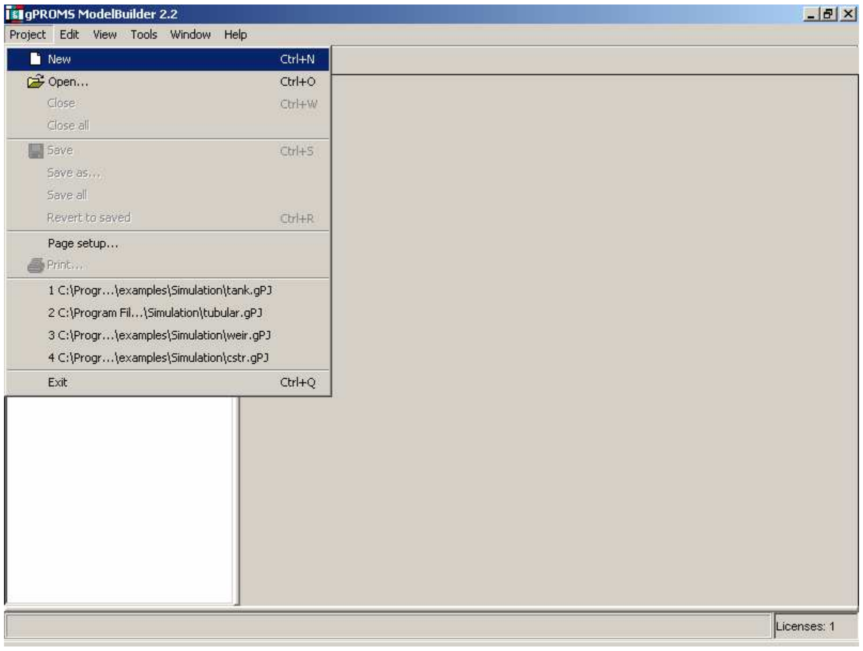

gPROMS ModelBuilder 2.2
Project Edit View Tools Window Help
New Ctrl+N
Open... Ctrl+O
Close Ctrl+W
Close all
Save Ctrl+S
Save as...
Save all
Revert to saved Ctrl+R
Page setup...
Print...
1 C:\Progr...\examples\Simulation\tank.gPJ
2 C:\Program Fil...\Simulation\tubular.gPJ
3 C:\Progr...\examples\Simulation\weir.gPJ
4 C:\Progr...\examples\Simulation\cstr.gPJ
Exit Ctrl+Q
Licenses: 1

Figure 2.4: Creating a new gPROMS Project.

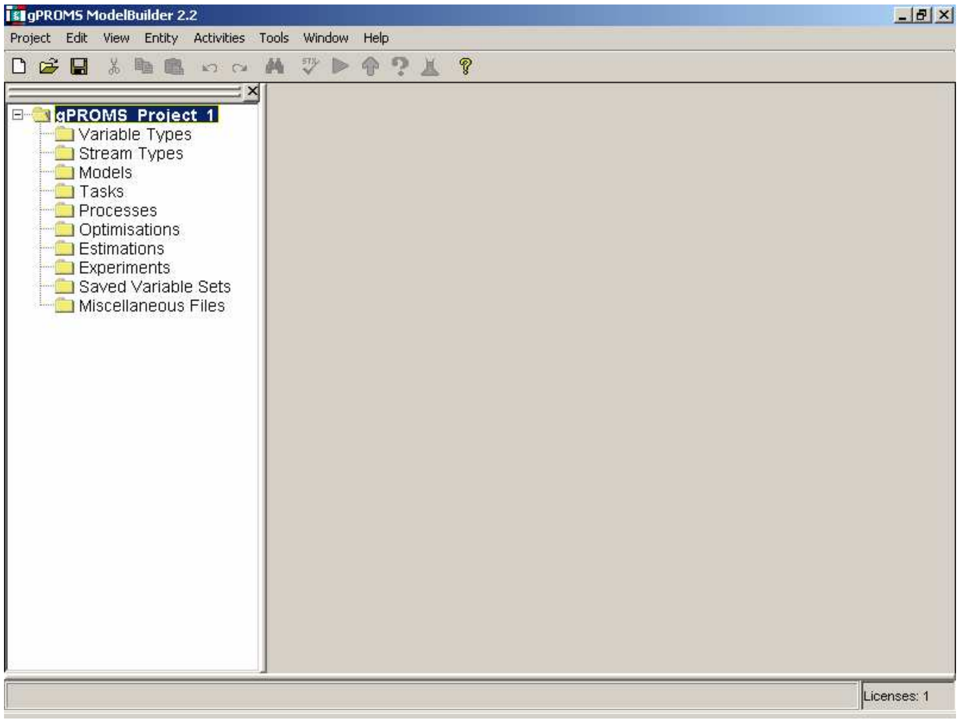
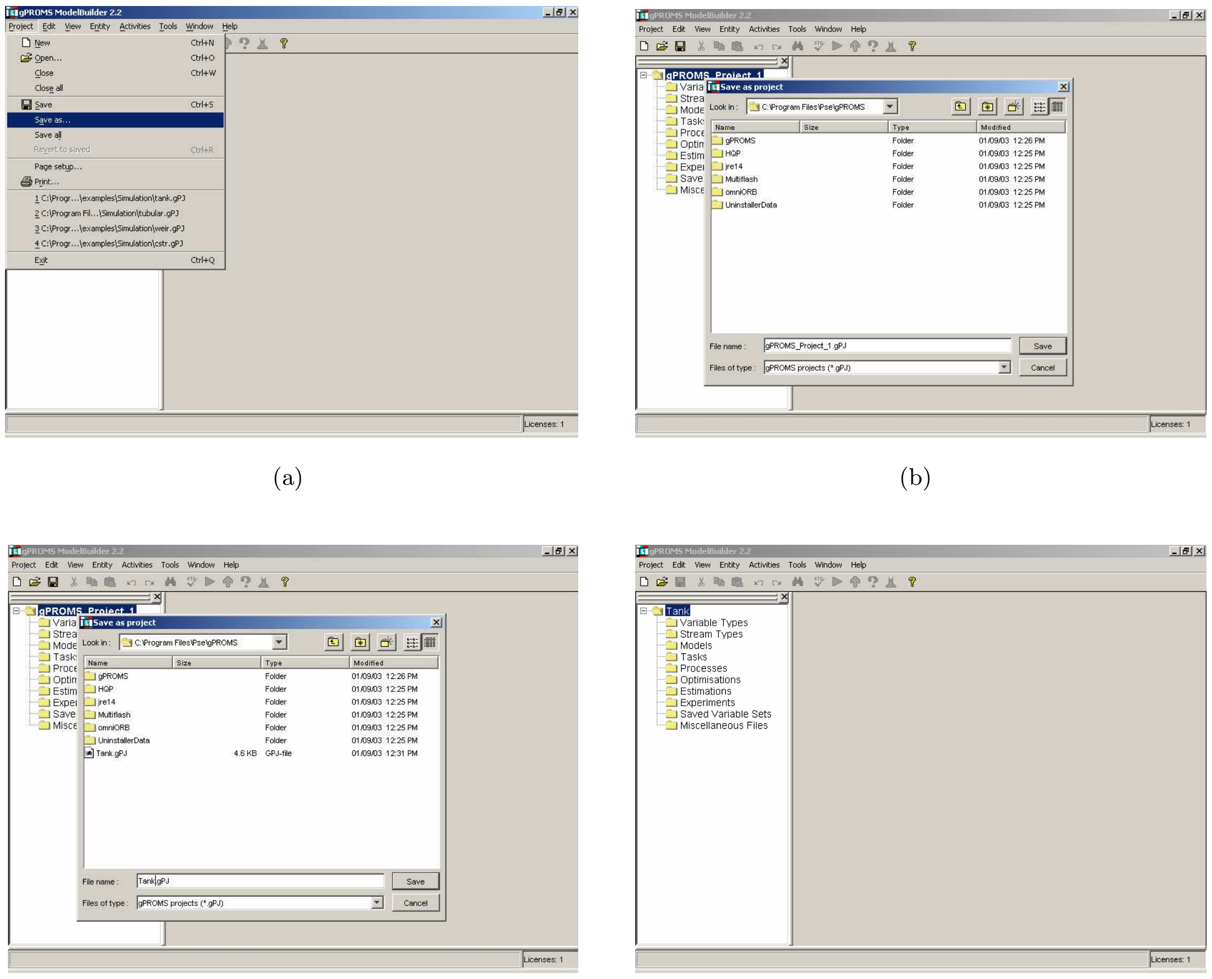


Figure 2.5: A gPROMS Project tree.


gPROMS ModelBuilder 2.2
Project Edit View Entity Activities Tools Window Help
New Ctrl+N
Open... Ctrl+O
Close Ctrl+W
Close all
Save Ctrl+S
Save as...
Save all
Revert to saved Ctrl+R
Page setup...
Print...
1 C:\Progr...\examples\Simulation\tank.gPJ
2 C:\Program Fil...\Simulation\tubular.gPJ
3 C:\Progr...\examples\Simulation\weir.gPJ
4 C:\Progr...\examples\Simulation\cstr.gPJ
Exit Ctrl+Q
Licenses: 1
(a)
gPROMS_Project_1
Save as project
Look in: C:\Program Files\Pse\gPROMS
File name: gPROMS_Project_1.gPJ
Files of type: gPROMS projects (*.gPJ)
Save
Cancel
(b)
Save as project
Look in: C:\Program Files\Pse\gPROMS
File name: Tank.gPJ
Files of type: gPROMS projects (*.gPJ)
Save
Cancel
(c)
Tank
Variable Types
Stream Types
Models
Tasks
Processes
Optimisations
Estimations
Experiments
Saved Variable Sets
Miscellaneous Files
(d)

Name	Size	Type	Modified
gPROMS		Folder	01/09/03 12:26 PM
HQP		Folder	01/09/03 12:25 PM
jre14		Folder	01/09/03 12:25 PM
Multiflash		Folder	01/09/03 12:25 PM
omniORB		Folder	01/09/03 12:25 PM
UninstallerData		Folder	01/09/03 12:25 PM
Tank.gPJ	4.6 KB	GPJ-file	01/09/03 12:31 PM

Figure 2.6: Renaming a gPROMS Project.

2.2.3 Describing physical behaviour - `MODEL`s

The first type of gPROMS Entity that we will look at is `MODEL`. This appears as the third entry down in the gPROMS Project tree shown in figure 2.5.

In gPROMS, the declaration of primitive process models is done via `MODEL`s. A gPROMS Project should contain at least one `MODEL`. A `MODEL` contains a mathematical description of the physical behaviour of a given system. It comprises a number of sections, each containing a different type of information regarding the system being modelled.

In order to create a new `MODEL`:

1. Go to the `Entity` menu, which is the fourth menu from the left on the top pane, and left-click on `New Entity...` (figure 2.7a). This will cause a dialog box to appear.
2. Pull down the `Entity Type` menu and choose `MODEL`.
3. Fill in the `Name` field with a name of your choice (*e.g.* `BufferTank`) (see figure 2.7b and c).
4. At this point the user has an option of including a gPROMS language template by checking the "Use template" box. This template provides help on gPROMS syntax. A brief description for the `MODEL` can also be added at this point.
5. When finished click `OK` (see figure 2.7b and c).

This will bring up an Entity editor window which has the name of the `Project` followed by the name of the `MODEL` at the top left corner. Since a `MODEL` Entity has now been created, the Project tree in the left-hand pane now shows a "key" symbol next to `MODEL`s (see figure 2.7d).

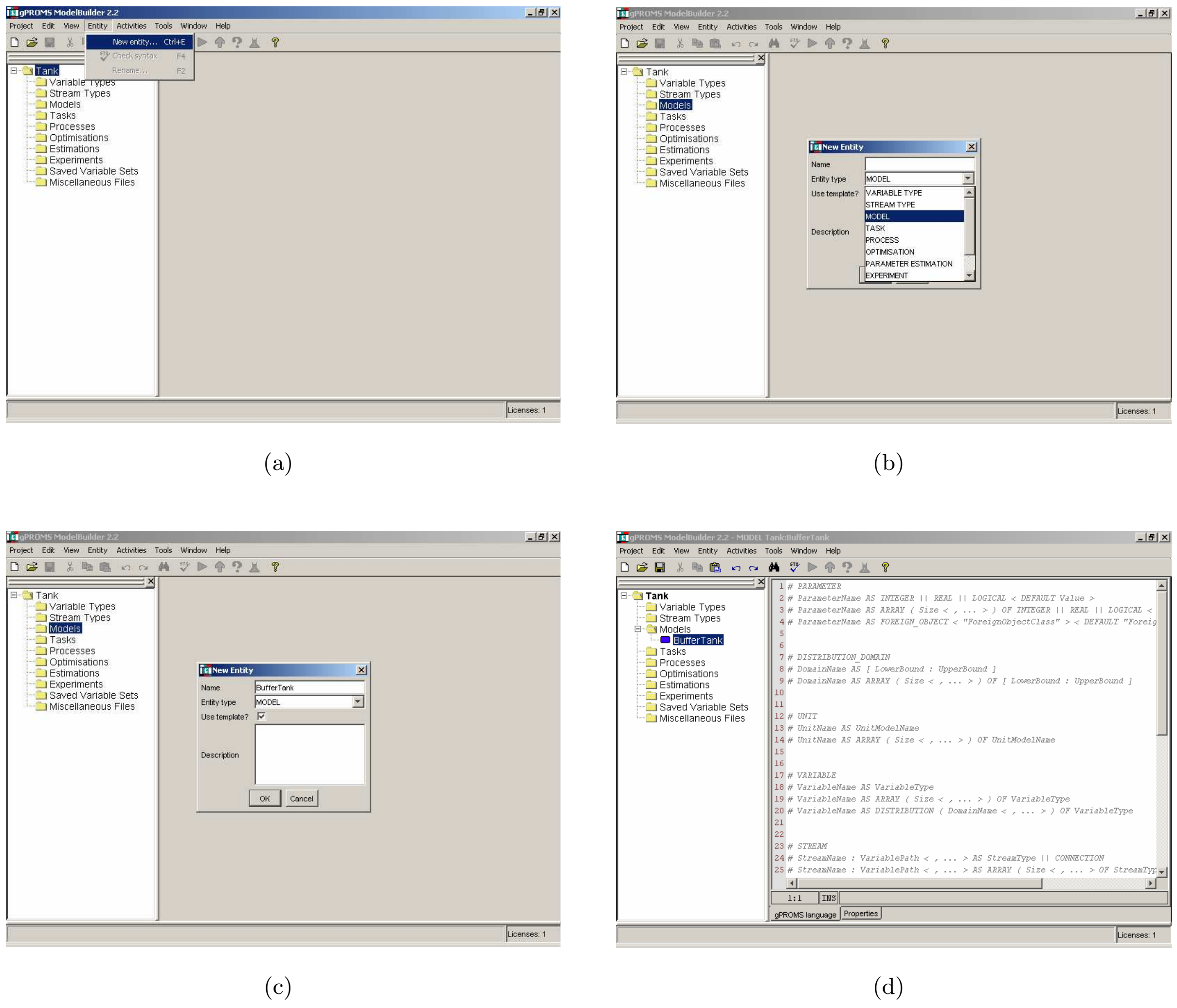
Steps 1 and 2 above can be combined in a short-cut method by right clicking on the `MODEL`s symbol in the left-hand pane and selecting `New Entity...`. This is shown in figure 2.8. In fact, this alternative can be used for all types of Entity.

The `MODEL` Entity editor window has two "tabs": a gPROMS language tab and a Properties tab. Further information on Entity editors can be found by reference to the ModelBuilder User Guide.

2.2.3.1 The Properties tab

All Entity editors in gPROMS ModelBuilder have an Entity Properties tab.

The "Properties tab" shows the current description of the Entity - this is an arbitrary text provided by the Entity developer(s) for future reference. It also contains various read-only information that is maintained automatically by the ModelBuilder, such as Entity creation and last modification information, including the user who performed these actions, and their times and dates. This is shown in figure 2.9.



Name: BufferTank

Entity type: MODEL

```
# PARAMETER
# ParameterName AS INTEGER || REAL || LOGICAL < DEFAULT Value >
# ParameterName AS ARRAY ( Size < , ... > ) OF INTEGER || REAL || LOGICAL <
# ParameterName AS FOREIGN_OBJECT < "ForeignObjectClass" > < DEFAULT "Foreig


# DISTRIBUTION_DOMAIN
# DomainName AS [ LowerBound : UpperBound ]
# DomainName AS ARRAY ( Size < , ... > ) OF [ LowerBound : UpperBound ]


# UNIT
# UnitName AS UnitModelName
# UnitName AS ARRAY ( Size < , ... > ) OF UnitModelName


# VARIABLE
# VariableName AS VariableType
# VariableName AS ARRAY ( Size < , ... > ) OF VariableType
# VariableName AS DISTRIBUTION ( DomainName < , ... > ) OF VariableType


# STREAM
# StreamName : VariablePath < , ... > AS StreamType || CONNECTION
# StreamName : VariablePath < , ... > AS ARRAY ( Size < , ... > OF StreamTyp
```

1:1 INS

Figure 2.7: Creating a new model

2.2.3.2 The gPROMS language tab

Almost all Entity editors in gPROMS ModelBuilder have a tab that displays and allows the editing of the representation of the Entity in the gPROMS language. To support the creation and modification of each Entity, ModelBuilder automatically employs syntax-sensitive highlighting of the gPROMS language.

The `MODEL` that describes the buffer tank process is shown in figure 2.10. We suggest that you type this `MODEL` in yourself before we go on to explain the various sections. As you do so, notice how the bottom left-hand corner of the text editor gives you the line number and column in the format `line number:column`.

The minimal information that needs to be specified in any `MODEL` is the following:

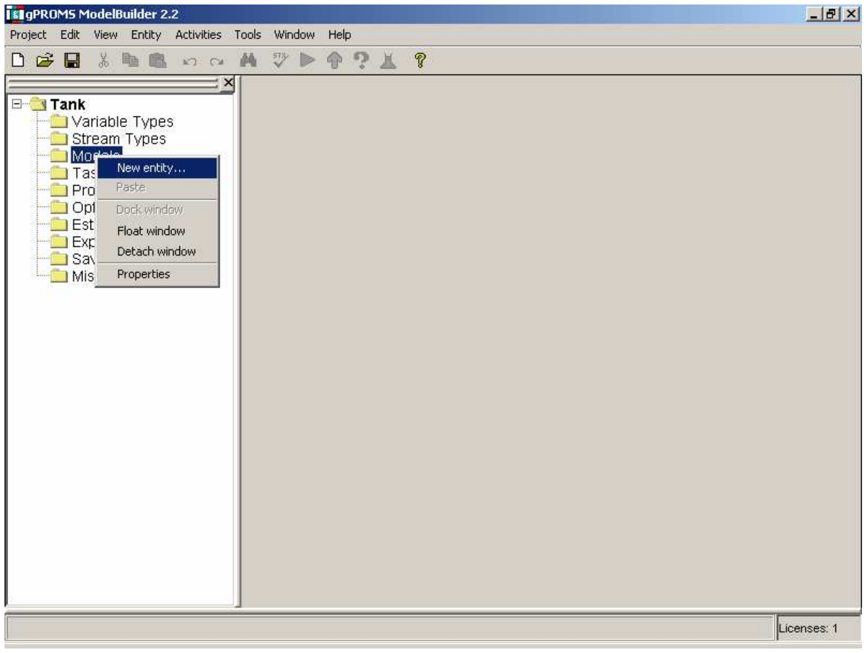
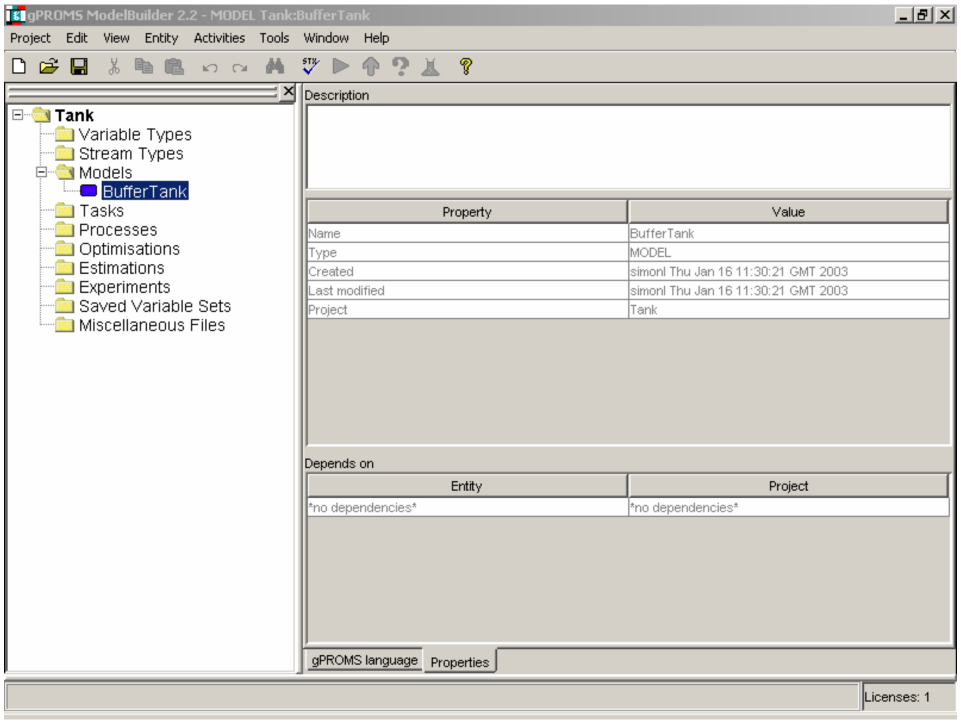


Figure 2.8: An alternative way of creating a new MODEL

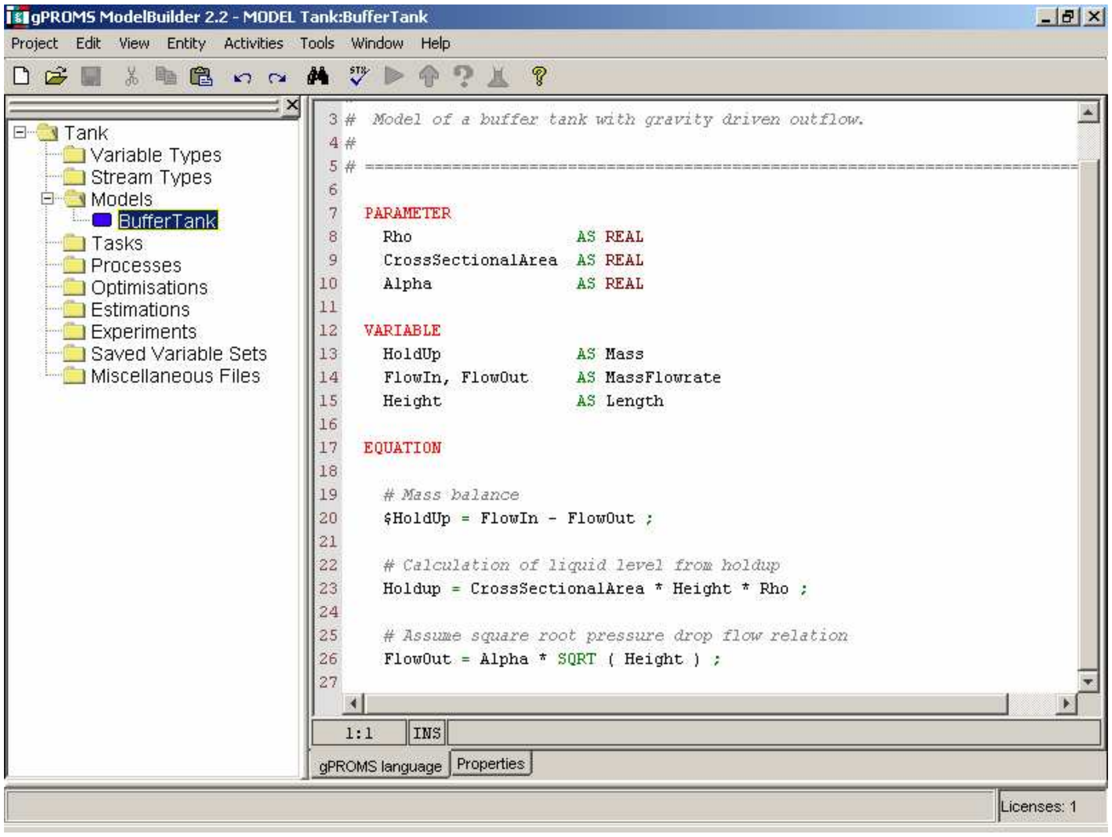


Property	Value
Name	BufferTank
Type	MODEL
Created	simonl Thu Jan 16 11:30:21 GMT 2003
Last modified	simonl Thu Jan 16 11:30:21 GMT 2003
Project	Tank

Depends on

Entity	Project
no dependencies	*no dependencies*

Figure 2.9: Entity Properties



```
# Model of a buffer tank with gravity driven outflow.
#
# ==========================================================================

PARAMETER
  Rho                  AS REAL
  CrossSectionalArea   AS REAL
  Alpha                AS REAL

VARIABLE
  HoldUp               AS Mass
  FlowIn, FlowOut      AS MassFlowrate
  Height               AS Length

EQUATION

  # Mass balance
  $HoldUp = FlowIn - FlowOut ;

  # Calculation of liquid level from holdup
  Holdup = CrossSectionalArea * Height * Rho ;

  # Assume square root pressure drop flow relation
  FlowOut = Alpha * SQRT ( Height ) ;
```

Figure 2.10: Buffer tank `Model`

- A set of constant *parameters* that characterise the system. These correspond to quantities that will *never* be calculated by any simulation or other type of calculation making use of this `MODEL`. Therefore, their values must always be specified before the simulation begins and remain unchanged thereafter. They are declared in the `PARAMETER` section.
- A set of *variables* that describe the time-dependent behaviour of the system. These may be specified in later sections or left to be calculated by the simulation. They are declared in the `VARIABLE` section.
- A set of *equations* involving the declared variables and parameters. These are used to determine the time-dependent behaviour of the system. They are declared in the `EQUATION` section.

Overall, the structure of a simple `MODEL` declaration in the gPROMS language is the following[6]:

> `PARAMETER`
>
> ... *Parameter declarations* ...
>
> `VARIABLE`
>
> ... *Variable declarations* ...
>
> `EQUATION`
>
> ... *Equation declarations* ...

The next three sections take a more detailed look at the `PARAMETER`, `VARIABLE` and `EQUATION` sections.

2.2.3.3 The `PARAMETER` section

The `PARAMETER` section is used to declare the parameters of a `MODEL`. As mentioned before, parameters are time-invariant quantities that will *not*, under any circumstances, be the result of a calculation. Quantities such as physical constants (π, R, *etc.*), Arrhenius coefficients and stoichiometric coefficients usually fall into this category. In the buffer tank process, ρ, A and α were assumed constant and are thus declared as parameters of the BufferTank `MODEL`:

```
PARAMETER
  Rho                     AS REAL
  CrossSectionalArea      AS REAL
  Alpha                   AS REAL
```

Each parameter has a unique name (identifier) by which it can be referenced (for example, in expressions). Identifiers in the gPROMS language start with a letter (`a-z` and `A-Z`) and may comprise letters, numbers (`1-9`) and underscores (`_`). The gPROMS language is *not* case sensitive, *i.e.* `Temp` and `TEMP` are considered to be identical.

Each parameter is also declared to be of a certain type (*e.g.* `INTEGER` or `REAL`). All three parameters of the BufferTank `MODEL` are of type `REAL`.

Parameter declarations may optionally include the assignment of default values. For instance:

[6]When describing gPROMS syntax, we adopt the convention that typewriter style `CAPITALS` denote gPROMS keywords and mixed-case *italics* indicate information to be supplied by the user (*e.g.* the names of parameters, variables, models, *etc.*). Sections of an actual gPROMS input file are shown entirely in typewriter style, with keywords in capitals and user input in mixed case.

```
PARAMETER
  NoComp      AS INTEGER
  NoReactions AS INTEGER DEFAULT 1
```

Finally, note that the categorisation of certain quantities as parameters is sometimes tenuous. Designating a quantity as a parameter has the advantage of reducing the total number of variables in a model. However, this quantity can no longer be treated as an unknown in any future use of the model. Consider, for instance, the quantities that characterise the size and geometry of a vessel. From the point of view of dynamic simulation, these can be viewed as `PARAMETER`s. However, from the point of view of steady-state design calculations performed with the same model, these quantities may be considered unknowns under certain circumstances. It may, therefore, be better to classify them as `VARIABLE`s (see below).

2.2.3.4 The `VARIABLE` section

The `VARIABLE` section is used to declare the variables of a `MODEL`. These represent quantities that describe the time-dependent behaviour of a system. For instance, in the example process, M, h, F_{in} and F_{out} are variables of the BufferTank `MODEL`:

```
VARIABLE
  HoldUp           AS Mass
  FlowIn, FlowOut  AS MassFlowrate
  Height           AS Length
```

Like parameters, variables are declared to be of certain type. However, variable types are *user-defined*. The declaration of these variable types is discussed in section 2.2.4.

2.2.3.5 The `EQUATION` section

The `EQUATION` section is used to declare the equations that determine the time trajectories of the variables already declared in the `VARIABLE` section.

The gPROMS language is purely declarative, *i.e.* the order in which the equations are declared is of no importance. Simple equations are equalities between two real expressions (figure 2.10). These expressions may comprise:

- Integer or real constants (*e.g.* 2, 3.14159, *etc.*).
- Parameters that have been declared in the `PARAMETER` section (*e.g.* `Rho`, `Alpha`, `PI`, *etc.*).

- Variables that have been declared in the `VARIABLE` section (*e.g.* `HoldUp`, `Height`, `FlowOut`, *etc.*). The special symbol `$` preceding a variable name denotes the derivative with respect to time of that variable (*e.g.* `$HoldUp` *etc.*).

Similarly to most programming languages, expressions are formed by combining the above operands with the arithmetic operators `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division) and `^` (exponentiation), as well as built-in intrinsic functions (*e.g.* square root: `SQRT()`). The latter are described in greater detail in Chapter 3.

Intrinsic functions have the highest precedence priority, followed by the `^` operator and then the division and multiplication operators. The addition and subtraction operators have the lowest precedence. Naturally, parentheses may be used to alter these precedence rules as required.

Finally, note that comments can be added to clarify the contents of the `MODEL` where needed. As shown in figure 2.10, gPROMS accepts two types of comments. One type begins with `#` and extends to the end of the current line. The other type starts with { and ends with } and may span multiple lines. Moreover, comments of this type may be nested within one another.

2.2.4 Declaring variable types

We now turn our attention to another type of gPROMS Entity, namely `Variable Types`. These appear under the first entry in the Project tree shown in figure 2.5. As we have seen in section 2.2.3.4, each `VARIABLE` in a `MODEL` is declared to be of a particular variable type. Variable types are user-defined. However, their declaration is not part of the `MODEL` Entity itself. Instead, they are declared as new Entities so that `VARIABLE`s from different `MODEL`s can belong to the same type.

In order to create a new `Variable Type`:

1. Pull down the `Entity` menu from the top bar and click on `New Entity`. A dialog box will appear.
2. Choose `Variable Type` for the `Entity Type`.
3. Fill in the `Name` field and click on `OK`.

The right-hand pane now contains a table with some default values and a "key" symbol appears next to `Variable Types` in the Project tree in the left-hand pane. To help the user navigate the various `Variable Types`, when a particular `Variable Type` is selected in the Project tree, the corresponding row is automatically highlighted in the `Variable Types` table.

Figure 2.11 shows the `Variable Types` for the buffer-tank process. Three variable types (`Mass`, `MassFlowrate` and `Length`) are declared, in accordance with the types shown in figure 2.10. Notice how these are automatically listed in alphabetical order in both panes of the editor.

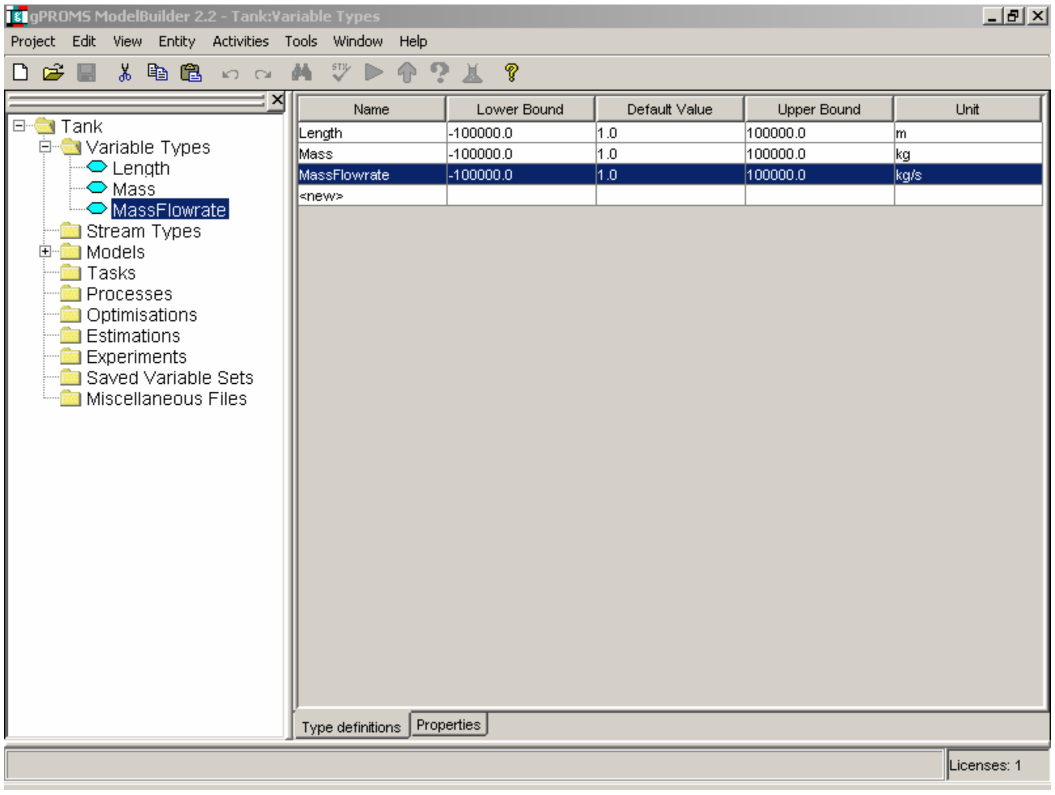
It is also possible to add a new `Variable Type` by completing the entries in a blank row that always appears at the bottom of the `Variable Types` table. When this happens, a corresponding new `Variable Type` is created and a new blank row is automatically appended to the table.

In gPROMS, all variables are real numbers. Variable types are refinements of the simple real type and include the following information:

- A *name*, to which the type may be referred globally.
- A *default value* for variables of this type. This value will be used as an initial guess for any iterative calculation involving variables of this type, unless it is overridden for individual variables or a better guess is available from a previous calculation.
- *Upper and lower bounds* on the values of variables of this type. Any calculation involving variables of this type must give results that lie within these bounds. These bounds can be used to ensure that the results of a calculation are physically meaningful; they can also be used to select the desired solution in situations where multiple solutions may exist. Again, these bounds may be overridden for individual variables of this type.
- An optional *unit of measurement*. Whenever variables of this type are reported, their values will be accompanied by this unit of measurement.

Hence, in figure 2.11, the right-hand pane shows these four properties and the names of all of the `Variable Types`. The values of the lower bounds, initial values and upper bounds are checked for consistency (*i.e.* you cannot enter an initial value outside the bounds or enter a lower bounds greater than the upper bound).

Also notice that the units of measurement are not enclosed in double quotes (`"`), which are normally used to denote strings. This is because Model Builder knows that this Entity must be a string. In general, when entering strings in `MODEL` and other Entities, double quotes must be used to signify this.



Name	Lower Bound	Default Value	Upper Bound	Unit
Length	-100000.0	1.0	100000.0	m
Mass	-100000.0	1.0	100000.0	kg
MassFlowrate	-100000.0	1.0	100000.0	kg/s
<new>				

Figure 2.11: `Variable Types` for the BufferTank `MODEL`.

2.2.5 Describing simulation activities - PROCESSes

So far, we have seen how the gPROMS language can be used to define the physical behaviour of a system in terms of a `MODEL` that contains `PARAMETER`, `VARIABLE` and `EQUATION` declarations. However, we have not actually specified *what* we want to do with this model. Indeed, a model can usually be used to study the behaviour of the system under many different circumstances. Each such specific situation is called a *simulation activity*. We now proceed to describe how the information provided so far can be used to specify simulation activities.

The gPROMS language makes a clear distinction between, on one hand, the *model* which represents a generic class of systems and, on the other hand, the specific details of one or more *instances* of this model employed by particular activities.

For instance, initial conditions are *not* defined within a `MODEL`. Instead, they remain unspecified until an instance of a `MODEL` is used for a particular dynamic simulation activity. In this way, a system can be simulated under different sets of initial conditions without any alterations to its underlying `MODEL`[7].

Moreover, as will be seen in later chapters, a simulation activity may involve multiple instances of the same `MODEL` used to describe a number of equipment items of the same type. The characteristics of each individual item may be different; they are specified by providing appropriate information within the context of the simulation activity.

The coupling of `MODEL`s with the particulars of a dynamic simulation activity is done in a `PROCESS`. To create a new `PROCESS`, follow a similar procedure to that described earlier for `MODEL`s and `VARIABLE TYPE`s, *i.e.* pull down the `Entity` menu, click on `New Entity`, choose `PROCESS` for the `Entity Type`, give it a name in the `Name` field and then click on `Create` (or right click on the `Processes` symbol, select `New Entity...`, enter the name and description and click `OK`). As with new `MODEL` Entities, new `PROCESS`es can contain a template giving the syntax of the `PROCESS` section.

Note that a gPROMS Project may contain multiple `PROCESS`es, each corresponding to a different simulation activity (*e.g.* simulation of plant startup, simulation of plant shutdown, *etc.*). Each such `PROCESS` must be given a different name and these will be automatically placed in alphabetical order in the gPROMS Project tree.

A `PROCESS` is partitioned into sections, each containing information required to define the corresponding dynamic simulation activity:

```
UNIT
  ... Process equipment declarations ...
SET
```

[7] In fact, the same model can be used to perform a variety of other activities, other than dynamic simulation (*e.g.* steady-state and dynamic optimisation, steady-state and dynamic parameter estimation, *etc.*). This document only considers dynamic simulation; details of the other activities can be found in the gPROMS Advanced User Guide.

  … *Parameter value setting* …

```
ASSIGN
```

  … *Degrees of freedom assignment* …

```
INITIAL
```

  … *Initial conditions specifications* …

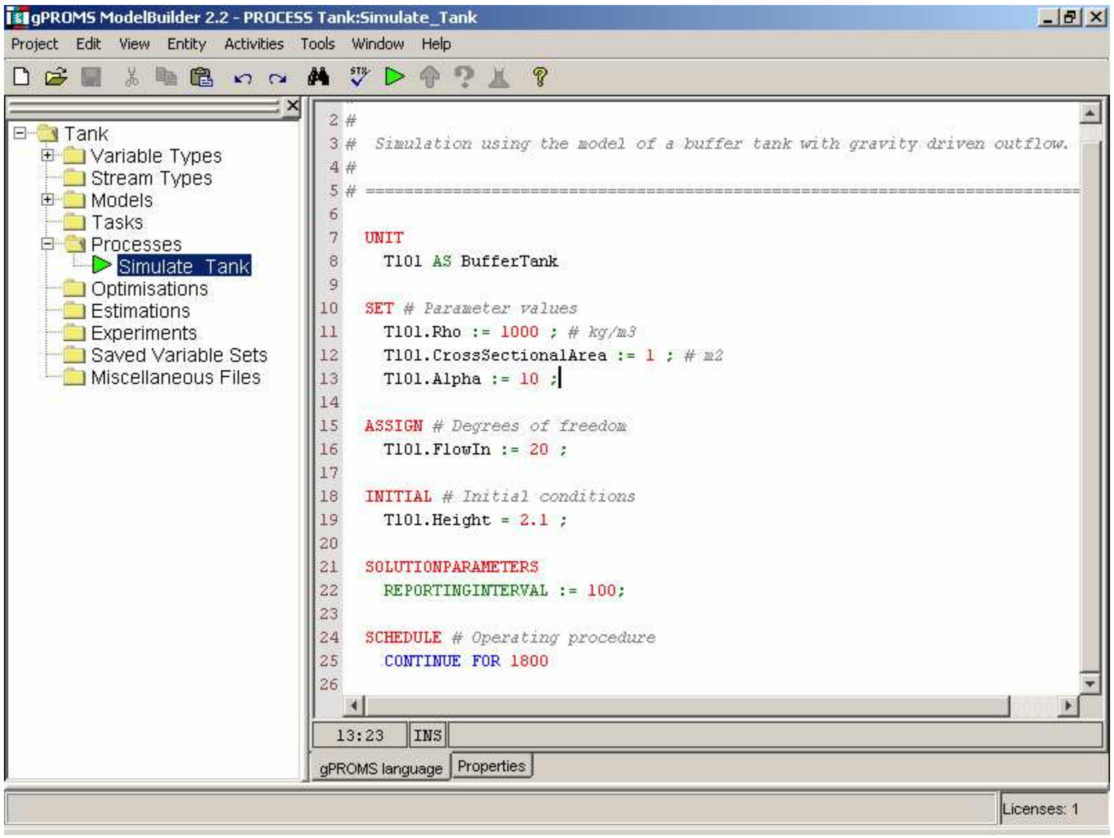
```
SOLUTIONPARAMETERS
```

  … *Model-based activities specifications* …

```
SCHEDULE
```

  … *Operating policy specifications* …

The entire `PROCESS` (named `SimulateTank`) for a dynamic simulation activity involving the buffer tank process is shown in figure 2.12. Next, we take a more detailed look at each of the sections of this `PROCESS` in sequence.



```
#
#  Simulation using the model of a buffer tank with gravity driven outflow.
#
# ==========================================================================

UNIT
  T101 AS BufferTank

SET # Parameter values
  T101.Rho := 1000 ; # kg/m3
  T101.CrossSectionalArea := 1 ; # m2
  T101.Alpha := 10 ;

ASSIGN # Degrees of freedom
  T101.FlowIn := 20 ;

INITIAL # Initial conditions
  T101.Height = 2.1 ;

SOLUTIONPARAMETERS
  REPORTINGINTERVAL := 100;

SCHEDULE # Operating procedure
  CONTINUE FOR 1800
```

Figure 2.12: An example `PROCESS` for the buffer tank.

2.2.5.1 The UNIT section

The first item of information required to set up a dynamic simulation activity is the process equipment under investigation. This is declared in the `UNIT` section of a `PROCESS`.

Equipment items are declared as instances of `MODEL`s. For example,

```
UNIT
  T101 AS BufferTank
```

creates an instance of `MODEL BufferTank`, named `T101`. `T101` is described by the variables declared within the `BufferTank MODEL` and its time-dependent behaviour is partially determined by the corresponding equations.

2.2.5.2 The SET section

Before an instance of a `MODEL` can actually be used in a simulation, all its parameters must be assigned appropriate values (unless they have been given default values). This is done in the `SET` section of a `PROCESS`[8].

For example,

```
SET
  T101.Rho                  := 1000 ;  # kg/m3
  T101.CrossSectionalArea   := 1    ;  # m2
  T101.Alpha                := 10   ;
```

sets the parameters of `T101` to appropriate values. Note that:

- in order to refer to parameter `Rho` of instance `T101` of `MODEL BufferTank`, we use the "pathname notation" `T101.Rho`;
- parameter values are set using the assignment operator (`:=`). In other words, the arithmetic expression appearing on the right hand side is first evaluated; its value is then given to the parameter appearing on the left hand side. This is another general rule of the gPROMS language:

[8]The assignment of parameter values can also be performed within `MODEL`s, using a `SET` section that is completely equivalent to the one described here. However, it is generally advisable that parameters be set at the `PROCESS` level. This practice maximises the reusability of the underlying `MODEL`s and minimises the probability of error. See section 6.5 for more details on this subject.

A General Rule of the gPROMS Language

gPROMS always uses the symbol `:=` to *assign* a value or expression appearing on the right hand side to the *single* identifier appearing on the left hand side. gPROMS always uses the symbol `=` to declare the *equality* of the two general expressions appearing on either side of this symbol.

2.2.5.3 The `ASSIGN` section

The set of equations resulting from the instantiation of `MODEL`s declared in the `UNIT` section is typically under-determined. This simply means that there are more variables than equations. The number of *degrees of freedom* in the simulation activity is given by:

Number of degrees of freedom (N_{DOF}) = Number of variables - Number of equations.

For the simulation activity to be fully defined, N_{DOF} variables must be specified as either constant values or given functions of time. Variables specified in this way are the input variables (or "inputs") of this simulation activity. The remainder of the variables are the *unknown* variables, the time variation of which will be determined by the solution of the system equations. Clearly, the number of unknowns is equal to the number of available equations - we therefore have a "square" system of equations.

The specification of input variables is provided in the `ASSIGN` section of the `PROCESS`. For instance,

```
ASSIGN
  T101.Fin := 20 ;
```

designates the inlet flowrate as an input and assigns it a constant value of 20. Again, in order to emphasise the assignment form of these specifications, input specifications use the assignment operator (`:=`).

2.2.5.4 The `INITIAL` section

Before dynamic simulation can commence, consistent values for the system variables at $t = 0$ must be determined. To this end, a number of additional specifications are needed. These augment the system of equations that describe the behaviour of the system and result in a square system of equations at $t = 0$. The solution of the latter provides the condition of the system at $t = 0$.

Traditionally, the term "initial condition" refers to a set of values for the differential variables at $t = 0$. However, gPROMS follows a more general approach where the initial conditions are regarded as additional equations that hold at $t = 0$ and can take any form. This, of course, allows for the traditional specification of "initial values" for the differential variables or, indeed, for any appropriate subset of system variables; however, it also makes possible the specification of much more general initial conditions as equations of arbitrary complexity.

The `INITIAL` section is used to declare the initial condition information pertaining to a dynamic simulation activity. For instance,

```
INITIAL
  T101.Height = 2.1 ;
```

specifies an initial condition for the buffer tank system by stating that the height of liquid in the tank at $t = 0$ is 2.1m. Note that, in contrast to the `SET` and `ASSIGN` sections, the equality operator (`=`) is used here to emphasise the fact that initial conditions are general equations.

An initial condition that is frequently employed for the dynamic simulation of process systems is the assumption of steady-state, constraining the time derivatives of the differential variables to zero. In gPROMS, this can be achieved by manually specifying all derivatives to be zero; *e.g.*:

```
INITIAL
  $T101.Holdup = 0 ;
```

However, this would be tedious for models with large numbers of differential variables, so the keyword `STEADY_STATE` may be utilised to specify this initial condition, as shown below:

```
INITIAL
  STEADY_STATE
```

In this latter case, no further specifications are required.

2.2.5.5 The `SOLUTIONPARAMETERS` section

The user also has the option to control various aspects of model-based activities carried out in gPROMS such as solver settings and output specifications. The `SOLUTIONPARAMETERS` section is used for this purpose. Detailed information regarding this topic will be covered in more detail in Chapter 10.

For example,

```
SOLUTIONPARAMETERS
  REPORTINGINTERVAL := 60;
```

The `REPORTINGINTERVAL` is the interval at which result values will be collected during the dynamic simulation (note that it does **not** effect the accuracy of the subsequent integration in any way). For this example, an interval of 60 is a reasonable choice.

The user does not need give any settings in this section. In such a case the user will be prompted to enter a `REPORTINGINTERVAL` in a dialog box.

2.2.5.6 The `SCHEDULE` section

The information that we have provided so far defines the condition of the system at the *start* of the simulation, which by convention corresponds to the time $t = 0$. Of course, *during* its operation (*i.e.* for $t > 0$), the system may be subjected to externally imposed manipulations, such as deliberate control actions and/or disturbances - indeed, the main motivation for performing a simulation is usually to understand how the system behaves under these manipulations.

Information on the external manipulations that are to be simulated is provided in the `SCHEDULE` section of the `PROCESS`. For the purposes of this chapter, we restrict our attention to the simplest possible case, allowing the system to operate without any external disturbance over a specified period of time. This is achieved via the:

`CONTINUE FOR` *TimePeriod*

construct in the `SCHEDULE` section of the `PROCESS`.

gPROMS can be used to simulate much more complex cases, including detailed operating procedures of entire plants. This is discussed in Chapters 7 and 8.

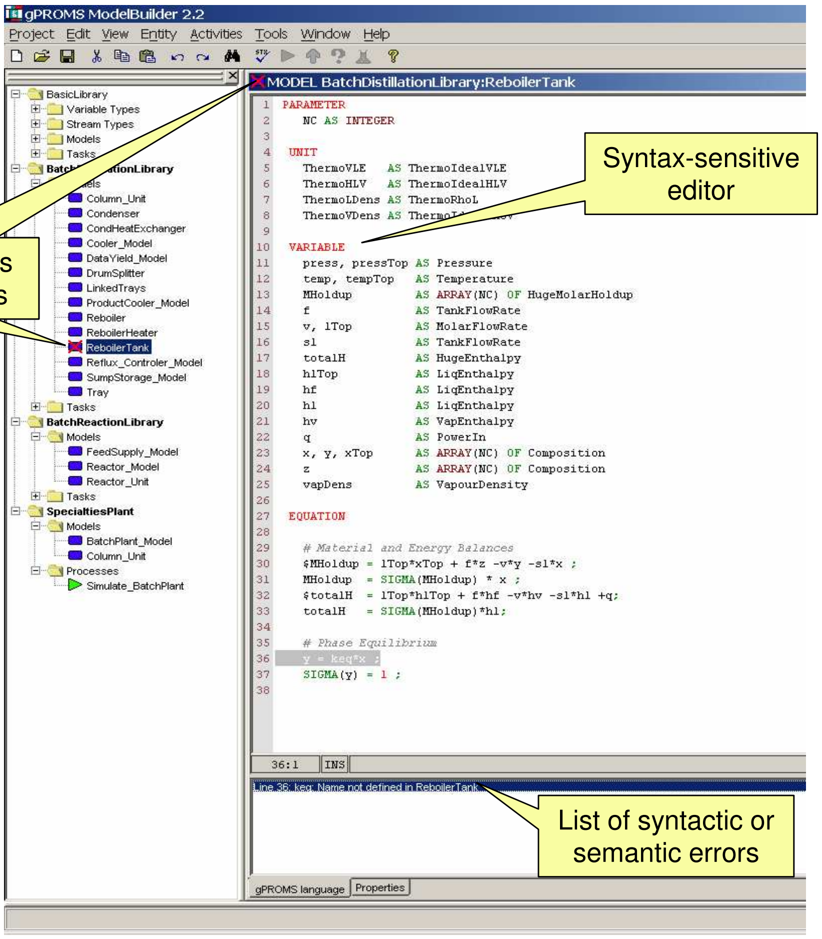
2.2.6 Syntax checking

gPROMS ModelBuilder will automatically check the syntax of the gPROMS language in any of the Entities that you have written. In addition, gPROMS will check for unidentified local variables (local semantic checking).

You can ask gPROMS to check the syntax by a number of different methods:

1. By saving the Project.
2. By clicking on the check syntax button just under Tools on the top toolbar.
3. By selecting Check Syntax from the Entity menu.
4. Right clicking on the Entity and selecting Check syntax.
5. By using the keyboard short-cut (F4).

If ModelBuilder finds an error, as shown in figure 2.13, a small box appears just underneath the text editor reporting the error. Double-clicking on the error message in the



```
PARAMETER
   NC AS INTEGER

 UNIT
   ThermoVLE   AS ThermoIdealVLE
   ThermoHLV   AS ThermoIdealHLV
   ThermoLDens AS ThermoRhoL
   ThermoVDens AS ThermoI

 VARIABLE
   press, pressTop AS Pressure
   temp, tempTop   AS Temperature
   MHoldup         AS ARRAY(NC) OF HugeMolarHoldup
   f               AS TankFlowRate
   v, lTop         AS MolarFlowRate
   sl              AS TankFlowRate
   totalH          AS HugeEnthalpy
   hlTop           AS LiqEnthalpy
   hf              AS LiqEnthalpy
   hl              AS LiqEnthalpy
   hv              AS VapEnthalpy
   q               AS PowerIn
   x, y, xTop      AS ARRAY(NC) OF Composition
   z               AS ARRAY(NC) OF Composition
   vapDens         AS VapourDensity

 EQUATION

   # Material and Energy Balances
   $MHoldup = lTop*xTop + f*z -v*y -sl*x ;
   MHoldup  = SIGMA(MHoldup) * x ;
   $totalH  = lTop*hlTop + f*hf -v*hv -sl*hl +q;
   totalH   = SIGMA(MHoldup)*hl;

   # Phase Equilibrium
   y = keq*x ;
   SIGMA(y) = 1 ;
```

Line 36: keq: Name not defined in ReboilerTank

Figure 2.13: Syntax and semantic checking

error dialog box and gPROMS will automatically go to the corresponding line number to show where the syntax error is.

You will also see that the error is highlighted in the text editor window and that a red cross appears through the icon for the Entity in the Project tree. Correcting this error will then cause this cross and the error dialog box to disappear.

In addition to syntax checking, gPROMS Model Builder has a range of powerful features aimed at improving your productivity. These include search-and-replace tools, Project and Entity comparison capabilities, and the ability to develop library projects. For further details on these and other features - please refer to the ModelBuilder User Guide.

2.3 Running a gPROMS simulation activity

Sections 2.2.3, 2.2.4 and 2.2.5 have presented different aspects of the description of a gPROMS simulation activity in terms of the `MODEL`, `VARIABLE TYPE`s and `PROCESS` sections respectively. These are the groups in the gPROMS Project tree for which Entities must be defined in order for a simulation to be executed. Entities do not necessarily need to be defined for the other groups within the Project tree (`Stream Types` are described in chapter 6, `Saved Variable Sets` in chapter 7, `Tasks` in chapter 8, `Optimisations`, `Parameter Estimations` and `Experiments` are explained in the gPROMS Advanced User Guide).

Once you have prepared the `MODEL`, `VARIABLE TYPE`s and `PROCESS` shown in figures 2.10, 2.11 and 2.12, you are ready to execute a simulation.

For each model based activity that you execute, such as a simulation, a new Case is created[9].

2.3.1 Cases

Briefly, a Case is a combined record of all the input information that defines a model-based activity and the results generated by the execution of this activity, as well as any diagnostic messages that may have been issued during its execution. The intention is that a Case may serve as a permanent record of a particular model-based activity that can be archived for future reference, thus providing auditability and traceability of model-based decisions.

Cases are described in more detail in the ModelBuilder User Guide.

2.3.2 Executing a simulation

To execute a simulation first select the `PROCESS` and then choose one of the following options:

1. Go to the `Activities` pull down menu and select `Simulate...` ;
2. Click on the "green play" button underneath the `Tools` pull down menu ;
3. Right click on this `PROCESS` and then left click on `Simulate...`;
4. Press the `F5` or `Alt-S` keys on the keyboard.

Provided that there are no syntax errors and that all the entities that are referred to have been defined, an execution control dialog appears as shown in figure 2.14. The

[9]However, it is possible, if desired, to configure ModelBuilder to delete previous cases automatically before running a new simulation - refer to the ModelBuilder User Guide.

execution control dialog allows the user to configure various aspects of the Case - this is discussed further in the ModelBuilder User Guide - such as the name and contents of the case. It also allows the user whether the licence required by the model-based activity should be retained at the end of the execution[10]

In this case, just select “OK” in the execution control dialog.

ModelBuilder creates the Case. Just like a Project, a Case appears as a sub-tree of the ModelBuilder's navigation tree (see figure 2.15). However, unlike most Projects (also see the ModelBuilder User Guide), all entries in a Case are read-only, and this is indicated by a lock symbol annotating each entry in the Case sub-tree.

2.3.3 Execution diagnostics

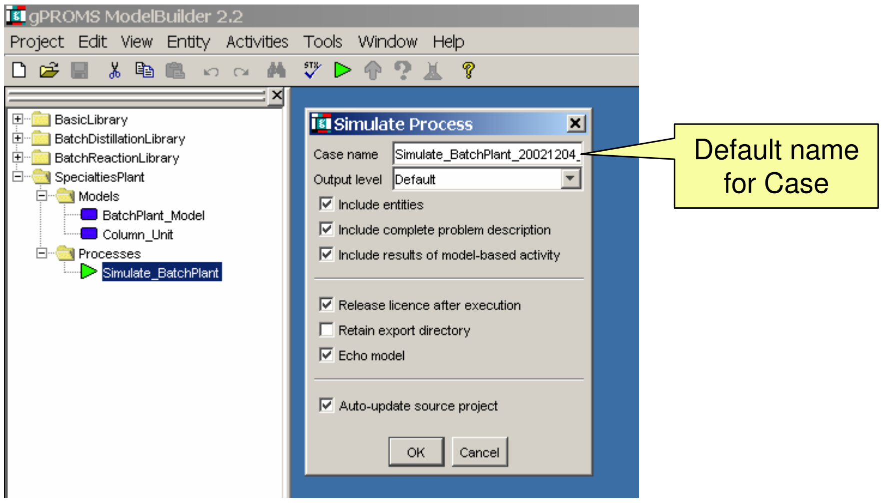
Once the necessary licence is obtained, ModelBuilder also creates an execution diagnostics window which displays all the messages relating to the solution of the model-based activity[11].

At this point, the simulation should proceed as outlined below. Note that whenever a `PROCESS` is executed, gPROMS analyses the mathematical models in the gPROMS Project so as to assist the user in identifying structural problems and errors in the modelling and/or the problem specification. Further details of these diagnostic features can be found in Appendix A.

```
     gPROMS (TM) - Version 2.2  for Windows XP Service Pack 1  Jan  7 2003
     general PROcess Modelling System

     Copyright (c) 1997-2003 Process Systems Enterprise Limited
     gPROMS and ModelEnterprise are trademarks of Process Systems
     Enterprise Limited. All rights reserved.
     No part of this material may be copied, distributed, published,
     retransmitted or modified in any way without the prior written
     consent by Process Systems Enterprise Limited.

Translating file : Simulate_Tank...

gPROMS translation initialisation took 0 seconds.

 .. MODEL BUFFERTANK
 .. PROCESS SIMULATE_TANK

gPROMS translation took 0 seconds.
```

[10]Retaining the licence allows some interaction with the model at the end of the execution, see the ModelBuilder User Guide. The licence can always be released manually at any time.

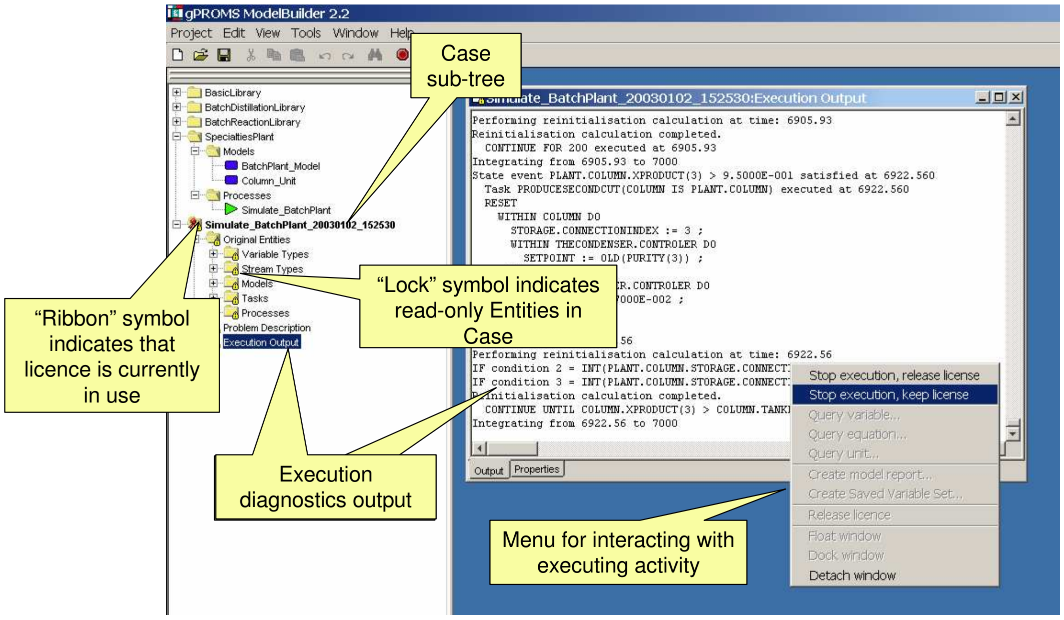
[11]The execution diagnostics window is associated with the CASE and can be referred back to even when the simulation has finished.



Case name: Simulate_BatchPlant_20021204
Output level: Default

- [✓] Include entities
- [✓] Include complete problem description
- [✓] Include results of model-based activity
- [✓] Release licence after execution
- [ ] Retain export directory
- [✓] Echo model
- [✓] Auto-update source project

Figure 2.14: The execution control dialog



```
Performing reinitialisation calculation at time: 6905.93
Reinitialisation calculation completed.
  CONTINUE FOR 200 executed at 6905.93
Integrating from 6905.93 to 7000
State event PLANT.COLUMN.XPRODUCT(3) > 9.5000E-001 satisfied at 6922.560
  Task PRODUCESECONDCUT(COLUMN IS PLANT.COLUMN) executed at 6922.560
  RESET
    WITHIN COLUMN DO
      STORAGE.CONNECTIONINDEX := 3 ;
      WITHIN THECONDENSER.CONTROLER DO
        SETPOINT := OLD(PURITY(3)) ;

                 R.CONTROLER DO
                 7000E-002 ;


                 56
Performing reinitialisation calculation at time: 6922.56
IF condition 2 = INT(PLANT.COLUMN.STORAGE.CONNECT
IF condition 3 = INT(PLANT.COLUMN.STORAGE.CONNECT
Reinitialisation calculation completed.
  CONTINUE UNTIL COLUMN.XPRODUCT(3) > COLUMN.TANK
Integrating from 6922.56 to 7000
```

Figure 2.15: Cases and activity execution

```
The following processes are available:

  SIMULATE_TANK

Executing process SIMULATE_TANK...
All 4 variables will be monitored during this simulation!
Building mathematical problem description took 0 seconds.
Loaded 'gRMS.dll'.
gRMS output channel: Using version 2.1.15 compiled on Jan  6 2003.
gRMS output channel: Connecting to host localhost, port 3345.
Loaded 'DASOLV.dll'.
Loaded 'NLSOL.dll'.
Loaded 'MA48.dll'.
Loaded 'MA48.dll'.
Loaded 'NLSOL.dll'.
Loaded 'MA48.dll'.
Simulation will proceed with the following configuration:
            DASolver := "DASOLV" [
 "AbsolutePerturbationFactor" :=  1e-007,
       "AbsoluteTolerance" :=  1e-005,
                    "Diag" :=  FALSE,
           "EffectiveZero" :=  1e-005,
          "EventTolerance" :=  1e-005,
          "FDPerturbation" :=  1e-006,
       "FiniteDifferences" :=  FALSE,
             "OutputLevel" :=  0,
 "RelativePerturbationFactor" :=  0.0001,
       "RelativeTolerance" :=  1e-005,
                  "SenErr" :=  FALSE,
  "InitialisationNLSolver" :=  "NLSOL" [
          "ConvergenceTolerance" :=  1e-005,
                 "EffectiveZero" :=  1e-005,
                "FDPerturbation" :=  1e-005,
                      "MaxFuncs" :=  1000000,
              "MaxIterNoImprove" :=  10,
                 "MaxIterations" :=  1000,
               "NStepReductions" :=  10,
                   "OutputLevel" :=  0,
                     "SLRFactor" :=  50,
                "SingPertFactor" :=  0.01,
        "UseBlockDecomposition" :=  TRUE,
                      "LASolver" :=  "MA48" [
                           "BLASLevel" :=  32,
                     "ExpansionFactor" :=  5,
                    "FullSwitchFactor" :=  0.5,
                            "MinBlock" :=  1,
                         "OutputLevel" :=  0,
                    "PivotSearchDepth" :=  3,
                "PivotStabilityFactor" :=  0.1
                                            ]
                                       ],
                "LASolver" :=  "MA48" [
                     "BLASLevel" :=  32,
               "ExpansionFactor" :=  5,
              "FullSwitchFactor" :=  0.5,
                      "MinBlock" :=  1,
                   "OutputLevel" :=  0,
              "PivotSearchDepth" :=  3,
          "PivotStabilityFactor" :=  0.1
                                      ],
 "ReinitialisationNLSolver" :=  "NLSOL" [
          "ConvergenceTolerance" :=  1e-005,
                 "EffectiveZero" :=  1e-005,
                "FDPerturbation" :=  1e-005,
                      "MaxFuncs" :=  1000000,
              "MaxIterNoImprove" :=  10,
                 "MaxIterations" :=  1000,
               "NStepReductions" :=  10,
                   "OutputLevel" :=  0,
                     "SLRFactor" :=  50,
                "SingPertFactor" :=  0.01,
         "UseBlockDecomposition" :=  TRUE,
                      "LASolver" :=  "MA48" [
                           "BLASLevel" :=  32,
                     "ExpansionFactor" :=  5,
                    "FullSwitchFactor" :=  0.5,
                            "MinBlock" :=  1,
                         "OutputLevel" :=  0,
                    "PivotSearchDepth" :=  3,
                "PivotStabilityFactor" :=  0.1
                                            ]
                                       ]
                                  ]
Execution begins...
  Variables
       Known           : 1
       Unknown         : 3
          Differential  : 1
          Algebraic     : 2
   Model equations      : 3
   Initial conditions   : 1
Checking consistency of model equations and ASSIGN specifications...
OK!
Checking index of differential-algebraic equations (DAEs)...
OK!
Checking consistency of initial conditions...
OK!
Initialisation calculation completed.
  CONTINUE FOR 1800 executed at 0
Integrating from 0 to 100
Integrating from 100 to 200
Integrating from 200 to 300
Integrating from 300 to 400
Integrating from 400 to 500
Integrating from 500 to 600
Integrating from 600 to 700
Integrating from 700 to 800
Integrating from 800 to 900
Integrating from 900 to 1000
Integrating from 1000 to 1100
Integrating from 1100 to 1200
Integrating from 1200 to 1300
Integrating from 1300 to 1400
Integrating from 1400 to 1500
Integrating from 1500 to 1600
Integrating from 1600 to 1700
Integrating from 1700 to 1800
Integrating from 1800 to 1900
Time event occurs at 1800.000
Execution of SIMULATE_TANK completed successfully.
gPROMS simulation took 0 seconds.
 Total CPU Time: 0.07
  User CPU Time: 0.07
System CPU Time: 0
```

You may now modify the gPROMS Project if you wish, save it, and then `execute` the `PROCESS` again.

2.4 Displaying gPROMS output

You are now in a position to plot some of the simulation results - this is done using the gRMS ("gPROMS Results Management System") application.

2.4.1 On MS Windows workstations

(a) Select the gRMS window and create a new 2D Plot. There are two ways to do this:

- *either* left click on the `Graph` menu and select `New 2D Plot` (figure 2.16(a));
- *or* left click on the "2D" button (figure 2.16(b)).

In either case, an empty 2D Plot window will be created.

(b) Add a line to the plot. Again, there are two ways to do this:

- *either* left click on the `Line` menu and select `Add...` (figure 2.17(a));
- *or* left click on the button with a curve icon (figure 2.17(b)).

In either case, this will bring up an `Add Line Dialog` window (figure 2.18(a)).

(c) Double click on `SIMULATE_TANK`, then on `T101`. A list of variables for this unit will appear (figure 2.18(b)). Alternatively, you could click on the "+" symbol to expand the tree.

(d) Double click on variable `HOLDUP`, or left click on it and press the `Add` button. A `2D Line Properties Dialog` for this variable will appear (figure 2.19). The line will be added to the 2D Plot window in the background.

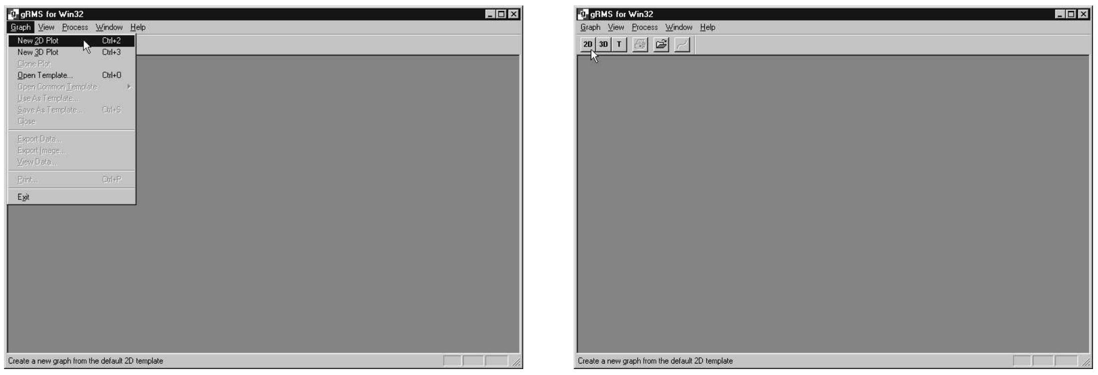
(e) Click `OK` on the `2D Line Properties Dialog` and click `Cancel` on the `Add Line Dialog` so that the graph can be seen clearly.

(f) To close the gRMS window, select the `Graph` menu and left click on `Exit`. You will be prompted to save the process if you have not already done so. Click `Yes` to bring up a save dialog before quitting, click `No` to quit without saving or click `Cancel` to continue using gRMS. See Appendix B for more on saving processes and graphs.

You will notice that the gRMS window remains on the screen even if you exit gPROMS. Normally, there is only one gRMS window on any given machine at any given time. Therefore, if you later re-enter gPROMS, the old gRMS window will be used. Similarly, a single gRMS window will handle results from two or more gPROMS sessions running simultaneously.

If you execute a number of `PROCESS`es, you will notice that gRMS records the results from each run under a separate "process" (see section B.1) even if two or more runs involve the execution of the same `PROCESS` in the input file. This allows you to use gRMS to compare results obtained from, say, different specifications of parameters, input variables or initial conditions by plotting variable trajectories arising from different executions on the same `PROCESS`.

A more detailed description of the gRMS utility can be found in Appendix B.

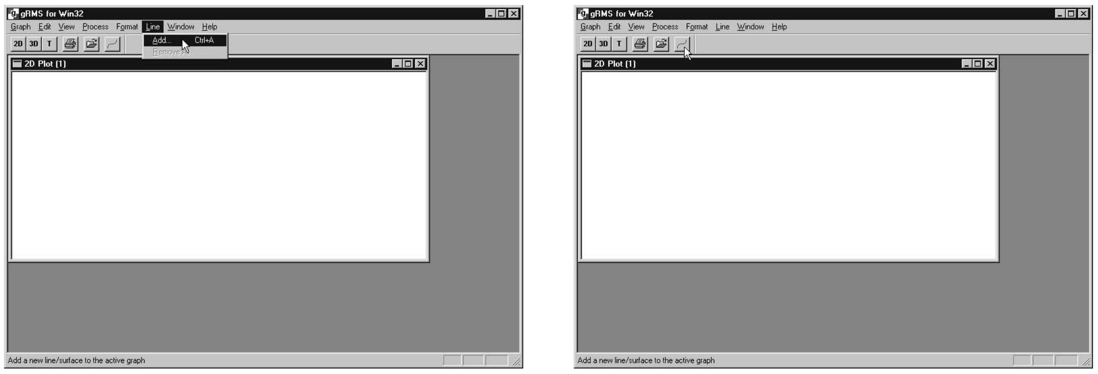


Create a new graph from the default 2D template

Create a new graph from the default 2D template

(a) Using the Graph menu. (b) Using the 2D Plot button

Figure 2.16: Adding a new 2D Plot window in gRMS (under Windows).

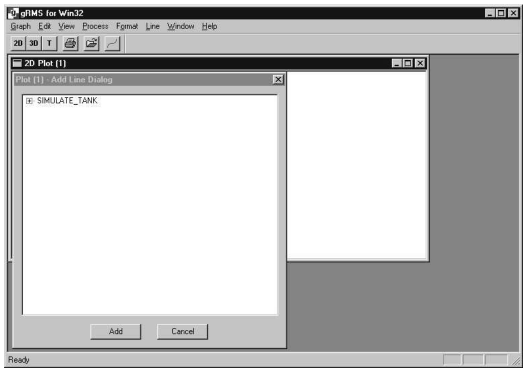


Add a new line/surface to the active graph

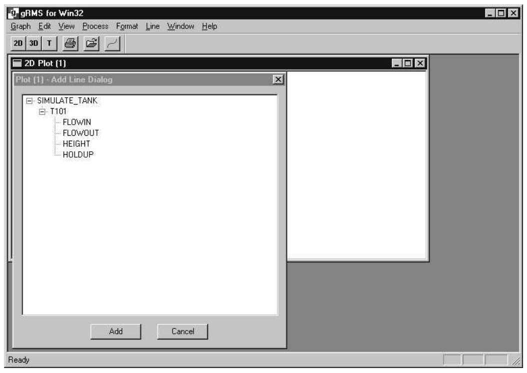
Add a new line/surface to the active graph

(a) Using the Line menu. (b) Using the add-line button

Figure 2.17: Adding a line to a plot in gRMS (under Windows).

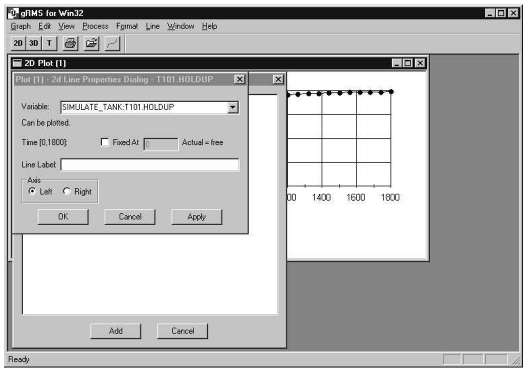

gRMS for Win32
Graph Edit View Process Format Line Window Help
2D Plot (1)
Plot (1) - Add Line Dialog
SIMULATE_TANK
Add
Cancel
Ready

(a) When first opened



(b) Expanded to show all variables

Figure 2.18: The Add Line Dialog.


gRMS for Win32
Graph Edit View Process Format Line Window Help
2D Plot (1)
Plot (1) - 2d Line Properties Dialog - T101.HOLDUP
OK
Cancel
Apply
Add
Cancel
Ready

Variable: SIMULATE_TANK:T101.HOLDUP
Can be plotted.
Time [0,1800]: ☐ Fixed At 0 Actual = free
Line Label:
Axis: (● Left) ○ Right

Figure 2.19: 2D Line Properties Dialog.

2.4.2 On UNIX workstations

(a) Go to the `gRMS` window and click on `Graph`.

(b) Select option `New 2D Plot`. A 2D plot window will appear (figure 2.20(a)).

(c) Click on `Lines` and choose `Add...` (figure 2.20(b)). An `Add Line Dialog` will appear (figure 2.20(c)).

(d) Double click on `SIMULATE_TANK:*`, then on `T101.*`. A list of variables for this unit will appear (figures 2.20(d) and 2.20(e)).

(e) Double click on variable `HOLDUP`. A `2D Line Properties Dialog` for this variable will appear (figure 2.20(f)).

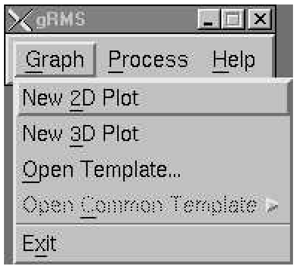
(f) Click `OK` on the `2D Line Properties Dialog` and click `Close` on the `Add Line Dialog`. A plot of variable `HOLDUP` against time will now appear in the plot window (figure 2.20(g)).

(g) To close the gRMS window, select the `Graph` menu and left click on `Exit`. Click on the `Yes` button to quit without saving (see Appendix B for instructions on saving processes and graphs).

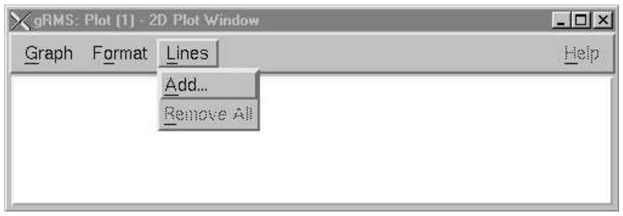
You will notice that the gRMS window remains on the screen even if you close the gPROMS execution window and the model building environment. Normally, there is only one gRMS window on any given machine at any given time. Therefore, if you later re-enter gPROMS, the old gRMS window will be used. Similarly, a single gRMS window will handle results from two or more gPROMS sessions running simultaneously in different command windows.

If you execute a number of `PROCESS`es, you will notice that gRMS records the results from each run under a separate "process" (see section B.1) even if two or more runs involve the execution of the same `PROCESS` in the input file. This allows you to use gRMS to compare results obtained from, say, different specifications of parameters, input variables or initial conditions by plotting variable trajectories arising from different executions on the same `PROCESS`.

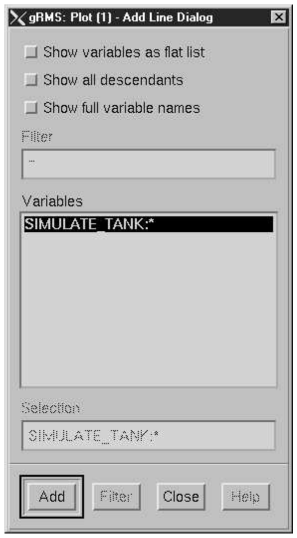
A more detailed description of the gRMS utility can be found in Appendix B.



(a)

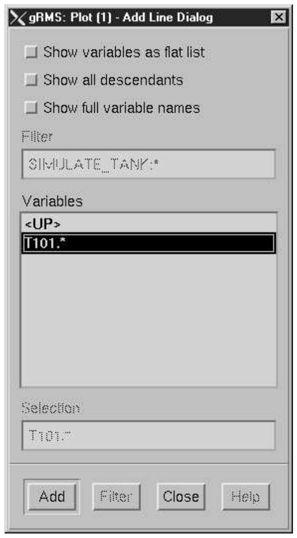


(b)



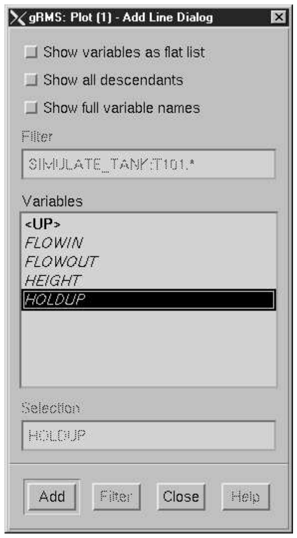
Show variables as flat list
Show all descendants
Show full variable names
Filter: *
Variables: SIMULATE_TANK:*
Selection: SIMULATE_TANK:*

(c)



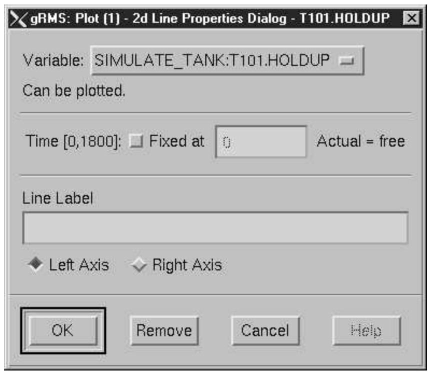
Show variables as flat list
Show all descendants
Show full variable names
Filter: SIMULATE_TANK:*
Variables
<UP>
T101.*
Selection: T101.*

(d)



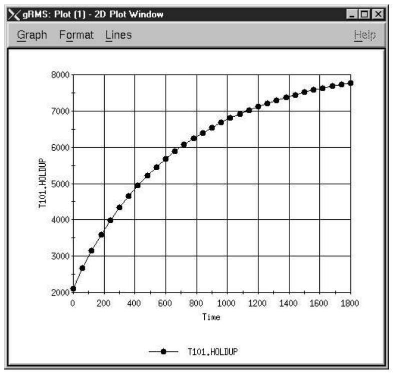
Show variables as flat list
Show all descendants
Show full variable names
Filter: SIMULATE_TANK:T101.*
Variables
<UP>
FLOWIN
FLOWOUT
HEIGHT
HOLDUP
Selection: HOLDUP

(e)



Variable: SIMULATE_TANK:T101.HOLDUP
Can be plotted.
Time [0,1800]: Fixed at 0 Actual = free
Line Label
Left Axis Right Axis

(f)



(g)

Figure 2.20: Using gRMS on UNIX to display simulation results.

Chapter 3

Arrays and Intrinsic Functions

Contents

In this chapter, we examine some of the more advanced mechanisms provided by the gPROMS language for the declaration of complex equation structures in `MODEL`s.

In many cases, a number of parameters, variables or equations that appear in a `MODEL` are closely related. Examples include:

- the stoichiometric coefficients, ν_{ij}, of a set of components $i = 1, .., NoComp$ participating in a set of reactions $j = 1, .., NoReact$;
- the concentrations, C_i, of components $i = 1, .., NoComp$ in a multi-component system;
- the equations expressing the conservation of components $i = 1, .., NoComp$ in a multi-component system.

In such cases, we need effective mechanisms for declaring and handling these entities as a group rather than individually. In a manner similar to most high-level programming languages, gPROMS achieves this aim via the use of *arrays*.

The next section of this chapter discusses how arrays of parameters and variables can be declared in `MODEL`s. Section 3.2 explains how these arrays can be used to construct array expressions, and then section 3.3 shows how the latter form the basis of the definintion of array equations in `MODEL`s.

The last section describes the intrinsic functions that are available in the gPROMS language, and the relations between these and the array concepts introduced earlier.

3.1 Declaring arrays of parameters and variables in MODELs

3.1.1 Arrays of parameters

As we have seen in section 2.2.3.3, model parameters are declared in the PARAMETER section of gPROMS MODELs to belong to the basic types INTEGER or REAL. Such parameters may be scalars or arrays of one, two or more dimensions.

Consider, for instance, the PARAMETER section in a model of a liquid-phase continuous stirred tank reactor (CSTR). This is shown in figure 3.1.

```
# MODEL LiquidPhaseCSTR

  PARAMETER

    # Number of components
    NoComp  AS INTEGER

    # Number of reactions
    NoReact AS INTEGER

    # Component molar densities
    Rho     AS ARRAY(NoComp) OF REAL

    # Stoichiometric coefficient of component i in reaction j
    Nu      AS ARRAY(NoComp,NoReact) OF REAL   DEFAULT 0

    # Order of component i in reaction j
    Order   AS ARRAY(NoComp,NoReact) OF REAL   DEFAULT 0
```

Figure 3.1: PARAMETER section of a liquid-phase CSTR MODEL

Here, NoComp and NoReact denote the numbers of chemical components and chemical reactions occurring in this system. Each of these is a simple (scalar) INTEGER parameter. On the other hand, the densities of the pure components[1] form a vector of real quantities declared as as an array of length NoComp:

```
Rho AS ARRAY(NoComp) OF REAL
```

Similarly, Nu and Order are two-dimensional arrays of REAL parameters. The number of elements in the first dimension is NoComp; the number of elements in the second dimension is NoReact. We note that, if a DEFAULT value is specified for an array parameter (*cf.* section 2.2.3.3), this is taken to refer to *all* elements of that array.

[1]For the purposes of this example, the pure component densities are assumed to be constant but different to each other.

3.1.2 Arrays of variables

Arrays of `MODEL` variables are declared in a manner very similar to that used for parameters (*cf.* section 3.1.1). For example, the `VARIABLE` section of the liquid-phase CSTR `MODEL` entity (*cf.* figure 3.1) is shown in figure 3.2.

```
# MODEL LiquidPhaseCSTR

  PARAMETER
     . . . . . . . . . . . . . . . . . . .

  VARIABLE

    # Input and output molar flowrates
    Flow_In, Flow_Out AS MolarFlowrate

    # Liquid phase volume
    V                 AS Volume

    # Component molar holdups
    HoldUp            AS ARRAY(NoComp) OF Moles

    # Input and output component mole fractions
    X_In, X_Out       AS ARRAY(NoComp) OF MoleFraction

    # Component concentrations
    C                 AS ARRAY(NoComp) OF Concentration

    # Reaction rates
    Rate              AS ARRAY(NoReact) OF ReactionRate
```

Figure 3.2: `VARIABLE` section of a liquid-phase CSTR `MODEL`

Arrays of parameters and variables may have any number of dimensions. The number of elements in each dimension is specified in terms of an integer expression (*e.g.* `HoldUp AS ARRAY(NoComp+1) OF REAL` is acceptable). The total number of elements in an array is the product of the number of elements in each dimension.

3.1.3 Rules for array declarations

There are some general rules that govern all arrays[2] in gPROMS:

[2]Including arrays of not only parameters and variables but also other entities such as streams and units: see chapter 6.

General Rules for Array Declarations in gPROMS

- Arrays can have any number of dimensions.
- The size of each dimension can be a general integer expression involving a combination of:
  - integer constants;
  - scalar integer `MODEL` parameters;
  - individual elements of arrays of integer `MODEL` parameters (see section 3.2.1 below); and
  - integer arithmetic operators[a].
- The index of each dimension ranges from 1 to the size of dimension.

[a]These include the usual arithmetic operators `+`, `-`, `*` and ˆ, as well as the integer division operator `DIV` and the division remainder operator `MOD`.

3.2 Using arrays of parameters and variables in expressions

Section 3.1 has described how arrays of parameters and variables can be declared in a `MODEL`. We now turn to consider how these arrays can be used in arithmetic expressions such as those appearing in `MODEL` equations.

First, we see how arrays may be referenced in arithmetic expressions. Then, we introduce the concept of an array equation.

3.2.1 General rules for referring to gPROMS arrays

The contents of an array may be referenced in several different ways[3]. These are explained below as well as being illustrated schematically in figure 3.3:

- Entire arrays can be referenced by using their names alone. For instance, `Rho` denotes the entire array of component molar densities.
- Individual elements can be referenced by using the name of the array and an index to the element in question enclosed in brackets. For one-dimensional arrays, this index should be an integer expression. For instance, the second element of array `Rho` is `Rho(2)` while the element of array `HoldUp` before the last is `HoldUp(NoComp-1)`. For multi-dimensional arrays, the index is a list of such expressions, one for each dimension. Thus, `Nu(2,4)` refers to the element on the second row and fourth column of array `Nu`.
- A subset of the elements in one or more dimensions can be referenced through the use of "slice" notation. For instance, `Holdup(2:4)` refers to the 2^{nd}, 3^{rd} and 4^{th} elements of array `Holdup`. `Nu(2:4,3:5)` refers to the slice of array `HoldUp` included between rows 2 to 4 and columns 3 to 5 (a 3×3 array in itself). Naturally, `Nu(1:NoComp,1:NoReac)` is equivalent to `Nu`. Similarly, `Nu(1:1,3:3)` is equivalent to `Nu(1,3)`.
- An entire dimension of an array can be referenced by leaving a blank. For instance, `Nu(2,)` refers to the entire second row of array `Nu`, while `Nu(,1:3)` refers to columns 1 to 3. Naturally, `Nu(,)` is equivalent to `Nu`.

3.2.2 Array expressions

A powerful concept in gPROMS is that of array expressions. Consider, for example, the algebraic expression:

```
x * y + w * z
```

If `x`, `y`, `w` and `z` are scalar variables, then the above also corresponds to a scalar. However, in gPROMS, the expression `x * y + w * z` is valid even if `x`, `y` and `z` are arrays provided they have the same dimensionality and size. For example, if we have the declarations:

[3]Although in thic chapter we are primarily interested in arrays of parameters and variables, the rules provided here actually apply to arrays of any gPROMS entity, *e.g.* streams and units (*cf.* chapter 6).

```
PARAMETER
    n, m      AS   INTEGER

VARIABLE
    x, y, z   AS ARRAY (n,m) OF SomeQuantity
    w         AS                SomeQuantity
```

then the expression `x * y + w * z` also represents a two-dimensional array of size $n \times m$, the $(i, j)^{th}$ element of which is equal to $x_{ij}y_{ij} + wz_{ij}$ for $i = 1, .., n$ and $j = 1, .., m$.

Although in the above examples, the gPROMS interpretation of the array expression coincided with the standard mathematical one, this is not always the case. For example, the expressions $x * y/z$ and $w/x + z\hat{\ }y$ are also valid in gPROMS, representing two-dimensional arrays of size $n \times m$, the $(i, j)^{th}$ elements of which are equal to $\frac{x_{ij}y_{ij}}{z_{ij}}$ and $w/x_{ij} + z_{ij}^{y_{ij}}$ respectively.

In general, consider an expression $x \otimes y$ where x and y are scalar or array expressions, and $\otimes$ is a binary arithmetic operator (+, -, *, /,^). This is a valid gPROMS expression if and only if it conforms to one of the four cases listed below:

Case	x	y	Dinensionality of x ⊗ y	Interpretation of x ⊗ y
1.	Scalar	Scalar	Scalar	xy
2.	Array	Scalar	Same as x	$x_{ijk...} \otimes y$
3.	Scalar	Array	Same as y	$x \otimes y_{ijk...}$
4.	Array	Array	Same as x *and* y	$x_{ijk...} \otimes y_{ijk...}$

Clearly, case 4 is valid only if both x and y have exactly the same dimensionality and size.

The above rules can be applied recursively to check the validity and to interpret expressions of arbitrary complexity. At the lowest level, `x` and `y` will be (scalar) constants, scalar parameters or variables, or arrays of parameters or variables, or slices of arrays of parameters or variables. For example, it can be verified that the following is a valid two-dimensional expression of size 3 $\times$ 2 :

```
3.23 / x(1:3, 4:5) * z(5:7, 1:2) + w(10:12, 2:3) + 4.13
```

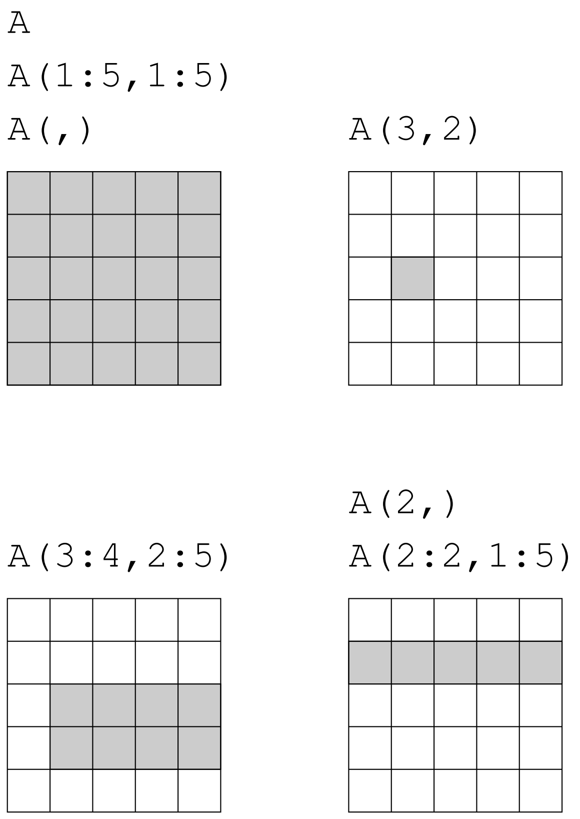


Figure 3.3: Referencing with arrays

3.3 Using arrays in equations

Elements of arrays (both parameters and variables) can be used in equations as if they were individual parameters or variables. For example, the equation that defines the concentration of component 2 in the liquid-phase CSTR can be written as follows:

```
HoldUp(2) = C(2) * V ;
```

However, arrays can be used more effectively to declare several equations simultaneously. This can be done in two different ways:

- *implicitly*, via the use of array expressions (*cf.* section 3.2.2),
- *explicitly*, via the `FOR` construct.

In the following two sections, we examine each of these mechanisms in detail.

3.3.1 Array equations

In section 3.2.2, we have seen how array expressions can be formed by combining arrays of parameters or variables, or elements or slices of these. By analogy, gPROMS allows the definition of array equations of the form:

Expression E = *Expression F* ;

that are valid provided they conform to one of the following four cases:

Case	E	F	Dimensionality of $E = F$	Interpretation of $E = F$
1.	Scalar	Scalar	Scalar	$E = F$
2.	Array	Scalar	Same as `x`	$E_{ijk...} = F$
3.	Scalar	Array	Same as `y`	$E = F_{ijk...}$
4.	Array	Array	Same as `x` *and* `y`	$E_{ijk...} = F_{ijk...}$

Thus, in view of the variable definitions shown in figure 3.2, the following is a valid equation:

```
HoldUp = C * V ;
```

gPROMS automatically expands such equations into an set of equations. For example, if `NoComp = 5`, the above will expand to:

```
HoldUp(1) = C(1) * V ;
HoldUp(2) = C(2) * V ;
  ...
HoldUp(5) = C(5) * V ;
```

3.3.2 The FOR construct

In section 3.3.1, we have seen how array equations can be written in an implicit manner by exploiting the array expression capability of gPROMS. An alternative is to write array equations explicitly using a FOR construct that is similar to that provided by most high-level programming languages.

Thus, consider the equation describing the conservation of component i in a multi-component buffer tank. This can be written mathematically as:

$$\frac{dM_i}{dt} = F^{in}x_i^{in} - F^{out}x_i, \quad i = 1, .., NoComp$$

In gPROMS, this can be written in two completely equivalent ways, namely implicitly, in the form (*cf.* section 3.3.1):

```
    $M = Fin*Xin - Fout*X ;
```

or explicitly, in the form:

```
  FOR i := 1 TO NoComp DO
    $M(i) = Fin*Xin(i) - Fout*X(i) ;
  END
```

The above are completely equivalent: which one you use depends entirely on your preference. However, situations do exist in which the required equations cannot be described via implicit declaration, and the use of explicit FOR constructs is essential. We will consider such situations in section 3.4.2.

The counter of a FOR construct (*e.g.* i in the above example) is an integer quantity that may be referenced only by equations enclosed within the construct. The range of this counter must be specified in terms of any arithmetic expressions involving integer constants, integer parameters and/or integer arithmetic operators. Moreover, a step increment may be specified. For instance,

```
  FOR i := NoComp+1 TO 2*NoComp STEP 2 DO
    ...
  END
```

will start by assign i a values of NoComp+1 and then will increment it by 2 until it exceeds 2*NoComp. If no increment is specified, its value defaults to 1.

A FOR construct may enclose an arbitrary number of equations of any type – including other FOR constructs. This allows nesting of FOR constructs to arbitrary depth; in such cases:

- each FOR construct must use a different name for its counter variable;
- any expression that appears within each FOR construct (including those defining its range and increment) may involve the counters of any enclosing FOR constructs.

3.4 Intrinsic gPROMS functions

Intrinsic gPROMS functions are used in equations to perform mathematical operations that would be difficult or even impossible to declare using normal language operators.

The gPROMS language contains two categories of intrinsic functions, as described below.

3.4.1 Vector intrinsic functions

All vector intrinsic functions have the following characteristics:

- they take a single argument representing a scalar or array expression;
- they return a result of dimensionality and size identical to those of their argument.

Table3.1 lists all vector functions that are recognised by gPROMS.

Identifier	Function
`ACOS`	The arccosine (in radians) of the argument
`ASIN`	The arcsine (in radians) of the argument
`ATAN`	The arctangent (in radians) of the argument
`ABS`	The absolute value of the argument
`COS`	The cosine of the argument (in radians)
`COSH`	The hyperbolic cosine of the argument
`EXP`	The exponential of the argument
`INT`	The largest integer that does not exceed the argument
`LOG`	The natural logarithm of the argument
`LOG10`	The logarithm to base 10 of the argument
`SGN`	The sign of the argument
`SIN`	The sine of the argument (in radians)
`SINH`	The hyperbolic sine of the argument
`SQRT`	The square root of the argument
`TAN`	The tangent of the argument (in radians)
`TANH`	The hyperbolic tangent of the argument

Table 3.1: Intrinsic vector functions

The result of each of the above functions is obtained by applying the corresponding operation to each element of the argument. For example, consider the declarations:

```
PARAMETER
    n, m      AS   INTEGER

VARIABLE
    x, y, z   AS ARRAY (n,m) OF SomeQuantity
    w         AS                SomeQuantity
```

Then, $x * SQRT(y + w)/SIN(z)$ is a valid expression (*cf.* section 3.2.2) representing an $n \times m$ array, the $(i, j)^{th}$ element of which is equal to:

$$\frac{x_{ij}\sqrt{y_{ij} + w}}{\sin z_{ij}}$$

for $i = 1, .., n$ and $j = 1, .., m$.

3.4.2 Scalar intrinsic functions

All scalar intrinsic functions have the following characteristics:

- they take an arbitrary number of arguments, each representing a scalar or array expression;
- they return a scalar result.

Table3.2 lists all scalar functions that are recognised by gPROMS.

Identifier	Function
`SIGMA`	The sum of all elements of all arguments
`PRODUCT`	The product of all elements of all arguments
`MIN`	The smallest of all elements of all arguments
`MAX`	The largest of all elements of all arguments

Table 3.2: Intrinsic scalar functions

The use of scalar intrinsic functions provides a powerful mechanism for writing complex mathematical expressions in gPROMS. However, some care is necessary in their use with array equations written using automatic expansion (*cf.* section 3.3.1). Consider, for instance, a mixing tank receiving a number of multi-component input streams. The conservation equation for component i can be written mathematically as:

$$\frac{dM_i}{dt} = \sum_{k=1}^{NoInput} F_k^{in} x_{k,i}^{in} - F_{out} x_i, \quad i = 1, .., NoComp$$

In gPROMS, this can be written as:

```
FOR i := 1 TO NoComp DO
  $M(i) = SIGMA(Fin*Xin(1:NoInput,i)) - Fout*X(i) ;
END
```

Note that the "alternative" formulation using automatic expansions:

```
  $M = SIGMA(Fin*Xin) - Fout*X ;
```

is actually incorrect since:

- the expression `Fin*Xin` violates the conformance rules set out in section 3.2.2 for array expressions;
- the expression `SIGMA(Fin*Xin)` is a scalar, not a vector of length `NoComp`.

A complete model for the mixing tank, illustrating many of the important points made in this chapter, is shown in figure 3.4.

```
# MODEL MixingTank

  PARAMETER
    NoComp, NoInput    AS INTEGER
    CrossSectionalArea AS REAL
    Rho                AS ARRAY(NoComp) OF REAL
    ValvePosition      AS REAL

  VARIABLE
    Fin                AS ARRAY(NoInput) OF Flowrate
    Xin                AS ARRAY(NoInput,NoComp) OF MassFraction
    Fout               AS Flowrate
    X                  AS ARRAY(NoComp) OF MassFraction
    M                  AS ARRAY(NoComp) OF Mass
    TotalHoldup        AS Mass
    TotalVolume        AS Volume
    Height             AS Length

  EQUATION

    # Mass balance
    FOR i := 1 TO NoComp DO
      $M(i) = SIGMA(Fin*Xin(,i)) - Fout*X(i) ;
    END

    # Mass fractions
    TotalHoldup = SIGMA(M) ;

    M = X * TotalHoldup ;

    # Calculation of liquid level from holdup
    TotalVolume = SIGMA(M/Density) ;

    TotalVolume = CrossSectionalArea * Height ;

    Fout = ValvePosition * SQRT ( Height ) ;
```

Figure 3.4: Multi-component mixing tank `MODEL` entity

As an additional example, figure 3.5 illustrates the use of nested `FOR` constructs to implement the matrix-matrix multiplication operation between matrices A $(n \times m)$ and B $(m \times q)$, resulting in a matrix C $(n \times q)$.

```
# MODEL MatrixMultiplication

  PARAMETER
     n, m, q    AS INTEGER

  VARIABLE
     A    AS ARRAY (n, m) OF SomeQuantity
     B    AS ARRAY (m, q) OF SomeQuantity
     C    AS ARRAY (n, q) OF SomeQuantity

  EQUATION
    FOR i := 1 TO n DO
      FOR j := 1 TO q DO
        C(i,j) = SIGMA(A(i,)*B(,j))
      END
    END
```

Figure 3.5: Matrix multiplication `MODEL` entity

Chapter 4

Conditional Equations

Contents

In section 2.2.3.5, we saw how to define simple equations in gPROMS `MODEL`s. Later, in section 3.3, we introduced the concept of array equations. All the equations that we have encountered so far are *continuous* in the sense that they involve the same expressions irrespective of the values of the variables that occur in them.

The physical behaviour of many process operations, however, are described in terms of *discontinuous* equations, the form of which depends on the current variable values and, in certain cases (*e.g.* involving hysteresis effects), also some aspects of the past history of the system.

This chapter describes the powerful facilities provided by gPROMS for the modelling of discontinuous physics. The next section introduces the concept of State-Transition Networks (STNs) that forms the basis of this modelling mechanisms. Section 4.2 presents the `CASE` construct which provides a direct description of general STNs in the gPROMS language. Finally, section 4.3 presents the `IF` construct which can be used to describe a special form of STNs that occur very frequently in practical applications.

We note that, in this chapter, we are interested in discontinuities that arise because of the intrinsic physical behaviour of the system and not as a result of external discrete actions imposed on the system by its environment or its operatots (*e.g.* the opening and closing of manual valves, the action of discrete controllers at the end of each sampling interval). The description of discontinuities of this latter kind in gPROMS will be considered in chapters 7 and 8.

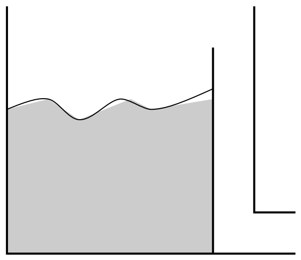


Figure 4.1: Vessel with overflow weir

4.1 State-Transition Networks

Discontinuities in the description of the physical behaviour of process systems may arise in different ways such as:

- transitions from laminar to turbulent flow;
- reversal of the direction of flow;
- appearance and disappearance of thermodynamic phases;
- equipment failure;

and many others.

State-Transition Networks (STNs) provide a general way of describing discontinuous systems. This concept is best introduced via an example. Consider the vessel depicted in figure 4.1. It is similar to the buffer tank of figure 2.3, the only difference being the presence of an overflow weir. When the level of liquid in the vessel, h, is below the height of the weir, h_w, no outflow is observed. When, on the other hand, the liquid level exceeds the weir height, the rate of outflow is assumed to be proportional to $h - h_\mathrm{w}$.

The mathematical model of the transient behaviour of this system can be written as follows:

Mass balance

$$\frac{dM}{dt} = F_\mathrm{in} - F_\mathrm{out} \tag{4.1}$$

Calculation of liquid level in the tank

$$M = \rho A h \tag{4.2}$$

Characterisation of the output flowrate

$$F_\mathrm{out} = \begin{cases} 0, & \text{if } h \le h_\mathrm{w} \\ \alpha(h - h_\mathrm{w}), & \text{if } h > h_w \end{cases} \tag{4.3}$$

We note that two different sets of equations are needed to describe the behaviour of this system depending on whether the level of the liquid is above or below the weir. Thus, the system may

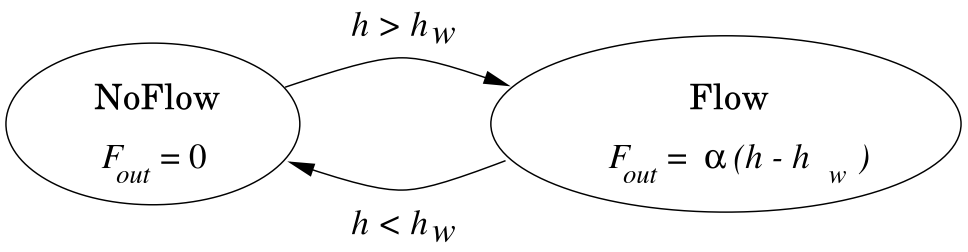


Figure 4.2: STN representation of vessel with overflow weir

exist in two distinct *states*, FLOW and NOFLOW, that correspond respectively to whether or not liquid flows over the weir. At any particular time, the system is in exactly one of these states. However, *transitions* from one state to the other will occur instantaneously if certain conditions are met. For example, if, while the system in in state NOFLOW, the height of the liquid exceeds that of the weir, the system will instantaneously jump to state FLOW. Conversely, if, while the system in in state FLOW, the height of the liquid drops below that of the weir, the system will instantaneously jump to state NOFLOW.

The above situation can be represented graphically in terms of an STN as shown in figure 4.2. The two circles (or ellipses) denote the two possible system states; for convenience, the form of the discontinuous equation 4.3 in each of these states is also shown within these circles. On the other hand, equations 4.1 and 4.2 do not appear in this figure as their form is independent of the state the system is in. The transitions between the two states are also shown in figure 4.2 as arrows connecting the corresponding circles. Again for convenience, each arrow is labelled with the logical condition that triggers the corresponding transition.

The STN of figure 4.2 represents a *reversible, symmetric discontinuity* because:

- the system may jump from either of the two states to the other and,
- the logical condition for one the two transitions is the exact negation of that for the other.

An example of a different type of discontinuity is shown in figure 4.3. Here, a vessel is fitted with a bursting disc. The disc can either be intact (with no gas flow from the vessel) or burst (with gas venting from the disc to the flare stack). This gives rise to two distinct system states (INTACT and BURST). As in the previous example, some of the equations that describe the system take a different form in each of these states while some others remain unchanged.

A transition from INTACT to BURST occurs when the pressure in the vessel rises above the set pressure and the disc shatters. The resulting outflow of gas will then cause an reduction of the pressure which, eventually, may drop below its set value. However, the system cannot return to its INTACT state once the disc has shattered[1]. Consequently, this is an example of an *irreversible discontinuity.*

A final example is that of a vessel fitted with a safety relief valve (see figure 4.4). The valve can be either open or closed, which again gives rise to two system states (OPEN and CLOSED). A transition from the CLOSED to the OPEN state occurs when the pressure in the vessel rises

[1]Unless, of course, it is repaired as a result of an external action – see section 7.1.2).

above the set pressure, while a transition from the Open to the Closed state occurs when the pressure falls below a (lower) reseat pressure. This is a *reversible, asymmetric discontinuity* because, although there are possible transitions in both directions, the two transition conditions are not the exact negation of each other.

We note that, in all three examples, only a subset of the model equations are directly related to the discontinuity and change from one state to another. The rest of the equations remain unchanged regardless of the state the system is in.

Summarising, a discontinuity in the physical behaviour of a system gives rise to a number of possible system states. Naturally, at any given time, the system can be in exactly one of these states. Some of the equations that determine the behaviour of the system hold irrespective of the system state. However, some others take a different form in each state. Transitions between the different states take place when certain logical conditions are satisfied.

A system may exhibit more than one physical discontinuity described by multiple STNs and/or more than two states within the same STN. For instance, a more detailed model of the weir vessel would seek to characterise the nature of the fluid flow in the outlet pipe. This would give rise to three system states, *i.e.* Laminar, Turbulent and Choked.

A complex STN for a hypothetical system is depicted in figure 4.5. Here, the system exhibits two separate physical discontinuities involving three and two possible states respectively.

- Equations 1, 2 and 3 remain unaffected by the discontinuities and are valid throughout.
- Equation 4 is affected by the first discontinuity and is written as 4a, 4b or 4c, depending on the state of the system.
- Equations 5 and 6 are affected by the second discontinuity and are written as 5a, 5b and 6a, 6b in each of the two states respectively.

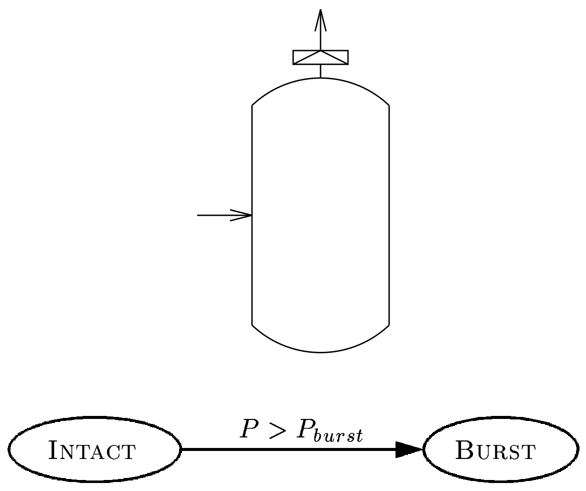


Figure 4.3: Vessel with bursting disc

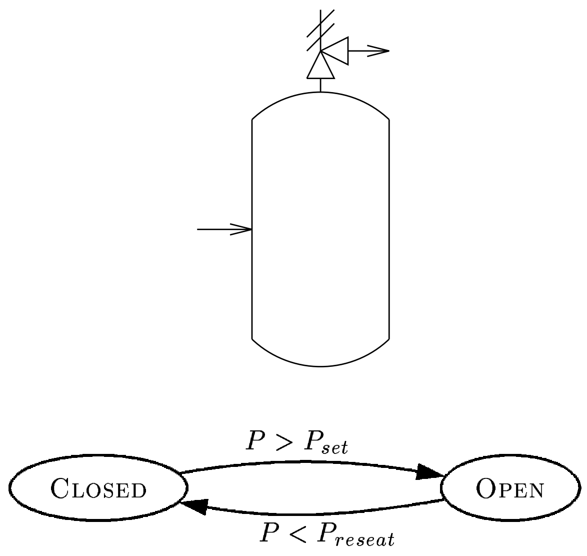


Figure 4.4: Vessel with safety relief valve

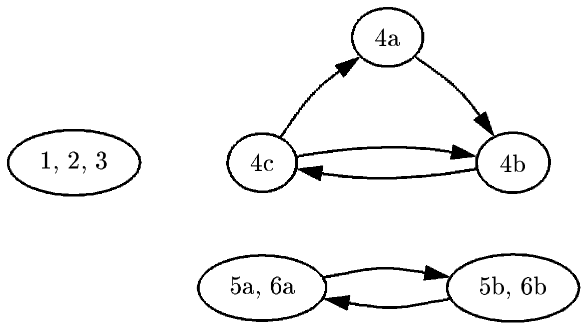


Figure 4.5: Hypothetical system model

4.2 The CASE conditional construct

The CASE construct permits the description of general STNs of the type discussed in the previous section within a MODEL.

4.2.1 An example of the use of CASE construct

Figure 4.6 illustrates the use of the CASE construct in a MODEL of a vessel with a bursting disc (*cf.* figure 4.3).

```
# MODEL VesselWithDisc

  PARAMETER
    R                     AS REAL  DEFAULT 8.314  # J/K.mol
    VesselVolume          AS REAL
    BurstPressure         AS REAL
    AtmPressure           AS REAL
    DiscConstant          AS REAL

  VARIABLE
    FlowIn, FlowOut       AS MolarFlowrate
    ReliefFlow            AS MolarFlowrate
    HoldUp                AS Moles
    T                     AS Temperature
    Pressure              AS Pressure

  SELECTOR
    DiscFlag              AS (Intact, Burst) DEFAULT Intact

  EQUATION

    # Mass balance
    $HoldUp = FlowIn - FlowOut - ReliefFlow ;

    # Ideal gas law
    Pressure * VesselVolume = Holdup * R * T ;

    CASE DiscFlag OF
      WHEN Intact : ReliefFlow = 0 ;
                    SWITCH TO Burst IF Pressure > BurstPressure ;
      WHEN Burst  : ReliefFlow = DiscConstant * SQRT(R*T) ;
    END # Case

    # Energy balance
    . . . . . . . . . . . . . . . . . . . . . .
```

Figure 4.6: MODEL entity for a vessel equipped with a bursting disc

We note that this MODEL presents two features that we encounter for the first time:

1. A SELECTOR section is used for the declaration of the system states that arise from the discontinuity:

   ```
   SELECTOR
     DiscFlag AS (Intact, Burst) DEFAULT Intact
   ```

   declares an enumerated ("selector") variable, DiscFlag that can take only two values, namely Intact or Burst, with the former being the default value at the start of the simulation (see below). The default specification is optional and may be omitted.

2. The MODEL equations include the CASE equation:

   ```
   CASE DiscFlag OF
     WHEN Intact : ReliefFlow = 0 ;
                   SWITCH TO Burst IF Pressure > BurstPressure ;
     WHEN Burst  : ReliefFlow = DiscConstant * SQRT(R*T) ;
   END # Case
   ```

   that defines

   - the equation(s) that hold in each state, and
   - the logical condition(s) that trigger transitions between states.

More precisely, the above CASE construct states that:

- When the system is in the Intact state, the relief flow is zero. The system remains in this state as long as the pressure in the vessel is lower than the bursting pressure of the disc. When it exceeds that limit, a transition to the Burst state is initiated.
- When the system is in the Burst state, the relief flow is calculated from a sonic flow relationship[2]. As this is an irreversible discontinuity, there is no transition going back to the Intact state.

4.2.2 General considerations in the use of CASE constructs

In general, a CASE equation comprises two or more clauses, one for each possible value of the corresponding SELECTOR variable. Each of the clauses comprises a list of equations, followed by an optional list of SWITCH statements that define transitions from the current clause to other clauses of the CASE equation. The list of equations may include any combination of simple, array and even conditional equations[3]. The latter feature allows nesting of conditional equations to arbitrary depth.

It is important, however, to observe the following restrictions:

[2]The form of the equation presented here is only for illustration purposes. A more detailed form would take account of various other effects, including the transition from sonic to sub-sonic flow as the pressure in the vessel decreases.

[3]Including not only CASE constructs but also IF constructs – see section 4.3.

Restrictions on the Form of CASE Constructs

- The number of equations in each clause of a CASE construct must be the same.
- Each SELECTOR variable may be used in *only one* CASE construct.

The reason for the first restriction is obvious if one considers that the number of variables in the MODEL remains unaffected by the occurrence of transitions. Consequently, any change in the number of equations would lead to an over- or under-specified problem. The second restriction is imposed to avoid inconsistencies that might arise from different CASE attempting to force the same SELECTOR variable to switch to different values at the same time.

4.2.3 Initial values of SELECTOR variables

Another important consideration concerning the use of CASE equations is that, in order for a simulation experiment to commence, the initial (*i.e.* at time $t = 0$) value of the corresponding SELECTOR variable has to be specified. Thus, in our example the user must specify whether or not the disc is initially intact or not. This information cannot be inferred automatically by gPROMS: the mere fact that the initial system pressure is below the bursting value does not necessarily mean that the disc is intact – it may well have burst as a result of earlier operation!

If a default value has been specified for the SELECTOR variable in the MODEL, then this will be used as its initial value for the simulation. More generally, this value can be set or overridden for individual simulation experiments. This is done in the SELECTOR section of the PROCESS section (*cf.* section 2.2.5). For instance:

```
UNIT T101 AS VesselWithDisc
...
SELECTOR
  T101.DiscFlag := T101.Burst ; # Disc is initially burst
INITIAL
  T101.Pressure - T101.BurstPressure = -1E5 ;
...
```

Thus, in this example, we specify the disc as being burst despite the fact that the initial pressure of the system is specified to be 1 bar below the bursting pressure. From the syntactical point of view, it is important to note the use of the full pathname T101.Burst to denote the value of the SELECTOR variable. A specification of the form:

```
UNIT T101 AS VesselWithDisc
 ...
SELECTOR
   T101.DiscFlag := Burst ; # Disc is initially burst
```

would be meaningless as the identifier Burst is not known directly to the PROCESS section – it is defined only within the context of MODEL VesselWithDisc of which T101 is an instance.

4.3 The IF conditional construct

As we have seen in section 4.2, CASE constructs provide a general way in which STNs of arbitrary complexity can be described in the gPROMS language. However, reversible and symmetric discontinuities are by the far the most commonly encountered discontinuities in industrial processing systems. Although such discontinuities can be declared using CASE constructs, gPROMS provides the alternative IF conditional construct specifically as a convenient shorthand for the declaration of this common type of discontinuity.

```
# MODEL VesselWithWeir

  PARAMETER
    Rho                  AS REAL
    MolecularWeight      AS REAL
    CrossSectionalArea   AS REAL
    WeirHeight           AS REAL
    WeirLength           AS REAL

  VARIABLE
    HoldUp               AS Mass
    FlowIn, FlowOut      AS MassFlowrate
    Height               AS Length

  EQUATION

    # Mass balance
    $HoldUp = FlowIn - FlowOut ;

    # Calculation of liquid level from holdup
    Holdup = CrossSectionalArea * Height * Rho ;

    IF Height > WeirHeight THEN
      # Francis formula for flow over a weir
      FlowOut = 1.84 * (Rho/MolecularWeight)
                  * WeirLength * ABS(Height-WeirHeight)^1.5 ;
    ELSE
      FlowOut = 0 ;
    END # If
```

Figure 4.7: MODEL entity for a vessel equipped with an overflow weir

Figure 4.7 demonstrates the use of an IF equation in the declaration of a MODEL for a vessel fitted with an overflow weir (*cf.* figure 4.1).

```
IF Height > WeirHeight THEN
  # Francis formula for flow over a weir
  FlowOut = 1.84 * (Rho/MolecularWeight)
              * WeirLength * ABS(Height-WeirHeight)^1.5 ;
ELSE
  FlowOut = 0 ;
END # If
```

A logical condition (in this case, `Liquid_Level > Weir_Height`) is used to choose between two clauses, each comprising a list of equations. If the logical condition is satisfied, the equations declared in the first clause are included in the system model, otherwise the equations of the second clause are included. As with `CASE` equations, the number of equations in each clause must be the same.

As `IF` equations are a special case of `CASE` equations, there is always a `CASE` equation that achieves the same result. For instance, the `IF` equation for the overflow weir is equivalent to the following `CASE` equation:

```
SELECTOR
  WeirFlag AS (Above, Below)
  ...
EQUATION
  ...
  CASE WeirFlag OF
    WHEN Above : FlowOut = 1.84 * (Rho/MolecularWeight)
                           * WeirLength * ABS(Height-WeirHeight)^1.5 ;
                 SWITCH TO Below IF Height < WeirHeight ;
    WHEN Below : FlowOut = 0 ;
                 SWITCH TO Above IF Height > WeirHeight ;
  END # Case
  ...
```

A subtle difference between `IF` and `CASE` equations is that, in `IF` equations, the initially active state of the system cannot be specified explicitly by the user (*cf.* section 4.2.3). Instead, it is determined automatically by the initialisation calculation, which ensures that the consistent initial values obtained satisfy both the logical condition *and* the equations in this state. However, the solution of non-linear systems involving such conditional equations is far from trivial. Moreover, it is possible that a valid solutions exists in either clause of an `IF` equation; in such cases, the solution found will depend on the initial guesses and the numerical method employed during the initialisation procedure. In view of these factors, it may sometimes be preferable to use a `CASE` equation instead, especially if the initial state of the system is known *a priori*.

Chapter 5

Distributed Models

Contents

A significant number of unit operations in chemical or biochemical processes take place in distributed systems in which properties vary with respect to one or more spatial dimensions as well as time. For instance, a tubular reactor is described in terms of parameters and variables that, in addition to time, depend on the axial and radial position within the reactor (*e.g.* $T(z, r, t)$ *etc.*). Other common examples of distributed unit operations include packed bed absorption, adsorption and distillation columns and chromatographic columns.

In other unit operations, material properties are characterised by probability density functions instead of single scalar values. Examples include crystallisation units and polymerisation reactors, in which the size of the crystals and the length of the polymer chains respectively are described in terms of distribution functions. The form of the latter may also vary with both time and spatial position.

In fact, most complex processes involve a combination of distributed and lumped unit operations. The equations that determine the behaviour of such unit operations are typically systems of integral, partial differential, ordinary differential and algebraic equations (IPDAEs).

This chapter discusses the mechanisms provided by the gPROMS language for the declaration of `MODEL`s that describe distributed unit operations.

The system shown in figure 5.1 will be used to illustrate the relevant features. It is a tubular reactor used to carry out a liquid-phase exothermic chemical reaction. The intensive properties of the fluid in the tube vary with both axial and radial position as well as with time. The reactor is surrounded by a well-mixed cooling jacket. Thus, the intensive properties of the cooling medium are assumed to be uniform throughout the jacket but may still vary with time.

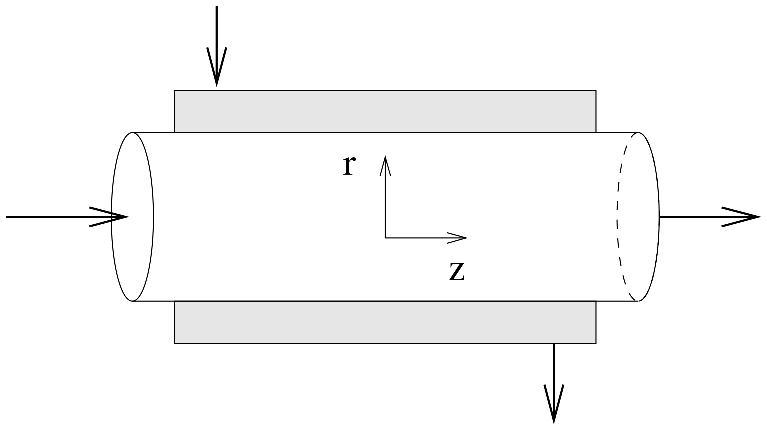


Figure 5.1: Tubular flow reactor

5.1 Declaring DISTRIBUTION_DOMAINs

The temperature in the reactor of figure 5.1 varies with time, axial and radial position ($T(z, r, t)$). As was mentioned in Chapter 2, all variables that are declared within a MODEL are automatically assumed to be functions of time. However, variations over other *distribution domains* (in this case the axial and radial domains, z and r respectively) have to be specified explicitly.

Distribution domains are declared in the DISTRIBUTION_DOMAIN section of a MODEL. Figure 5.2 shows how two such domains called Axial and Radial are declared in a MODEL of a tubular reactor. The extents of both domains are specified in terms of two model parameters, namely ReactorLength and ReactorRadius, respectively.

```
# MODEL TubularReactor

  PARAMETER

    # Number of components
    NoComp         AS INTEGER

    # Geometrical parameters
    ReactorRadius,
    ReactorLength  AS REAL

    # Transport properties

    # Axial and radial mass diffusivities
    Dz, Dr         AS REAL

    # Axial and radial thermal conductivities
    Kz, Kr         AS REAL

    # Reaction

    # Stoichiometric coefficients
    Nu                     AS ARRAY(NoComp) OF INTEGER

    ...

  DISTRIBUTION_DOMAIN
    Axial   AS   [ 0 : ReactorLength ]
    Radial AS    [ 0 : ReactorRadius ]

...
```

Figure 5.2: PARAMETER and DISTRIBUTION_DOMAIN sections for a MODEL of a tubular reactor

In general, the lower and upper bounds of the range of each distribution domain can be specified in terms of real expressions involving real constants and/or real parameters. Thus, the following are also valid distribution domain declarations:

```
# Normalised axial and radial domains
DISTRIBUTION_DOMAIN
  z AS [ 0 : 1 ]
  r AS [ 0 : 1 ]

# Axial domain encompassing the second half of the reactor
DISTRIBUTION_DOMAIN
  HalfAxial     AS [ ReactorLength/2 : ReactorLength ]
```

5.2 Declaring distributed VARIABLEs

A `MODEL` may involve variables with different degrees of distribution. For instance, in the tubular reactor example, the temperature of the fluid and the concentrations of the various chemical components within the tube are indeed functions of both the radial and axial positions. However, the wall temperature is a function only of axial position, while the temperature in the cooling jacket does not vary with spatial position at all[1].

```
# MODEL TubularReactor

  PARAMETER
      ...

  DISTRIBUTION_DOMAIN
      ...

  VARIABLE

    # Reactor temperature
    T   AS DISTRIBUTION(Axial,Radial) OF        Temperature

    # Concentrations
    C   AS DISTRIBUTION(NoComp,Axial,Radial) OF Concentration

    # Feed composition
    Cin AS ARRAY(NoComp) OF                     Concentration

    # Cooling jacket temperature
    Tc  AS                                      Temperature

    ...
```

Figure 5.3: `VARIABLE` section for a `MODEL` of a tubular reactor

- Variable `T`, which represents the temperature in the reactor, is declared as a `DISTRIBUTION` over the two continuous domains, `Axial` and `Radial`.
- Variable `C`, representing the concentrations of the various components in the reactor, is clearly an array of variables distributed over both the radial and axial domains. In gPROMS, an array of distributions is represented by adding one or more extra domains to a `DISTRIBUTION`. These domains are *discrete* in nature and they do not need to be declared explicitly in the `DISTRIBUTION_DOMAIN` section — a simple integer expression in the `VARIABLE` declaration suffices.

  For example, the concentration variable `C` ranges from `1` to `NoComp`, the number of components in the system), as well as over the two *continuous domains*, `Axial` and `Radial`.
- Variable `Cin`, representing the concentrations of the various components in the feed, is distributed over the discrete domain of components only. Although this could be written

[1]Of course, all of these variables are also functions of time, as is always the case with `VARIABLE`s in gPROMS.

as:

```
    Cin AS DISTRIBUTION(NoComp) OF Concentration
```

here we prefer to use the `ARRAY` concept as it is more natural.

- Variable `Tc`, representing the temperature in the cooling jacket, is solely a function of time.

In essence, then, `DISTRIBUTION` is a generalisation of the `ARRAY` concept that allows a `VARIABLE` to vary over both continuous and discrete domains. Conversely, `ARRAY`s is a special case of a `DISTRIBUTION` that is used to declare variables that are distributed over discrete domains only. Although `DISTRIBUTION`s can also be used for that purpose, the `ARRAY` construct is retained for compatibility with other programming languages.

Note that it is not permitted to declare a variable that is distributed over two domains of the same type. For instance, a temperature field over a square domain **cannot** be declared as:

```
T AS DISTRIBUTION(XDomain,XDomain) OF Temperature
```

The reason for this restriction is that a more complex syntax would then be required in order to distinguish between partial derivatives of the variable `T` with respect to the first and the second independent variables (see section 5.3.2).

This does not actually lead to any real restriction in functionality: variable `T` could easily be declared as:

```
T AS DISTRIBUTION(XDomain,YDomain) OF Temperature
```

where `XDomain` and `YDomain` are declared to be identical:

```
DISTRIBUTION_DOMAIN
  XDomain, YDomain AS [ 0.0 : Length ]
```

This also has the advantage of allowing different discretisation methods to be applied to each of the two domains (see section 5.4).

5.3 Defining distributed EQUATIONs

As with lumped models, the variables in MODELs are related through sets of equations that are declared in the EQUATION section. The following sections discuss these declarations, examining in detail a number of issues that are unique to distributed systems.

5.3.1 Distributed expressions

In sections 3.2.2 and 3.4, we saw how array expressions can be constructed by combining array variables using arithmetic operators and functions. In a very analogous manner, distributed expressions may be built from distributed variables. For example, consider the following declarations within a MODEL of a tubular reactor:

```
# MODEL TubularReactor

  PARAMETER

    # Geometrical parameters
    ReactorRadius,
    ReactorLength  AS REAL

    # Heat transfer parameters
    U, S           AS REAL

    ...

  DISTRIBUTION_DOMAIN
    Axial  AS   [ 0 : ReactorLength ]
    Radial AS   [ 0 : ReactorRadius ]

  VARIABLES
    Vz, Vr AS  DISTRIBUTION (Axial, Radial) OF Velocity
    T      AS  DISTRIBUTION (Axial, Radial) OF Temperature
    Twall  AS  DISTRIBUTION (Axial)         OF Temperature
    Tc     AS                                  Temperature
```

In this case, Vz * T is a valid expression that is distributed over the entire Axial and Radial domains. Similarly,

```
U * S * ( T(,ReactorRadius) - Tc )
```

is also a valid expression distributed over the entire Axial domain.

In some cases, it may be desired to define an expression over *part* of a particular domain. This can be achieved by using *slices* of distributions, very similar to the slice concept for arrays (*cf.* section 3.2.1). For example, the expression:

```
Vz(0:ReactorLength/2, ) * T(0:ReactorLength/2, )
```

is distributed over the first half of the `Axial` domain and the entire `Radial` domain.

The mathematical modelling of distributed systems often requires a rather subtle distinction between the entire domain *including* its boundaries, and the domain *excluding* all or part of its boundaries. In standard mathematical terminology, these two kinds of domain are referred to as "closed" and "open" respectively. One major reason for introducing this distinction is that some of the equations (*e.g.* conservation laws) may hold only in the interior of a domain while being replaced by appropriate boundary conditions (*cf.* section 5.3.5) on the domain boundaries.

To allow the modellers to make the above distinction, gPROMS employs the notation shown in table 5.1.

Mathematical notation	Interpretation	gPROMS notation
$[a, b]$	$z \in [a, b] \Leftrightarrow a \leq z \leq b$	`a   : b`
$(a, b]$	$z \in (a, b] \Leftrightarrow a < z \leq b$	`a|+ : b`
$[a, b)$	$z \in [a, b) \Leftrightarrow a \leq z < b$	`a   : b|-`
(a, b)	$z \in (a, b) \Leftrightarrow a < z < b$	`a|+ : b|-`

Table 5.1: Closed and open domain notation

Thus, the variable slice `Vz(0|+:ReactorLength, 0|+:ReactorRadius|-)` denotes the values `Vz`(z, r) for the values of z and r satisfying $0 < z \leq$ `ReactorLength` and $0 < r <$ `ReactorRadius`.

We conclude by formally defining the validity of expressions involving distributed variables. Consider an expression $x \otimes y$ where x and y are scalar or distributed expressions, and $\otimes$ is a binary arithmetic operator (+, -, *, /, ^). Then this is a valid gPROMS expression if and only if it conforms to one of the four cases listed below:

Case	x	y	Dimensionality of x ⊗ y	Interpretation of x ⊗ y
1.	Scalar	Scalar	Scalar	xy
2.	Array	Scalar	Same as x	$x(...) \otimes y$
3.	Scalar	Distribution	Same as y	$x \otimes y(...)$
4.	Distribution	Distribution	Same as x *and* y	$x(...) \otimes y(...)$

Clearly, case 4 is valid only if both x and y are distributed over exactly the same domains, also taking account of whether each of these is open or closed. For example, the following is a valid expression:

```
    Vz(0|+:ReactorLength,ReactorRadius|-) * Twall(0|+:ReactorLength)
```

that is distributed over the `Axial` domain which is open on (*i.e.* does not include) the left boundary ($z = 0$) but closed on (*i.e.* includes) the right boundary ($z =$`ReactorLength`). On the other hand, the expression:

```
Vz(0|+:ReactorLength,ReactorRadius) * T(0:ReactorLength,ReactorRadius)
```

is **invalid** because the first operand `Vz` is distributed over a domain that is open on the left and closed on the right, while the second operand `T` is distributed over a domain that is closed on both ends.

5.3.2 The `PARTIAL` operator

Partial differentiation of a distributed variable or expression with respect to a domain over which it is distributed is achieved with the `PARTIAL` operator. Its syntax is of the form:

`PARTIAL (` *Expression*`,` *DistributionDomain* `)`

where *Expression* is a general expression and *DistributionDomain* is one of the distribution domains in the system. Normally, at least one of the variables involved in the differentiated expression will be distributed over the specified domain, otherwise the result of the differentiation will be zero. `PARTIAL` operators may also be nested.

The result of a `PARTIAL` operator is generally a distributed expression that has exactly the same degree of distribution as the *Expression* being differentiated.

5.3.2.1 First order partial derivatives

Considering the example presented in section 5.3.1, the following are examples of valid first order partial derivative expressions:

$$\frac{\partial T}{\partial z}, \quad z \in [0, L], r \in [0, R]$$

```
PARTIAL(T,Axial)
```

$$\frac{\partial (V_z T)}{\partial r}, \quad z \in (0, L], r = R$$

```
PARTIAL(Vz(0|+:L, ReactorRadius)*
T(0|+:L,ReactorRadius),Radial)
```

$$\frac{\partial}{\partial r}\left(k_r \frac{\partial T}{\partial r}\right), \quad z = 0, r \in (0, R)$$

```
PARTIAL(Kr(0,0|+:ReactorRadius|-)*
PARTIAL(T(0,0|+:ReactorRadius|-),Radial),Radial)
```

Note that the partial differentiation operator with respect to time is denoted by symbol `$` rather than the operator `PARTIAL`. Thus:

$$\frac{\partial T}{\partial t}$$

```
$T
```

There are two reasons for this:

- The use of `$` is consistent with the time derivative operator in lumped systems (*cf.* section 2.2.3.5).
- The numerical solution methods in gPROMS treat the time domain quite differently to the explicitly declared distribution domains (see section 5.4).

5.3.2.2 Higher order partial derivatives

`PARTIAL` operators may be nested to express higher order derivatives as follows:

`PARTIAL(`*Expression*`,PARTIAL(`*Expression*`,`*DistributionDomain*`),`*DistributionDomain*`)`

Alternatively, the following abbreviated form may be used:

`PARTIAL (` *Expression*`,` *DistributionDomain*`,` *DistributionDomain* `)`

Here differentiation first takes place with respect to the first domain, then with respect to the second *etc.* For example:

$$\frac{\partial^2 T}{\partial z^2}, \quad z \in (0, L), r \in (0, R)$$

```
PARTIAL(T(0|+:ReactorLength|-,0|+:ReactorRadius|-,Axial,Axial)
```

5.3.3 The `INTEGRAL` operator

Integrals occur frequently in equations arising in a number of branches of physics and engineering. In process engineering applications, they often occur in population balance models describing, for instance, crystal size distributions, activity distributions of recycled catalyst particles, and the age and size distribution of microbiological cultures. They also appear when average values of distributed variables need be calculated.

Integration of a distributed variable with respect to a domain is achieved with the `INTEGRAL` operator. Its syntax is of the form:

`INTEGRAL (` *IntegralRange* `;` *Expression* `)`

where *Expression* is a general expression involving variables that are distributed over one or more distribution domains and *IntegralRange* represents the range of integration.

The result of a `INTEGRAL` operator is generally an expression that is distributed over one less domain than the *Expression* being integrated.

The above are best illustrated by example (see below).

5.3.3.1 Single integrals

The following are examples of single integrals:

$$\int_0^1 e^{-z^2}\, dx$$ `INTEGRAL(z := 0:1 ; EXP(-z^2))`

$$\int_0^L (T(z,r) - T_c)\, dz \quad r \in (0, R)$$ `INTEGRAL(z := 0:ReactorLength ; T(z,0|+:ReactorRadius|-)-Tc)`

Note that an integration variable (*e.g.* `z` in the above examples) is introduced to define the range of integration. The integrand is generally an expression that may involve the `MODEL` variables and/or the integration variable.

Also note that, in the first example above, the result of the `INTEGRAL` is just a scalar quantity. On the other hand, the second example results in an expression that is distributed over the open domain $(0, R)$.

5.3.3.2 Multiple integrals

Multiple integrals can be defined via a shorthand notation.

For example, the mean temperature in the entire tubular reactor is given by:

$$\overline{T} = \frac{2}{LR^2} \int_0^L \int_0^R rT(z,r)\, dr\, dz$$

which, in gPROMS, can be written as:

```
 2 / (ReactorLength* ReactorRadius^2) *
INTEGRAL(z := 0:ReactorLength , r := 0:ReactorRadius ; r*Temp(z,r))
```

5.3.3.3 Relationship between the `INTEGRAL` and `SIGMA` Operators

Since the `DISTRIBUTION` concept is a generalisation of `ARRAY` (*cf.* section 5.2), the `INTEGRAL` operator can also be used for the integration of a given expression over a discrete domain. Thus, `INTEGRAL` can be viewed as a generalisation of the `SIGMA` intrinsic function introduced in section 3.4.2 for carrying out discrete domain summations.

For instance, the molar fractions of the reactants at the tubular reactor entrance are defined as:

$$x_i = \frac{c_i}{\sum_k c_k}$$

This equation can be represented in terms of either the `SIGMA` function or the `INTEGRAL` function, as shown below:

```
# Using SIGMA function
X(,0) = C(,0) / SIGMA(C(,0)) ;

# Using INTEGRAL operator
X(,0) = C(,0) / INTEGRAL(i := 1:NoComp ; C(i,0)) ;
```

However, the `INTEGRAL` operator is more general than the `SIGMA` function; whereas `SIGMA` always results in a scalar by summing *all* dimensions of its argument, `INTEGRAL` can have a more narrowly specified summation domain. For instance, consider a two-dimensional array variable `A(5,10)`. Then

```
SIGMA ( A(2:4, ) )
```

is the scalar $\sum_{i=2}^{4}\sum_{j=1}^{10} A_{ij}$, whereas

```
INTEGRAL (i := 2:4 ; A(i, ) )
```

is a vector of length `10`, the jth element of which is $\sum_{i=2}^{4} A_{ij}$.

5.3.4 Explicit and implicit definition of distributed equations

Having introduced the concept of distributed expressions (*cf.* section 5.3.1) and the `PARTIAL` and `INTEGRAL` operators (*cf.* sections 5.3.2 and 5.3.3), the definition of distributed equations is fairly straightforward.

As in the case of array equations, there are two different ways of writing distributed equations in gPROMS. The first exploits the concept of distributed expressions (*cf.* section 5.3.1) to define equations in an *implicit* manner. For example, the following equation sets the temperature throughout the interior of the reactor to a uniform value of 298 K:

```
   T(0|+:ReactorLength|-,0|+:ReactorRadius|-) = 0 ;
```

This compact form of the equation will be automatically expanded by gPROMS (*cf.* section 3.3.1).

An alternative way of writing the same equation is in an *explicit* form by making use of `FOR` constructs (*cf.* section 3.3.2):

```
FOR z := 0|+ TO ReactorLength|- DO
   FOR r := 0|+ TO ReactorRadius|- DO
      T(z,r) := 298 ;
   END
END
```

The two forms are completely equivalent, from the points of view of both the definition of the equation and its numerical solution. Thus, which one you use depends on your preferences. However, the definition of some distributed equations require the extra flexibility afforded by the use of the `FOR` construct. One such case involves equations which involve the independent variables directly. Another case arises with equations involving `SIGMA` and/or `INTEGRAL` operators that need to be applied only to *some* of the domains of their arguments.

For example, consider the chemical species conservation equations within the tubular reactor. These are of the form:

$$\frac{\partial C_i}{\partial t} = -v\frac{\partial C_i}{\partial z} + \mathcal{D}_z\frac{\partial^2 C_i}{\partial z^2} + \frac{\mathcal{D}_r}{r}\frac{\partial}{\partial r}\left(r\frac{\partial C_i}{\partial r}\right) + \sum_{j=1}^{NR}\nu_{ij}r_j,\ i = 1..NC,\ z \in (0, L),\ r \in (0, R),$$

These can be written in gPROMS as follows:

```
FOR i := 1 TO NoComp DO
  FOR z := 0|+ TO ReactorLength|- DO
    FOR r := 0|+ TO ReactorRadius|- DO
      $Concentration(i,z,r) =
        - Velocity * PARTIAL(C(i,z,r),Axial)
        + Dz * PARTIAL(C(i,z,r),Axial,Axial)
        + (Dr/r) * PARTIAL(r*PARTIAL(C(i,z,r),Radial),Radial)
        + SIGMA(Nu(i,)*r(,z,r)) ;
    END
  END
END
```

Note how the range of application of each `FOR` construct is defined so as to ensure that the equation is enforced only at the interior of the domain of interest.

As a second example consider the energy conservation equation for the cooling jacket. This leads to a lumped equation that is related to the reactor energy balance through an integral term describing the heat flux over the entire length of the reactor:

$$\rho_{\mathrm{c}} C_{\mathrm{p,c}} V_{\mathrm{c}} \frac{dT_{\mathrm{c}}}{dt} = f_{\mathrm{c}} C_{\mathrm{p,c}} (T_{\mathrm{c,in}} - T_{\mathrm{c}}) + US \int_0^L (T(z,R) - T_{\mathrm{c}})\, dz.$$

This can be written in gPROMS as follows:

```
Rhoc * Cpc * Vc * $Tc = Fc * Cpc * ( TcIn - Tc )
  + U * S * INTEGRAL( z := 0:ReactorLength ; T(z,ReactorRadius)-Tc) ;
```

5.3.5 `BOUNDARY` Conditions

In contrast to initial conditions, which may differ from one simulation experiment to the next, boundary conditions are part of the description of the physical system behaviour itself. In gPROMS, they are therefore specified within `MODEL`s.

Boundary conditions can be viewed simply as additional equations relating the `MODEL` variables; consequently, they may be included in the `EQUATION` section, together with all other model equations. However, for the sake of clarity, the user is encouraged to include the boundary conditions in a separate section, under the keyword `BOUNDARY`.

For instance, the boundary condition for heat transfer at the tubular reactor wall,

$$-k_r \left.\frac{\partial T}{\partial r}\right|_{r=R} = U_{\mathrm{h}} (T - T_{\mathrm{wall}}) , \quad z \in (0, L),$$

can be written as follows:

```
BOUNDARY

  ...
```

```
    # Heat transfer relation at tube wall
    FOR z := 0|+ TO ReactorLength|- DO
      - Kr * PARTIAL(T(z,ReactorRadius),Radial) =
        Uh * ( T(z,ReactorRadius) - Twall(z) ) ;
    END

    ...

   EQUATION
```

In any case, gPROMS currently treats the `BOUNDARY` and `EQUATION` sections in exactly the same way for the purposes of numerical solution.

5.4 Specifying discretisation methods for distribution domains

The solution of systems of IPDAEs is generally a difficult problem. Changing the value of a parameter or one of the boundary conditions may lead to completely different behaviour from that originally anticipated. Furthermore, although some numerical methods can accurately solve a given system, other numerical methods may be totally unable to do so.

The systems of IPDAEs defined within gPROMS `MODEL`s are normally solved using the method-of-lines family of numerical methods. This involves discretisation of the distributed equations with respect to all spatial domains, which reduces the problem to the solution of a set of DAEs.

A number of different techniques fall within the method-of-lines family of methods, depending on the discretisation scheme used for discretising the spatial domains. Ideally, this discretisation scheme should be selected automatically—or, indeed, a single discretisation method that can deal efficiently with all forms of equations and boundary conditions would be desirable. However, this is not technically feasible at the moment and therefore gPROMS relies on the user to specify the preferred discretisation method.

Three specifications are necessary to completely determine most discretisation methods:

- *Type of spatial discretisation method.* The proper choice of the discretisation method is often the critical decision for solving a system of IPDAEs. As we mentioned earlier, because no method is reliable for all problems, the incorrect choice of method may lead to physically unrealistic solutions, or even fail to obtain any results.
- *Order of approximation.* The order of approximation for partial derivatives and integrals in finite difference methods, and the degree of polynomials used in finite element methods has a great influence on the accuracy of the solution. This is especially true if coarse grids or only a small number of elements are used.
- *Number of discretisation intervals/elements.* The number of discretisation intervals in finite difference methods and the number of elements in finite element methods are also of great significance in determining the solution trajectory. A coarse grid or a small number of elements for a steep gradient problem may result in an unacceptably inaccurate solution. On the other hand, too fine a grid or too many elements will increase the required computational efforts drastically, leading to an inefficient solution procedure.

The gPROMS language allows users to specify all three characteristics. `DISTRIBUTION_DOMAIN`s are treated as parameters and can be `SET` to the desired discretisation method, order and granularity of approximation. Table 5.2 lists the currently available numerical methods, their corresponding keywords in the language and the currently supported orders of approximation for each.

An excerpt from a `PROCESS` involving an instance `R101` of the `TubularReactor` `MODEL` is shown in figure 5.4. The `Axial` domain within this instance is to be discretised using centered finite differences of second order over a uniform grid of 150 intervals. On the other hand, the `Radial` domain is to be handled using third order orthogonal collocation over four finite elements.

Note that the specification of discretisation methods is done separately for each distribution domain in each instance of the corresponding `MODEL`, thus allowing maximal flexibility in this respect.

Numerical method	Keyword	Order(s)	Partial derivatives	Integrals
Centered finite difference method	CFDM	2, 4, 6	✓	✓
Backward finite difference method	BFDM	1, 2	✓	✓
Forward finite difference method	FFDM	1, 2	✓	✓
Orthogonal collocation on finite elements method	OCFEM	2, 3, 4	✓	✓
Gaussian quadratures		5		✓

Table 5.2: Numerical methods for distributed systems in gPROMS

```
# PROCESS StartUpSimulation

    UNIT
      R101 IS TubularReactor
      ...

    SET
      R101.Axial  := [  CFDM, 2, 150] ;
      R101.Radial := [ OCFEM, 3,   4] ;
      ...
```

Figure 5.4: `SET`ting the discretisation methods, orders and granularities

Also, similarly to other parameters, although it is possible to specify numerical solution method information within the `MODEL`s themselves, in the interests of model reusability and generality, is is often better to associate these with the specific instances of `MODEL`s that are included in `PROCESS`es.

Chapter 6

Composite Models

Contents

The `MODEL`s that we have considered so far have been simple, comprising relatively small numbers of parameters, variables and equations. In practical terms, this implies that all of these could be declared within a single `MODEL` entity.

However, many processing systems are much more complex than that, and their models involve many thousands or even tens of thousands of variables and equations. Although in principle all of these could be written in a single `MODEL`, in practice such an undertaking would be extremely tedious and error-prone.

This chapter deals with the tools that the gPROMS language provides for managing this kind of complexity. Section 6.1 discusses the idea of hierarchical sub-model decomposition. Sections 6.2 to 6.4 then describe the gPROMS language constructs that support this strategy.

6.1 Hierarchical sub-model decomposition

Consider the task of developing a model for the hypothetical process shown in figure 6.1. One approach would be to construct a single, primitive `MODEL` that would contain the declarations of all parameters, variable and equations for the entire flowsheet. This is, of course, a conceptually simple task that can be accomplished entirely using the ideas that we have already introduced. However, in practice, it may be very inefficient and error-prone. An alternative approach is, therefore, required.

Indeed, closer examination reveals that the flowsheet can be decomposed into three interconnected sections representing the pre-treatment, reaction and separation sections respectively. The development of models for each of these sections can initially be considered in isolation. Once these models are developed and tested, we can connect instances of each one of them in an appropriate way to construct the flowsheet model.

In fact, we can apply the above decomposition idea to the development of the models for the individual sections. The separation section may, for example, consist of a train of three distillation columns. It is clearly more efficient to develop a generic model of a distillation column, and then connect three instances of this model to form a model of the separation section. And then, at the next level, a distillation column model can also be decomposed into instances of models for the reboiler, condenser, stripping and rectification sections.

At this point, we may well decide that the reboiler and condenser models are simple enough that they do not need to be decomposed further but can simply be defined directly in terms of explicitly declared variables and equations. We call these "primitive" models. On the other hand, there is considerable advantage in decomposing the models of the stripping and rectification section into sets of instances of a tray model connected in a regular structure.

The procedure outlined above is called *hierarchical sub-model decomposition*. Its net result is that the model for a complex process is constructed progressively in a number of hierarchical levels. This is much easier than constructing a very large primitive model for the entire flowsheet because at each level in the hierarchy one can concentrate on only a small fraction of the modelling task. As a result, the models are easier to construct and less likely to contain errors.

Moreover, this strategy promotes model reusability. Suitably parameterised models of common process components, such as pumps, valves, or even complex structures such as distillation columns, can be used several times. The models of these components will thus have to be constructed and tested *once only*. Once this is done, instances of these models may be used to form more complex models without having to repeat the effort involved.

The gPROMS language encourages hierarchical sub-model decomposition by offering mechanisms that support:

- the declaration of high-level `MODEL`s that contain instances of lower-level `MODEL`s (see section 6.2);
- the connection of the above instances to represent flows of material, energy and information between them (*cf.* sections 6.3 and 6.4).

An important practical issue that arises in the use of models involving multiple hierarchical levels is that of the ease of specification of `PARAMETER` values. In many practical applications, the same parameters occur in many different model instances within the hierarchy. Clearly, a mechanism for avoiding the need for multiple specifications of the values for these parameters is highly desirable. gPROMS provides such a mechanism which is discussed in section 6.5.

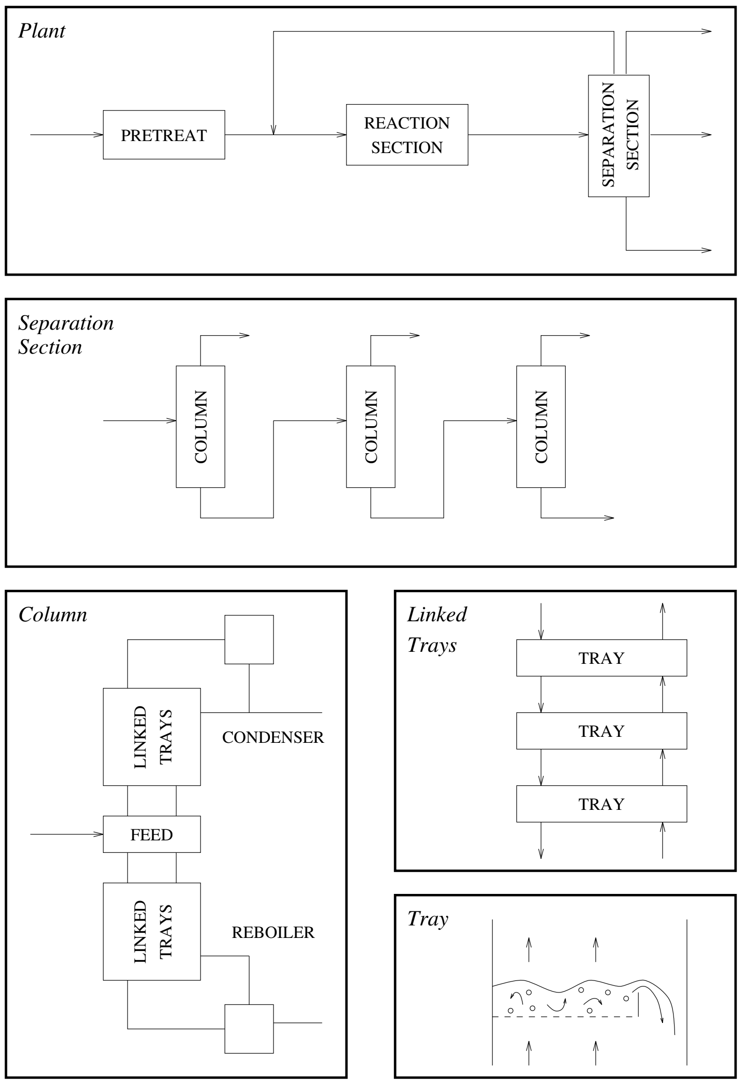


Figure 6.1: Hierarchical sub-model decomposition

6.2 Declaring higher-level MODELs

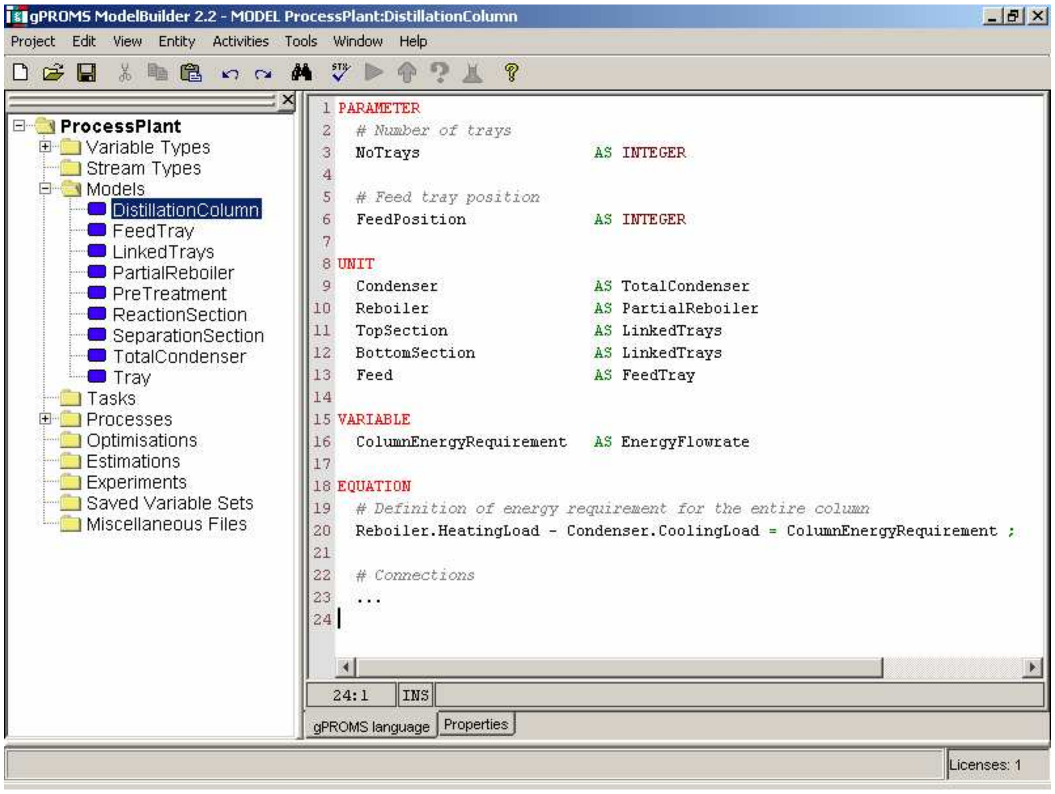
All models in gPROMS are described by `MODEL` entities, and this applies both to primitive ones and higher-level ones. The main difference is that the latter may contain instances of other models; however, just like primitive models, they can also contain their own parameters, variables and equations.

In this section, we see how instances of lower-level `MODEL`s may be inserted into higher-level ones.

6.2.1 Instances of lower-level models: the UNIT concept

In gPROMS, an instance of a `MODEL` is called a `UNIT`. Consequently, if we wish to insert one or more instances of one or more `MODEL`s within another (higher-level) `MODEL`, we have to introduce a `UNIT` section within the latter.

Consider, for instance, the declaration of a distillation column model that is outlined in figure 6.2.



```
PARAMETER
  # Number of trays
  NoTrays                     AS INTEGER

  # Feed tray position
  FeedPosition                AS INTEGER

UNIT
  Condenser                   AS TotalCondenser
  Reboiler                    AS PartialReboiler
  TopSection                  AS LinkedTrays
  BottomSection               AS LinkedTrays
  Feed                        AS FeedTray

VARIABLE
  ColumnEnergyRequirement     AS EnergyFlowrate

EQUATION
  # Definition of energy requirement for the entire column
  Reboiler.HeatingLoad - Condenser.CoolingLoad = ColumnEnergyRequirement ;

  # Connections
  ...
```

Figure 6.2: Distillation Column `MODEL`

The `PARAMETER` and `VARIABLE` sections of this model are very similar to those of simple models that we have seen earlier. In the former, two integer parameters (`NoTrays` and `FeedPosition`) are declared. These represent the number of trays in the column and the position of the feed tray respectively. The latter declares a single variable (`ColumnEnergyRequirement`) that represents the energy requirement of the entire column.

The new aspect that we wish to focus on is the `UNIT` section. This specifies that the `MODEL` also comprises a number of instances of other models, namely:

- `Condenser`. This is an instance of `MODEL TotalCondenser`.
- `Reboiler`. This is an instance of `MODEL PartialReboiler`.
- `TopSection`, `BottomSection`. These are both instances of `MODEL LinkedTrays`.
- `Feed`. This is an instance of `MODEL FeedTray`.

Each of the lower-level `MODEL`s may either be primitive or include a `UNIT` section themselves. For instance, `TotalCondenser`, `PartialReboiler` and `FeedTray` are likely to be primitive `MODEL`s. In contrast, `LinkedTrays` is most probably a composite `MODEL`. In any case, there is no limitation with respect to the number of levels in this hierarchical decomposition.

Finally, the `EQUATION` section introduces an equation that determines the net energy requirement for the entire column:

```
Reboiler.HeatingLoad - Condenser.CoolingLoad = ColumnEnergyRequirement ;
```

The equation involves three variables. One of these (`ColumnEnergyRequirement`) belongs directly to the `DistillationColumn` `MODEL` having been declared explicitly in its `VARIABLE` section. The other two variables (`Reboiler.HeatingLoad` and `Condenser.CoolingLoad`) belong to the `UNIT`s `Reboiler` and `Condenser` , respectively[1]

The above equation illustrates a *general* property of higher-level models. This is their ability to refer to entities (*e.g.* parameters and variables) that are declared within the `UNIT`s that they contain—as well, of course, as their own entities. Furthermore, as we have seen, this reference is done using a "pathname" construct. The latter can be arbitrarily long. For example, suppose that the `TotalCondenser` model is not primitive but comprises instances of several lower-level models; then the following may be a valid pathname:

```
Condenser.RefluxDrum.LevelController.Gain
```

referring to the gain of the controller used to control the liquid level in the reflux drum of the condenser.

With the introduction of hierarchical models, it is clear that a gPROMS project may contain more than one `MODEL` entity. This is illustrated in the `ProcessPlant` project in figure 6.2.

6.2.2 Arrays of `UNIT`s

As with variables and parameters, arrays of units may also be defined. Figure 6.3 illustrates a potential use of this feature in the definition of a `MODEL` for a series of distillation column trays.

[1]Of course, for this to be correct, `MODEL`s `PartialReboiler` and `TotalCondenser` must contain variables `Reboiler.HeatingLoad` and `Condenser.CoolingLoad` respectively.

Here, the higher-level `MODEL LinkedTrays` contains an array, called `Stage`, of instances of `MODEL Tray`. The parameters and variables within these instances can be referenced by combining the pathname and array notations. For instance, an equation within the `LinkedTrays` model may refer to the variable:

```
Stage(1).LiquidHoldup
```

Also, an equation within the `DistillationColumn` model of figure 6.2 may employ the variable:

```
TopSection.Stage(TopSection.NoTrays DIV 2).T
```

referring to the temperature at the middle tray of the top section of the column. Note that there is no confusion arising from the fact that the identifier `NoTray` appears in both the `DistillationColumn` and the `LinkedTrays` models.

6.2.3 The `WITHIN` construct

As the number of intermediate hierarchical levels increases, so does the length of the pathnames required to reference parameters and variables at or close to the bottom of the hierarchy. Pathnames of the form:

```
SeparationSection.Column(2).TopSection.Stage(1).T
```

are quite common when dealing with complex processes. Writing equations like that becomes increasingly tedious, especially if a large part of the pathname is common to many of the parameters or variables referenced by an equation. The `WITHIN` construct helps relieve some of this burden.

A `WITHIN` construct encloses a list of equations and defines a *prefix* to be used for all parameters and variables referenced by the enclosed equations. For instance:

Suppose, for instance, that `MODEL Tray` has a variable `Q` that determines the heat loss from the tray to the environment. However, it contains no equation that actually determines this heat

```
# MODEL LinkedTrays

  PARAMETER

    # Number of trays
    NoTrays AS INTEGER

  UNIT
    Stage   AS ARRAY(NoTrays) OF Tray

  ...
```

Figure 6.3: `MODEL` for a series of linked trays

loss. Consider now using this tray model within the top and bottom sections of a column. We wish to specify a simple heat transfer equation for determining the heat loss in the top section; however, the bottom section is well insulated. One way we can achieve this is as follows:

```
# MODEL DistillationColumn

  PARAMETER
    TAmbient   AS   REAL
    UA         AS   REAL
    NoTrays    AS   INTEGER

  UNIT
    TopSection, BottomSection    AS    LinkedTrays

  EQUATION
    WITHIN TopSection DO
      FOR k := 1 TO NoTrays DO
        WITHIN Stage(k) DO
          Q = UA * (T - TAmbient) ;
        END # Within Stage(k)
      END # For k
    END # Within TopSection

    WITHIN BottomSection DO
      FOR k := 1 TO NoTrays DO
        WITHIN Stage(k) DO
          Q = 0 ;
        END # Within Stage (k)
      END # For k
    END # Within BottomSection
```

In trying to interpret the equation:

```
                  Q = UA * (T - TAmbient) ;
```

gPROMS will need to identify ("resolve") the symbols `Q`, `UA`, `T` and `TAmbient`. It starts doing this by searching the `MODEL` corresponding to the `UNIT` mentioned in the innermost `WITHIN` statement. In this case, this is `MODEL Tray` which does, indeed, contain variables `Q` and `T`. However, symbols `UA` and `TAmbient` still remain unresolved. Therefore, gPROMS considers the next enclosing `WITHIN` statement; this indicates that it should search `MODEL LinkedTrays` (of which `TopSection` is an instance). However, this model does not contain the missing identifiers. So gPROMS now has to consider `MODEL DistillationColumn` itself—which does indeed allow it to resolve `UA` and `TAmbient`.

It is interesting to note that both the `DistillationColumn` and the `LinkedTrays` models contain the parameter `NoTrays`. However, this does not result in any ambiguity in resolving this parameter when it appears in each of the two `FOR` constructs: gPROMS always tries to resolve symbols by searching the innermost `WITHIN` first. Thus, `NoTrays` in:

```
        WITHIN TopSection DO
```

```
    FOR k := 1 TO NoTrays DO
       . . . . . . . . . . . . . .
    END
  END # within TopSection
```

clearly refers to `TopSection.NoTrays` and *not* to parameter `NoTrays` in `DistillationColumn`.

6.3 Specifying connections as `EQUATION`s

So far, we have seen how we can introduce `MODEL` instances within higher-level `MODEL`s. In most practically useful applications, these instances have to be connected with each other to complete the definition of the higher-level model. This section and the next one are concerned with mechanisms for effecting such connections.

In mathematical terms, connections between instances of mathematical models can be viewed simply as equality constraints between subsets of their variables[2].

Consider, for example, the `MixingTank` and `Pump` `MODEL`s shown in figures 6.4 and 6.5 respectively. `MODEL` `MixingTankWithPump` (figure 6.6) connects instances of these two to form a higher-level model of a mixing tank equipped with a product pump.

`MixingTankWithPump` defines an additional variable, namely `FlowAccumulation`, to represent the rate of material accumulation in the combined system. Then, the equations:

```
  StorageTank.Fout = ProductPump.Fin ;
  StorageTank.X    = ProductPump.Xin ;
```

connect the sub-models while:

```
  FlowAccumulation = StorageTank.Fin - ProductPump.Fout ;
```

is an additional equation that calculates `FlowAccumulation`.

While this way of specifying connections is both perfectly valid and conceptually simple, it is not particularly elegant. Even for this small example that involves only one connection, two equations must be specified in order to state that the flows and compositions in the two model instances should be equal to each other.

As the amount of information (*e.g.* intensive properties) that needs to be exchanged between model instances increases, the number of connecting equations will increase accordingly, and the task of specifying connectivity starts becoming rather unwieldy. Therefore, gPROMS provides a more efficient mechanism for achieving the same task. This makes use of the concept of `STREAM`s as described in the next section.

[2]It is possible to consider connections involving more general relations between subsets of variables of different model instances. However, gPROMS does not allow this. Instead, such connections may themselves be described as `MODEL`s (*e.g.* of a pipe).

```
# MODEL MixingTank
    PARAMETER
      NoComp, NoInput    AS INTEGER
      CrossSectionalArea AS REAL
      Density            AS ARRAY(NoComp) OF REAL
    VARIABLE
      Fin                AS ARRAY(NoInput) OF MassFlowrate
      Xin                AS ARRAY(NoInput,NoComp) OF MassFraction
      Fout               AS MassFlowrate
      X                  AS ARRAY(NoComp) OF MassFraction
      HoldUp             AS ARRAY(NoComp) OF Mass
      TotalHoldup        AS Mass
      TotalVolume        AS Volume
      Height             AS Length
    EQUATION
      # Mass balance
      FOR i := 1 TO NoComp DO
        $HoldUp(i) = SIGMA(Fin*Xin(,i)) - Fout*X(i) ;
      END
      # Mass fractions
      TotalHoldup = SIGMA(HoldUp) ;
      Holdup = X * TotalHoldup ;
      # Calculation of liquid level from holdup
      TotalVolume = SIGMA(Holdup/Density) ;
      TotalVolume = CrossSectionalArea * Height ;
```

Figure 6.4: Multi-component mixing tank

```
# MODEL Pump
    PARAMETER
      NoComp      AS INTEGER
      PumpFlow    AS  REAL
    VARIABLE
      Fin, Fout   AS MassFlowrate
      Xin, Xout   AS ARRAY(NoComp) OF MassFraction
    SELECTOR
      PumpStatus  AS (PumpOn,PumpOff) DEFAULT PumpOn
    EQUATION
      # Mass balance
      Fout = Fin ;
      Xout = Xin ;
      # Pump operation
      CASE PumpStatus OF
        WHEN PumpOn  : Fout = PumpFlow ;
        WHEN PumpOff : Fout = 0.0 ;
      END
```

Figure 6.5: On/off positive displacement pump

```
# MODEL MixingTankWithPump

    VARIABLE
      FlowAccumulation AS MassFlowrate

    UNIT
      StorageTank AS MixingTank
      ProductPump AS Pump

    EQUATION
      # Connection in terms of variables
      StorageTank.Fout = ProductPump.Fin ;
      StorageTank.X    = ProductPump.Xin ;

      # Calculate overall rate of material accumulation
      FlowAccumulation = StorageTank.Fin - ProductPump.Fout ;
```

Figure 6.6: Multicomponent mixing tank with pump

6.4 Specifying connections using STREAMs

6.4.1 The STREAM concept

As illustrated in figure 6.2, the second entry down in a gPROMS project tree within the ModelBuilder environment is `Stream Types`.

In gPROMS `MODEL`s, `STREAM`s are simply subsets (not necessarily disjoint) of the set of variables in the `MODEL`. They provide a convenient mechanism for specifying complex connections between different components of a physical system.

`STREAM`s are declared in the `STREAM` section of a `MODEL`. The declaration comprises:

1. An identifier (name) by which the stream will be referenced.
2. A list of the variables that the stream contains. These may be
   - single variables, arrays or slices of arrays declared in the `VARIABLE` section of the `MODEL`;
   - variables or streams declared in instances of lower-level models contained directly or indirectly within the `MODEL`.
3. A stream type. The declaration of stream types is discussed in section 6.4.2.

For instance, a potential `STREAM` section for the `Pump` `MODEL` of figure 6.5 is shown in figure 6.7. Here, two streams are defined, namely `Inlet` and `Outlet`. The first comprises the inlet flow and composition variables of the `MODEL` in question—a single variable of type `MassFlowrate` and an array of type `MassFraction` respectively. The second comprises the corresponding outlet variables. Both streams are of type `FXStream`. Detailed descriptions of `Stream Types` declarations will be discussed in section 6.4.2.

```
STREAM
  Inlet  : Fin,  Xin  AS FXStream
  Outlet : Fout, Xout AS FXStream
```

Figure 6.7: `STREAM` section for `Pump` `MODEL`

Arrays of streams can be declared in the same fashion. Figure 6.8 shows a potential `STREAM` section for the `MixingTank` `MODEL` of figure 6.4. `Inlet` is an `ARRAY(NoInput)` of streams of type `FXStream`. Each of the variables contained in `Inlet` is also an array, the first dimension of which must be the same as that of `Inlet`. In this case, `Fin` is an `ARRAY(NoInput) OF MassFlowrate`, while `Xin` is an `ARRAY(NoInput,NoComp) OF MassFraction`.

```
STREAM
  Inlet  : Fin,  Xin  AS ARRAY(NoInput) OF FXStream
  Outlet : Fout, Xout AS FXStream
```

Figure 6.8: `STREAM` section for `MixingTank` `MODEL`

6.4.2 Stream type declarations

Stream type declarations contain the following information:

1. An identifier (name) by which the stream type may be referenced globally.
2. An ordered list of variable types.

Stream types are defined in a similar manner to other entities (*cf.* section 2.2.3). Thus, in order to define a new stream type:

1. Pull-down the `Entity` menu from the top bar of the ModelBuilder environment and click on `New Entity`. A dialog box will appear.
2. Choose `STREAM TYPE` for the `Entity Type`.
3. Fill in the `Name` field (*e.g.* `FXStream`) and click on `OK`.

`Stream types` can be defined either using the "gPROMS language" tab or using the "Variable types" tab (see figure 6.9).

It is recommended that the "Variable types" tab is used, this provides a drop down list containing all currently defined Variable Types. The user may either select one of the existing Variable types, or enter a name for a new one that is still undefined at this stage. Pressing the "Add" button introduces the variable type in the Stream types's list immediately below the Variable type that is currently highlighted. "Raise" and "Lower" buttons are provided for the user to modify the position of any Variable type in this list at any point.

The "Variable types" and "gPROMS language" tabs in the stream editor remain synchronised and consistent at all times. Thus, any change effected by the user in either of these two tabs is immediately reflected in the other.

The declaration of stream type `FXStream` (which we have already seen being used in figures 6.7 and 6.8) is shown in figure 6.9. Stream type `FXStream` contains flowrate and composition information (in that order).

This declaration implies that *any* stream of type `FXStream` appearing in a `MODEL` must contain a variable of type `MassFlowrate`, followed by a variable of type `MassFraction`. However, the stream type declaration imposes no restrictions on the *dimensionality* of these variables. Thus, the second variable of a stream of type `FXStream` in a `MODEL` may be a single mass fraction, or an array of mass fractions of any number of dimensions and any size of each dimension, or, indeed, a distribution of mass fractions.

Nevertheless, the number and types of variables in a `MODEL` stream must match those in the stream type declaration. This is indeed the case for streams `Inlet` and `Outlet` in figure 6.7 which are of type `FXStream` and both contain a variable of type `MassFlowrate` (`Fin` and `Fout` respectively) and an array of variables of type `MassFraction` (`Xin` and `Xout` respectively).

6.4.3 Connecting models via `STREAM`s

Once defined, streams can be used to declare connections between model instances in much more compact and less error prone manner than by using equations to relate individual variables (as we had been doing in section 6.3).

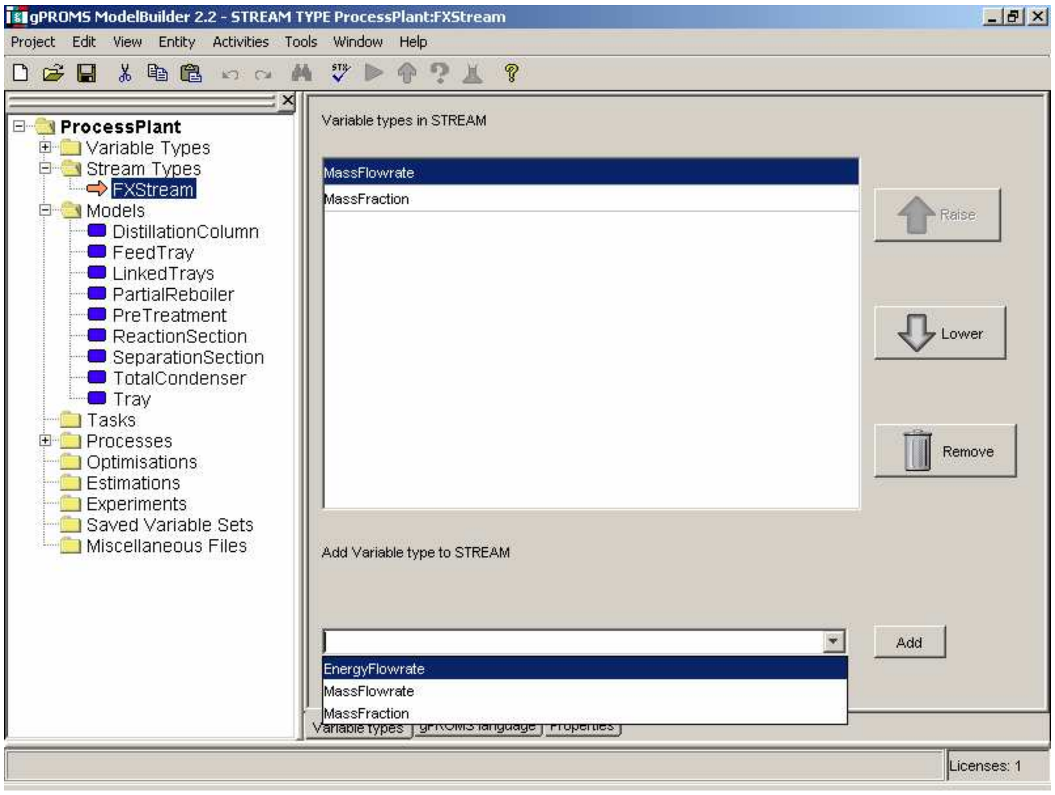


Figure 6.9: `StreamType` Entity Editor

Consider again the `MixingTankWithPump` `MODEL` of figure 6.6. Assuming that the `STREAM` sections in figures 6.7 and 6.8 are inserted in the `MixingTank` and `Pump` `MODEL`s of figures 6.5 and 6.4 respectively, the higher-level `MODEL` can take the form shown in figure 6.10.

Note, in particular, the stream connection equation[3]:

```
    StorageTank.Outlet = ProductPump.Inlet ;
```

For this equation to be valid:

- the two streams involved must belong to the same type (*e.g.* `FXStream` in this case);
- the corresponding variables in each stream must have the same dimensionality and size[4].

Assuming a stream connection equation is valid, it is then automatically expanded by gPROMS to enforce equality of the corresponding variables in each stream.

[3]gPROMS also supports the equivalent syntax `StorageTank.Outlet IS ProductPump.Inlet ;`.

[4]As we saw in section 6.4.2, the fact that a stream belongs to a certain type does not constrain the actual dimensionality or size of the variables in it.

Also note that `MODEL MixingTankwithPump` now defines its own streams, namely `Inlet` and `Outlet`.

```
STREAM
  Inlet  IS StorageTank.Inlet
  Outlet IS ProductPump.Outlet
```

The above statement is, therefore, a shorthand to specify that the inlet stream of the tank/pump configuration is simply the inlet stream of the tank while its outlet stream is simply the outlet stream of the pump.

```
STREAM
  Inlet  : StorageTank.Fin, StorageTank.Xin AS FXStream
  Outlet : ProductPump.Fout, ProductPump.Xout AS FXStream
```

Another example is shown in figure 6.11. Here, a model for a `MixingTankFarm` is formed by connecting several instances of multi-component mixing tanks in series.

The `Inlet` and `Outlet` streams are used to connect the sub-models, the output of each vessel being the first input of its successor in the series. The `Feed` and `Product` streams of the `MixingTankFarm` are declared to be the `Inlet` streams of the first tank and the `Outlet` stream of the last respectively.

```
# MODEL MixingTankWithPump

    VARIABLE
      FlowAccumulation AS MassFlowrate

    UNIT
      StorageTank AS MixingTank
      ProductPump AS Pump

    STREAM
      Inlet  IS StorageTank.Inlet
      Outlet IS ProductPump.Outlet

    EQUATION
      # Connection in terms of streams
      StorageTank.Outlet = ProductPump.Inlet ;

      # Calculate overall rate of material accumulation
      FlowAccumulation = StorageTank.Fin - ProductPump.Fout ;
```

Figure 6.10: Multicomponent mixing tank with pump

```
# MODEL MixingTankFarm

    PARAMETER
      NoTank AS INTEGER

    UNIT
      StorageTank AS ARRAY(NoTank) OF MixingTank

    STREAM
      Feed    IS StorageTank(1).Inlet
      Product IS StorageTank(NoTank).Outlet

    EQUATION
      FOR i := 1 TO NoTank - 1 DO
        StorageTank(i+1).Inlet(1) IS StorageTank(i).Outlet ;
      END
```

Figure 6.11: Several mixing tanks connected in series

6.5 Parameter value propagation

For a MODEL involving many hierarchical levels, it is important to be able to unambiguously set the values of the parameters occurring both in the MODEL itself and in instances of other MODELs contained within it, either directly or indirectly.

It is also important to be able to propagate such values from higher- to lower-level models. For example, a parameter that corresponds to the number of components (*e.g.* NoComp) may well be present in the higher-level model of a distillation column *and* in all of its constituent model instances. Ideally, we would like to be able to set the value of this parameter at the highest level only and rely on an automatic mechanism to propagate it through the hierarchy towards the lower levels. This not only saves effort in specifying the model but also reduces the possibility of errors arising due to inconsistent specifications, especially during model development (*e.g.* specifying NoComp=5 in some parts of the column and NoComp=4 in others).

The two sections that follow describe the two mechanisms that gPROMS provides for dealing with these problems.

6.5.1 Explicit parameter assignment

We have already seen (*cf.* section 2.2.5.2) that the value for a parameter may be specified explicitly in the SET section of a PROCESS. In fact, MODELs also have a similar SET section that can be used for the same purpose.

Consider, for instance, the following situation:

- MODEL X declares a parameter S to be of type REAL.
- MODEL Y contains a unit XX which is an instance of X.
- MODEL Z contains a unit YY which is an instance of Y.
- PROCESS P contains a unit ZZ which is an instance of Z.

Then, the value of parameter S can be explicitly set in *at most one* of the following ways:

1. In X:

   ```
   SET
     S := 1.5 ;
   ```

2. In Y:

   ```
   SET
     XX.S := 1.5 ;
   ```

3. In Z:

   ```
   SET
     YY.XX.S := 1.5 ;
   ```

4. In P:

```
SET
  ZZ.YY.XX.S := 1.5 ;
```

The existence of these different possibilities raises some important issues regarding model reusability and ease of use. In particular, note that:

- If a parameter is explicitly `SET` in a `MODEL`, it will have that value in *all* instances of that `MODEL`. For example, if we use option 2 above, `XX.S` will have a value of 1.5 in all subsequent instances of `MODEL Y` anywhere in the problem.
- A parameter may be given an explicit value as described above *at most once*. In other words, if we use option 3 above, we cannot override this value by using option 4 later.

It is usually advisable that parameters be explicitly set at the `PROCESS` level. This practice maximises the reusability of the underlying `MODEL`s and minimises the probability of error.

6.5.2 Automatic parameter propagation

If a parameter appearing in an instance of a `MODEL` is not `SET` explicitly, gPROMS will automatically search hierarchically the higher-level `MODEL`s containing it for a parameter *of the same name and type* which has been given an explicit value. If this is found, the parameter in the lower-level `MODEL` will adopt the value assigned to the parameter with the same name in the higher-level `MODEL`.

Another way of looking at this is that an explicit `SET` specification for a parameter in a higher-level `MODEL X` propagates downwards and covers all parameters of the same name and type in any lower-level `MODEL`s, instances of which are contained in `X`. This establishes an *automatic parameter propagation* mechanism.

For instance, consider the hierarchy of `MODEL`s `X`, `Y` and `Z` mentioned in section 6.5.1 and suppose that all of them contain a declaration of an integer parameter `NoComp`. We then have various possibilities:

1. Set parameter in P:

   ```
   SET
     ZZ.NoComp := 5 ;
   ```

   Although nothing is said explicitly about parameters `ZZ.YY.NoComp` and `ZZ.YY.XX.NoComp`, the automatic parameter propagation will ensure that these also take the value of 5.

2. Set parameter in P:

   ```
   SET
     ZZ.NoComp := 5 ;
   ```

   but also in `Y`:

   ```
   SET
     NoComp := 3 ;
   ```

This is equivalent to

```
ZZ.NoComp = 5 ;
ZZ.YY.NoComp = 3 ;
ZZ.YY.XX.NoComp = 3
```

Note that the value of the parameter in `YY` is set explicitly and automatically propagates downwards, setting the value of the parameter in `XX`. Therefore, when gPROMS automatically propagates the assignment in `P`, it cannot override the existing value.

We also recall that the specification of discretisation methods for distribution domains (*cf.* section 5.4) is treated exactly as that for parameters – hence, it also undergoes automatic propagation. For instance, if:

```
SET
  ZZ.Axial := [OCFEM, 3, 10] ;
```

appears in a `PROCESS`, all model instances within `ZZ` which declare an `Axial` domain will use the same discretisation method—unless, of course, their `Axial` specification is explicitly `SET` to a different value.

Automatic parameter propagation is very useful because it effectively allows parameter values (for numbers of components, physical property coefficients *etc.*) to be specified only once despite being used by many model instances in a problem.

If a parameter is not `SET` explicitly *and* gPROMS cannot deduce its value using automatic parameter propagation, it will check whether a default value for this parameter has been specified at the time of its declaration in the `MODEL` (*cf.* section 2.2.3.3). If this is the case, it will use that default value. Otherwise, it will issue an error message.

Chapter 7

Simple Operating Procedures

Contents

As was mentioned in chapter 2, operating procedures are specified in the `SCHEDULE` section of a `PROCESS`. The gPROMS language for describing operating procedures is based on a number of "elementary tasks" that can be used to specify simple actions (*e.g.* changing the values of simulation input variables, specifying periods of undisturbed operation *etc.*). Elementary tasks can be combined using "timing structures" which specify the manner in which they are executed (sequentially, concurrently, conditionally or iteratively) in order to form more complex operating procedures.

In the first section of this chapter, we examine each of the elementary tasks in isolation and describe the functions they perform. In the second section, we proceed to describe the timing structures and show how elementary tasks may be combined in order to define more complex operating procedures.

7.1 Elementary tasks

7.1.1 The RESET elementary task

The `RESET` elementary task is used to change the value assigned to one or more of a simulation's input variables. Figure 7.1 demonstrates some applications of the `RESET` task.

```
RESET
  V101.Position := 1.0 ;
END
```

(a)

```
WITHIN C101 DO
  RESET
    Signal:= Bias + Gain * ( Error + IntegralError/ResetTime ) ;
  END
END
```

(b)

```
RESET
  T101.FlowIn := OLD(T101.FlowIn) + 0.1 ;
END
```

(c)

Figure 7.1: Applications of the `RESET` task

In the first example, a `RESET` task is used to model the instantaneous opening of a manual valve by a process operator. The `MODEL` that corresponds to unit `V101` contains a variable called `Position`, which represents the position of the valve stem, and an equation that, according to the position of the stem and the inherent characteristics of the valve, relates the flowrate through the valve to the pressure drop across it.

The initial position of the valve stem is specified in the `ASSIGN` section of the corresponding `PROCESS`. During the simulation, the `RESET` task shown in figure 7.1 "reaches" into the model and changes the value of this input variable, just as an operator would walk into the plant and manipulate the valve. The action is considered to occur in such a small time interval relative to the length of the entire simulation, that it can be modelled as an instantaneous change.

The second example demonstrates how the action of a digital controller at the end of its sampling interval might be modelled. Here, the expression on the right hand side of the assignment is evaluated at the time of execution of the `RESET` task. The value obtained is used to update the value of the control signal instantaneously and according to a proportional-integral control law.

Finally, in the third example the `RESET` task is used to impose a step change of magnitude 0.1 on variable `T101.FlowIn`, representing the input flowrate to a vessel. This example also illustrates the use of the `OLD` built-in function to refer to the value of the variable *immediately before* the execution of the `RESET` task. Note that the `OLD` function has no meaning within `MODEL`s, because no well-defined values for the variables exist before the simulation commences.

7.1.2 The SWITCH elementary task

Similar to the `RESET` task, the `SWITCH` task may be used to alter the value of selector variables (*cf.* chapter 4). Manipulation of a selector variable by a `SWITCH` task forces the underlying model to change state as a result of an external action as opposed to a physico-chemical mechanism. Applications include the switching of a pump on or off (figure 7.2), the replacement of a shattered bursting disc by an operator (*cf.* section 4.1) *etc.*

```
#  MODEL Pump

    VARIABLE
      FlowIn, FlowOut   AS Flowrate
      PressIn, PressOut AS Pressure
      PressRise         AS Pressure

    SELECTOR
      PumpStatus        AS (PumpOn,PumpOff)

    EQUATION

      FlowOut = FlowIn ;

      PressOut = PressIn + PressRise ;

      CASE PumpStatus OF
        WHEN PumpOn  : FlowOut = f(PressRise) ;
        WHEN PumpOff : PressRise = 0 ;
      END # Case

  END # Model Pump
```

(a)

```
  SWITCH
    P101.PumpStatus := P101.PumpOn ;
  END # Reset
```

(b)

Figure 7.2: Manipulating selector variables using the `SWITCH` task

In figure 7.2, `MODEL Pump` has two states, `On` and `Off`, designated by the selector variable `Status` and representing whether the pump is on or off. When the pump is switched on, the pump characteristic relates the pressure rise across the pump to the flowrate through the pump. When the pump is switched off, the pressure rise is set to zero. Note that no `SWITCH` statements are present because no physico-chemical transitions link these two states.

Whether the pump is initially switched on or off forms part of the initial condition of each simulation experiment. This information is specified in the `SELECTOR` section of the corresponding `PROCESS`. On the other hand, external actions during the simulation are modelled by `SWITCH` tasks and cause dynamic changes to this status (figure 7.2).

7.1.3 The REPLACE elementary task

The REPLACE elementary task "unASSIGNs" an input variable (leaving it free to vary) and ASSIGNs a different one in its place.

An interesting application of the REPLACE task is the automatic calculation of the steady-state bias of a controller. In order to determine the bias, a steady-state calculation is performed in which the controller error is set to zero by an input equation, while the bias is free to vary. The bias value obtained by this calculation corresponds to the correct steady-state bias for the controller. Therefore, before dynamic simulation begins, the REPLACE task shown in figure 7.3 can be used to "unASSIGN" the error variable and ASSIGN the bias variable to its steady-state value. The controller error is then free to fluctuate as disturbances are introduced and the controller attempts corrective action.

```
REPLACE
  PI101.Error
WITH
  PI101.Bias := OLD(PI101.Bias) ;
END
```

Figure 7.3: Automatic calculation of controller bias using a REPLACE task

7.1.4 The REINITIAL elementary task

Both the RESET and REPLACE elementary tasks introduce discontinuities in the simulation. Although these discontinuities may affect the values of input and/or algebraic variables, they do not normally affect the values of differential variables. The latter usually represent quantities that are conserved according to the laws of physics (*e.g.* mass, energy, momentum *etc.*) and are therefore continuous across such discontinuities; gPROMS follows this assumption and normally expects the values of the differential variables before the discontinuity to be the same as those just after the discontinuity.

The REINITIAL elementary task makes it possible to introduce discontinuities in the differential variables themselves. Of course, once we drop the continuity assumption, we need to provide some other information to replace it.

Two examples of the application of the REINITIAL task are shown in figure 7.4. In the first example, the integral error of a PI controller is reset to zero. The execution of this task will result in a reinitialisation calculation in which the usual assumption concerning the continuity of differential variable PI101.IntegralError will be replaced by the equation in the second clause of the REINITIAL task. The latter simply states that the value of PI101.IntegralError after the discontinuity is zero. Note that this is a general equation and not just an assignment, which is why we do **not** write:

```
PI101.IntegralError := 0 ;
```

This is consistent with the treatment of general initial conditions in gPROMS.

In the second example, the holdups of components A and B in a chemical reactor change by instantaneous additions of material. The amounts added are such that, in the final mixture, the holdup of A is doubled while the mass fraction of B is 0.3. Note again that the condition specified is a general equation involving any variables in the problem and not just the ones that are reinitialised.

Naturally, the number of differential variables in the first clause of a `REINITIAL` task must match the number of equations in the second clause.

7.1.5 The `CONTINUE` elementary task

The execution of all elementary tasks described so far takes place instantaneously with respect to the simulation clock. The `CONTINUE` elementary task provides the mechanism by which periods of undisturbed operation between discrete actions can be specified.

We have already used the `CONTINUE` task in its simplest form:

`CONTINUE FOR` *TimePeriod*

This specifies a period of undisturbed process operation, starting from when the `CONTINUE` task is encountered and extending until the simulation clock has advanced *TimePeriod* time units. In fact, as well as being a real number, *TimePeriod* may alternatively be a real expression involving any quantities that the schedule has access to. For example:

```
# Continue for 100 time units
CONTINUE FOR 100

# Continue for period equal to the sampling interval
CONTINUE FOR C101.SamplingInterval
```

Alternatively, the period of undisturbed process operation can be specified implicitly, in terms of a logical condition:

`CONTINUE UNTIL` *LogicalCondition*

In this case, simulation continues until *LogicalCondition* becomes true. Again, *LogicalCondition* can be of arbitrary complexity and can involve any quantities that the schedule has access to. For example:

```
# Continue until required conversion has been achieved
CONTINUE UNTIL R101.Conversion > 0.95

# Continue until reactant holdups have been exhausted
CONTINUE UNTIL R101.HoldUp(1) < Epsilon AND R101.HoldUp(2) < Epsilon
```

The two forms described above may also be combined in a single `CONTINUE` task through the use of `AND` and `OR` operators:

```
CONTINUE FOR TimePeriod AND UNTIL LogicalCondition
CONTINUE FOR TimePeriod OR UNTIL LogicalCondition
```

Here, the period of undisturbed operation extends until the simulation clock has advanced *TimePeriod* time units and/or until *LogicalCondition* becomes true, respectively. For instance,

```
CONTINUE FOR 100 OR UNTIL R101.Conversion > 0.95
```

advances the simulation for *at most* 100 time units even if the reactor conversion never reaches the required value, while

```
CONTINUE FOR 100 AND UNTIL R101.Conversion > 0.95
```

advances the simulation for *a minimum of* 100 time units and then waits for the reactor conversion to reach the required value.

```
REINITIAL
  PI101.IntegralError
WITH
  PI101.IntegralError = 0 ;
END
```

(a)

```
REINITIAL
  R101.HoldUp(1),
  R101.HoldUp(2)
WITH
  R101.HoldUp(1) = 2 * OLD(R101.Holdup(1)) ;
  R101.X(2) = 0.3 ;
END
```

(b)

Figure 7.4: Applications of the `REINITIAL` task

7.2 Specifying the relative timing of multiple tasks

7.2.1 Sequential execution—`SEQUENCE`

Sequential execution begins with the first task and only proceeds to the next task when execution of the preceding task has terminated. Sequential execution of a series of tasks is specified by enclosing them within a `SEQUENCE` structure. The execution of a `SEQUENCE` structure is complete when the execution of the last task in the structure has terminated.

Figure 7.5 shows a `PROCESS` for a simulation experiment involving a multi-component mixing tank (a `MODEL` for a unit of this type was shown in figure 3.4). Unit `T101` is a tank with two input streams, containing a mixture of components A, B and C. The values of the flowrates and component mole fractions of the inlet streams are specified in the `ASSIGN` section. The outlet valve is closed. Initially, the tank contains 1000kg of component A, with additional components B and C to make up a volume of 1.5m^3. The initial amount of component C is twice that of component B. The schedule of operation in figure 7.5 contains only a `CONTINUE` task, thus defining a period of continuous operation with a duration of 120 time units.

Figure 7.6 illustrates how a more complicated operating procedure may be defined by using a `SEQUENCE` structure in the `SCHEDULE` section. The execution of this simulation experiment will result in the following:

1. Simulation begins. Based on the input equations specified in the `ASSIGN` section and the initial conditions specified in the `INITIAL` section, consistent initial values are determined for all variables in the system.
2. The first `CONTINUE UNTIL` task is executed. Simulation proceeds until the volume of liquid in the tank exceeds 3.5m^3.
3. The first `RESET` task is executed. The flowrate of the first inlet stream is set to zero, while that of the second inlet stream is increased by 50%.
4. The second `CONTINUE UNTIL` task is executed. Simulation proceeds until the volume of liquid in the tank exceeds 5m^3.
5. The second `RESET` task is executed. The flowrates of both inlet streams are set to zero. The outlet valve is opened completely.
6. The third `CONTINUE UNTIL` task is executed. Simulation proceeds until the tank drains.

7.2.2 Concurrent execution—`PARALLEL`

Tasks to be executed in parallel are enclosed within a `PARALLEL` structure. Execution of all tasks begins simultaneously and proceeds concurrently. The execution of a `PARALLEL` structure is completed when *all* tasks have terminated.

Figure 7.7 demonstrates the use of concurrent tasks in specifying an operating policy for the mixing tank example. The operating policy is in fact the same as in figure 7.6. However, here, the operating policies for the two inlet and the outlet streams are specified separately with three `SEQUENCE` structures. These are then enclosed in a `PARALLEL` structure, so that the three policies are executed concurrently.

7.2.3 Conditional execution—IF

In many circumstances, the correct external actions to apply to a system cannot be fully determined *a priori* and must be established from decisions that can only be made while the simulation is in progress. For instance, consider a batch operation involving a series of elementary processing steps applied to a batch of material. Once all steps are completed a decision is made as to whether the batch is acceptable, should receive further processing or should be discarded. This decision depends only on the quality of the batch, so the result can only be established after the preceding steps have been completed.

The `IF` conditional structure enables selection between alternative actions based on the current status of a system. In common with most programming languages, it comprises an `IF` clause, an optional `ELSE` clause and a logical condition. If the logical condition is true when the `IF` structure is encountered, the contents of the `IF` clause are executed; otherwise, the contents of the `ELSE` clause are executed. As with all other timing structures, conditional structures can be nested in arbitrary manner.

Figure 7.8 shows the application of the `IF` structure to "clipping" a digital control signal before sending it off to a control valve. If, at the time of execution, the signal proves to be greater than 1.0 or less that 0.0, the stem position is set to 1.0 and 0.0 respectively. Otherwise, the stem position is set to the value indicated by the control signal.

7.2.4 Iterative execution—WHILE

Many processing systems are characterised by the repetitive nature of external actions required to achieve the desired operation. For example, periodic processes, such as pressure swing or temperature swing adsorption, are usually brought to and maintained at a "cyclic steady-state" by a sequence of external actions that is applied repeatedly. Also, the action of a digital control system on a process can be considered to consist of a regular cycle between continuous operation, sampling and implementation of discrete actions.

The `WHILE` iterative structure permits the repeated execution of the tasks it encloses for as long as a logical condition is satisfied. When the `WHILE` structure is first encountered, the logical condition is examined. If it is satisfied, the enclosed tasks are executed. The condition is then examined again and, if still satisfied, the enclosed tasks are executed once more. This process continues until the condition is no longer satisfied, at which point the execution of the `WHILE` structure is completed. Note that, if the condition is not satisfied initially, the execution of the `WHILE` structure terminates immediately.

Figure 7.9 illustrates the use of a `WHILE` structure in specifying the operation of a digital PI controller. While the conversion in the reactor is below 0.95, the controller repeatedly goes through an inactive step of 5 time units (`CONTINUE` task), followed by a sampling and calculation step (`RESET` task), followed by a clipping and implementation step (`IF` structure).

```
# PROCESS SimpleSim

    UNIT
      T101 AS MixingTank

    SET # Parameter values
      WITHIN T101 DO
        NoComp := 3 ;
        NoInput := 2 ;
        ValveConstant := 10 ;
        CrossSectionalArea := 1 ; # m2
        Density := [ 950, 1000, 900 ] ; # kg/m3
      END # Within

    ASSIGN # Degrees of freedom
      WITHIN T101 DO
        # First inlet stream
        Fin(1) := 5 ;
        Xin(1,) := [ 0.12, 0.21, 0.67 ] ;
        # Second inlet stream
        Fin(2) := 15 ;
        Xin(2,) := [ 0.98, 0.01, 0.01 ] ;
        # Outlet stream valve fully closed
        ValvePosition := 0.0 ;
      END # Within

    INITIAL # Initial conditions
      WITHIN T101 DO
        HoldUp(1) = 1000 ;
        2 * Holdup(2) = HoldUp(3) ;
        TotalVolume = 1.5 ;
      END # Within

    SCHEDULE
      CONTINUE FOR 120
```

Figure 7.5: Mixing tank PROCESS

```
# PROCESS SeqSim

    ...

    SCHEDULE

      SEQUENCE

        # Fill up tank to 3.5 m3
        CONTINUE UNTIL T101.TotalVolume > 3.5

        # Turn off first inlet stream and
        #  increase the flowrate of the second by 50%
        RESET
          WITHIN T101 DO
            Fin(1) := 0 ;
            Fin(2) := 1.5 * OLD(Fin(2)) ;
          END # Within
        END # Reset

        # Fill up tank to 5 m3
        CONTINUE UNTIL T101.TotalVolume > 5.0

        # Turn off second inlet stream and
        #  open the outlet valve completely
        RESET
          WITHIN T101 DO
            Fin(2) := 0 ;
            ValvePosition := 1 ;
          END # Within
        END # Reset

        # Drain tank
        CONTINUE UNTIL T101.TotalVolume < 0.01

      END # Sequence
```

Figure 7.6: Mixing tank `PROCESS`—tasks in `SEQUENCE`

```
SCHEDULE

  PARALLEL

    # Operating policy for first inlet stream
    SEQUENCE
      # Fill up tank to 3.5 m3
      CONTINUE UNTIL T101.TotalVolume > 3.5
      # Turn off first inlet stream
      RESET
        T101.Fin(1) := 0 ;
      END # Reset
    END # Sequence

    # Operating policy for second inlet stream
    SEQUENCE
      # Fill up tank to 3.5 m3
      CONTINUE UNTIL T101.TotalVolume > 3.5
      # Increase the flowrate of the second inlet stream by 50%
      RESET
        T101.Fin(2) := 1.5 * OLD(T101.Fin(2)) ;
      END # Reset
      # Fill up tank to 5 m3
      CONTINUE UNTIL T101.TotalVolume > 5.0
      # Turn off second inlet stream and
      RESET
        T101.Fin(2) := 0 ;
      END # Reset
    END # Sequence

    # Operating policy for outlet stream
    SEQUENCE
      # Wait until both inlet streams have been turned off
      CONTINUE UNTIL ( T101.Fin(1) < 0.01 ) AND ( T101.Fin(2) < 0.01 )
      # Open the outlet valve completely
      RESET
        T101.ValvePosition := 1 ;
      END # Reset
      # Drain tank
      CONTINUE UNTIL T101.TotalVolume < 0.01
    END # Sequence

  END # Parallel
```

Figure 7.7: Mixing tank SCHEDULE—tasks in PARALLEL

```
SCHEDULE

  ...

  IF C101.ControlSignal > 1.0 THEN
    RESET V101.Position := 1.0 ; END
  ELSE
    IF C101.ControlSignal < 0.0  THEN
      RESET V101.Position := 0.0 ; END
    ELSE
      RESET V101.Position := OLD(C101.ControlSignal) ; END
    END # If
  END # If

  ...
```

Figure 7.8: Application of the `IF` conditional structure

```
SCHEDULE

  ...

  WHILE R101.Conversion < 0.95 DO
    SEQUENCE
      # Continuous operation
      CONTINUE FOR 5
      # Sampling and calculation
      RESET
        C101.Error := 150.0 - Sensor101.Measurement ;
        C101.IntegralError := C101.IntegralError + 5.0*C101.Error ;
        C101.ControlSignal := 0.5 + 1.2*(C101.Error +
                                     C101.IntegralError/20.0) ;
      END # Reset
      # Clipping and implementation
      IF C101.ControlSignal > 1.0 THEN
        RESET
          V101.Position := 1.0 ;
        END
      ELSE
        IF C101.ControlSignal < 0.0  THEN
          RESET
            V101.Position := 0.0 ;
          END
        ELSE
          RESET
            V101.Position := OLD(C101.ControlSignal) ;
          END
        END # If
      END # If
    END # Sequence
  END # While

  ...
```

Figure 7.9: Application of the WHILE iterative structure

7.3 More elementary tasks

7.3.1 The `STOP` and `MESSAGE` elementary tasks

`STOP` and `MESSAGE` are simple elementary tasks that may be used to halt a simulation and write a message to the screen respectively. The syntax for `STOP` and `MESSAGE` are:

```
STOP
```

and:

```
MESSAGE "text"
```

7.3.2 The `MONITOR` elementary task

Normally, during a gPROMS simulation the values of all variables at each reporting interval are sent to gRMS in order to be plotted. The `MONITOR` task may be used to restrict the amount of data sent to gRMS, and may be useful for a number of reasons:

- For large distributed models, which may consist of tens of thousands of variables, only a small proportion may be of particular importance and it may sometimes useful to prevent gPROMS from sending all of these variables to gRMS. One such example is chromatographic processes, where only the effluent profiles may be of importance. In this case, many of the variables are of secondary importance and could be suppressed from the gPROMS output.
- Restricting the output from gPROMS may useful in other circumstances: for example, periodic adsorption processes require many cycles of operation before a periodic steady state is achieved. If the modeller is only interested in the steady-state conditions, then the output from gPROMS may be disabled until the final cycle.
- Finally, restricting the data sent to gRMS results in much smaller files (gRMS files for large distributed models may require several Megabytes of storage).

The above situations can be dealt with in two ways, depending on whether variables should be suppressed permanently or only at certain times.

The `MONITOR` section of the `PROCESS` is used to specify which variables are to be monitored during the simulation; those that are not specified are permanently suppressed. If the `MONITOR` section is omitted, then all variables are monitored.

The syntax for the `MONITOR` section is as follows:

```
UNIT
  ...

MONITOR
  VariablePathPattern  ;
  ...

SET
  ...
```

where *VariablePathPattern* is the full pathname of the variable to be monitored. An asterisk can be used to specify that all components of an array or distributed variable are to be monitored. Some examples of *VariablePathPattern* are:

```
MONITOR
  aaa.bbb.x    ;
  aaa.bbb.y(*) ;
  aaa.x(1,*)   ;
  aaa.x(10,*)  ;
  aaa.y(*,*)   ;
```

Note that for distributed variables (*i.e.* those that depend on a `DISTRIBUTION_DOMAIN`), the indices must be integers and depend on the numerical method applied to the domain. For each distributed variable, gPROMS will generate an indexed variable with the same number of dimensions as the number of `DISTRIBUTION_DOMAIN`s that the variable depends on. The length of each dimension is equal to $NE \times O + 1$ for `OCFEM` and $NE + 1$ for finite difference methods, where NE is the number of elements and O is the order of the method. For example, if the variable `DV1` is defined by:

```
  DV1        AS DISTRIBUTION(x,y) OF NoType
```

where the `DISTRIBUTION_DOMAIN`s `x` and `y` are `SET` to the following methods:

```
  x := [ OCFEM, 3, 5 ] ;
  y := [ BFDM, 2, 20 ] ;
```

then the maximum values for the indices of `DV1` are 16 $(5 \times 3 + 1)$ and 21 $(20 + 1)$ respectively.

During the simulation, the output of all variables that are specified in the `MONITOR` section can be toggled using the `MONITOR` task. The syntax for the task is:

```
MONITOR ON
```

to enable monitoring and:

```
MONITOR OFF
```

to disable monitoring. An example of the `MONITOR` task is shown in figure 7.10, where the task "DoSomeCycles" operates the process until cyclic steady stage is achieved, then "DoOneCycle" operates the process for a single cycle after monitoring has been enabled.

```
SCHEDULE
  SEQUENCE
    MONITOR OFF
    DoSomeCycles ;
    MONITOR ON
    DoOneCycle ;
  END
```

Figure 7.10: Example of the `MONITOR` task.

7.3.3 The RESETRESULTS elementary task

The `RESETRESULTS` may be used in the `SCHEDULE` section of `PROCESS`es or `TASK`s to discard all previous data that was transmitted to a particular output channel. It may be called using one of the following commands:

```
RESETRESULTS gRMS
RESETRESULTS gPLOT
RESETRESULTS gExcelOutput
RESETRESULTS gUserOutput
RESETRESULTS ALL
```

Applying `RESETRESULTS` to any one of the four output channel options causes gPROMS to invoke a procedure, *gOCRESET*, that is provided by the corresponding output channel interface. In the three standard OCIs provided by gPROMS, this procedure erases all results that have been transmitted by gPROMS to the OCI up to this stage in the simulation; it then re-starts accumulating results from this point. This is particularly useful for very long simulations (*e.g.* in an operator training context) where it is desired occasionally (*e.g.* at the start of a new training exercise) to clear all past results and release the memory and/or disk space occupied by them. On the other hand, user-provided OCIs may decide to interpret the `RESETRESULTS` task in a different manner, by providing an appropriate implementation of the *gOCRESET* procedure.

7.3.4 The SAVE and RESTORE elementary tasks

Occasionally, it may be necessary to use the solution of one simulation in another. gPROMS provides the facility to `SAVE` the current values of all or some of the variables in a simulation and to `RESTORE` them in another simulation through the use of *Saved Variable Sets*. The syntax of the `SAVE` and `RESTORE` tasks are:

`SAVE` <*VarType*> "*V_Set_Name*"

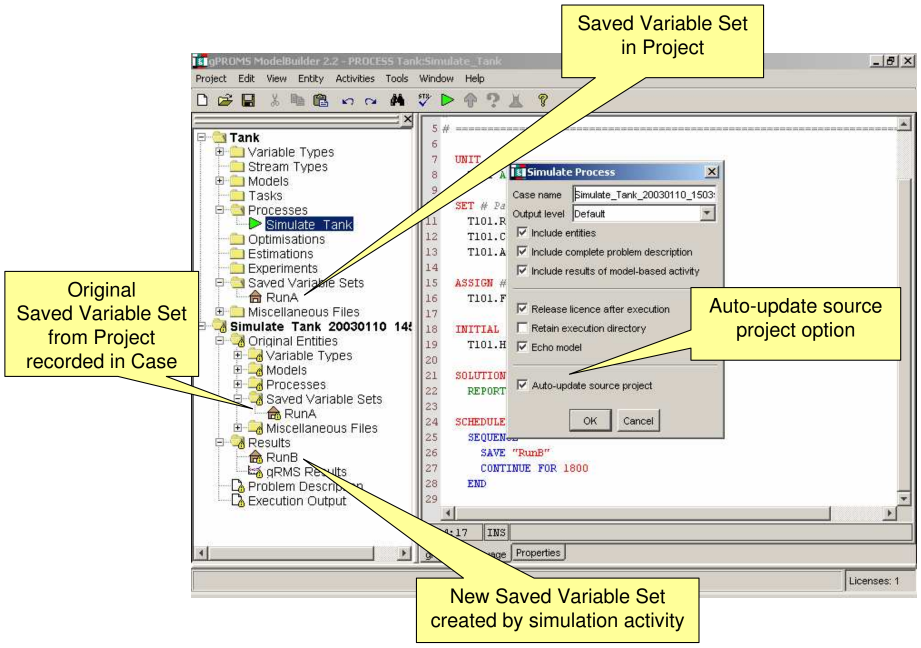
and

`RESTORE` <*VarType*> "*V_Set_Name*"

where the optional argument *VarType* can be one of `state`, `algebraic` and `input` (or any combination of these, separated by commas), and *V_Set_Name* is the name of the *Saved Variable Set* that the variables will be saved in (or restored from). If the *VarType* argument is omitted, then gPROMS will save the values of all variables; whereas arguments `state`, `algebraic` and `input` instruct gPROMS to save only the values of the state, algebraic or input variables respectively. The *VarType* optional argument works similarly with the `RESTORE` task.

Any new *Variable Set* created during a simulation activity using the `SAVE` elementary task will be stored in the "Results" Entity group within the Case folder. So that the *Variable Set* can be used in conjunction with the `RESTORE` elementary task, the Variable Set must then be copied into the gPROMS Project where it will appear in the *Saved Variable Sets* Entity group[1] as shown in figure 7.11.

[1]If they exceed the maximum file size large Saved Variable Sets created during a simulation activity will be stored as *Miscellaneous Files*. This behaviour is configured as part of the Users ModelBuilder Preferences and is discussed further in the ModelBuilder User Guide



Case name: Simulate_Tank_20030110_1503
Output level: Default
[✓] Include entities
[✓] Include complete problem description
[✓] Include results of model-based activity
[✓] Release licence after execution
[ ] Retain execution directory
[✓] Echo model
[✓] Auto-update source project

```
# ==========================================================================

UNIT
  A

SET # Pa
  T101.R
  T101.C
  T101.A

ASSIGN #
  T101.F

INITIAL
  T101.H

SOLUTION
  REPORT

SCHEDULE
  SEQUENCE
    SAVE "RunB"
    CONTINUE FOR 1800
  END
```

Figure 7.11: Saved Variables Sets in Projects and Cases

It is also possible to over-write (or modify) an existing Variable Set by using the SAVE elementary task on a Variable Set already present in the source project[2]. In this instance, ModelBuilder can be configured to automatically update the Variable Set in the Project. This is done by checking the "auto update source project" option in the execution dialog that can also be seen in figure7.11.

Variables may also be RESTOREd from multiple sets, separated by commas:

RESTORE "*v_set1*", "*v_set2*"

Figure 7.12 illustrates how the SAVE and RESTORE tasks could have been used in the example problem that was presented in the previous section. Rather than solving the problem in one PROCESS, with MONITOR used to show only the last cycle of operation, here we can use two PROCESSes: one to establish the cyclic steady state and to save the variables, and the second to simulate a single cycle using as initial conditions the values of the variables at the end of the simulation of the first PROCESS. One advantage of this approach is that the variables can be plotted against the *local* time for the cycle (*i.e.* the cycle starts at time = 0) as opposed to the global time of the whole simulation (where the start of the cycle will be at some time greater than zero).

[2]In this instance, two Variable Sets with the same name will appear in the Case: the original (from the project) will appear in the "Original Entities" and the new entity created by the SAVE elementary task will be stored under "Results"

```
SCHEDULE
  SEQUENCE
    MONITOR OFF
    DoSomeCycles ;
    SAVE state "CyclicSteadyState"
  END
```

(a) `SCHEDULE` from first `PROCESS`

```
SCHEDULE
  SEQUENCE
    MONITOR OFF
    RESTORE state "CyclicSteadyState"
    MONITOR ON
    DoOneCycle ;
  END
```

(b) `SCHEDULE` from second `PROCESS`

Figure 7.12: Application of the `SAVE` and `RESTORE` Tasks.

Chapter 8

Complex Operating Procedures

Contents

8.1 TASKs

A `TASK` is a model of an operating procedure; it corresponds to the fourth entry down in a generic gPROMS project tree (see figure 2.5).

`TASK`s are created within the gPROMS ModelBuilder environment in a similar manner to `MODEL`s (*cf.* section 2.2.3), the only (obvious) difference being that `TASKS` is chosen for `Entity Types` instead of `MODELS`.

Like a `MODEL`, a `TASK` is split into sections, containing different pieces of information. Overall, the structure of a `TASK` declaration is the following:

```
PARAMETER
   ... Parameter declarations ...
VARIABLE
   ... Local variable declarations ...
SCHEDULE
   ... Schedule declaration ...
```

In the next two sections we take a detailed look at the `PARAMETER`, `VARIABLE` and `SCHEDULE` sections.

8.1.1 The `VARIABLE` and `SCHEDULE` sections

The `VARIABLE` section is used to declare local variables. These can be of type `INTEGER` or `REAL`. They can only be used within the `TASK` in which they are declared.

The `SCHEDULE` section defines the operating policy implemented by the `TASK`. It is similar to the `SCHEDULE` section in `PROCESS`es, the only difference being that it has access to the local variables declared in the `VARIABLE` section. The values of the latter can be manipulated by using assignment statements.

Figure 8.1 shows a `TASK` that models the action of a digital PI controller. Three local variables are declared in the `VARIABLE` section, namely `Error`, `IntegralError` and `ControlSignal`. In the `SCHEDULE` section, an assignment statement initialises `IntegralError` to zero. Then, the repeated action of the controller is specified within a `WHILE` structure. This is executed until a termination condition is satisfied (in this case, when 1000 time units have passed on the simulation clock). The action of the controller is itself a sequence of elementary tasks. First, a `CONTINUE` task is used to enforce a period of undisturbed operation (5 time units). After this, sampling takes place. The signal of a temperature sensor is used to update the controller `Error` and `IntegralError` and a `ControlSignal` is calculated. An `IF` structure is used to clip the signal which is then implemented it through a `RESET` task.

In essence, `TASK`s are user-defined tasks. Once declared, they are equivalent to elementary tasks and can be used in `PROCESS SCHEDULE`s or even within the `SCHEDULE`s of other `TASK`s. For instance,

```
SCHEDULE
  PARALLEL
```

```
# TASK DigitalPI

    VARIABLE
      Error, IntegralError, ControlSignal AS REAL

    SCHEDULE
      SEQUENCE
        IntegralError := 0 ;
        WHILE TIME < 1000 DO
          SEQUENCE
            CONTINUE FOR 5.0
            Error := 150.0 - Sensor101.Measurement ;
            IntegralError := IntegralError + 5.0*Error ;
            ControlSignal := 0.5 + 1.2 * ( Error + IntegralError/20.0 ) ;
            IF ControlSignal > 1.0 THEN
              RESET Valve201.Position := 1.0 ; END
            ELSE
              IF ControlSignal < 0.0  THEN
                RESET Valve201.Position := 0.0 ; END
              ELSE
                RESET Valve201.Position := OLD(ControlSignal) ; END
              END # If
            END # If
          END # Sequence
        END # While
      END # Sequence
```

Figure 8.1: TASK for a digital PI control law

```
    DigitalPI
    CONTINUE FOR 1000
  END # Parallel
```

executes the digital control law in parallel with the operation of the rest of the process.

8.1.2 The PARAMETER section

The TASK shown in figure 8.1, although useful for grouping a series of elementary tasks together, has a big disadvantage: it is extremely specific. First of all, it refers to a unique sensor/valve pair, Sensor101 and Valve201 respectively. Moreover, the sampling interval (5 time units) and controller tuning parameters (150.0, 0.5, 1.2, 20.0) are expressed as constant values. Finally, the task always terminates after 1000 time units have elapsed and is thus appropriate for a simulation of that length only. If it were necessary to apply the same operating procedure to a different sensor/valve pair possibly using different tuning parameters for the controller, a new TASK would have to be declared. This is clearly unsatisfactory. In most instances, we want to be able to declare TASKs that are independent of the details of an individual simulation.

For instance, we want to be able to define a TASK that switches "a" pump on and another that switches "a" pump off. "A" is used to indicate that the actual pump on which the TASKs act remains unspecified until the moment they are used in a particular simulation experiment. Similarly, we want to be able to define a TASK for "a" digital controller and only specify the sensor/valve pair it uses and the values for its tuning parameters when the TASK is actually used in a simulation experiment.

This is achieved by using TASK parameters. Upon declaration, a TASK can be parameterised with respect to an arbitrary number of parameters. The actual values of these parameters have to be specified only when the TASK is actually used in a specific simulation experiment.

TASK parameters are declared in the PARAMETER section. Declared parameters may be of any of the following types:

- INTEGER, REAL or LOGICAL constants. These are used to parameterise a TASK with respect to, for instance, controller tuning parameters, event durations *etc.*
- INTEGER_EXPRESSION, REAL_EXPRESSION or LOGICAL_EXPRESSION. These are used to parameterise a TASK with respect to, for instance, logical conditions for the conditional and iterative structures *etc.*
- MODEL. These are used to parameterise a TASK with respect to the actual MODELs on which it acts.

For example, figure 8.2 shows a task that switches a pump on. Once this TASK has been defined, it can be used in a SCHEDULE section. For instance,

```
SCHEDULE
  ...
  SwitchPumpOn(Pump IS Plant.P201)
  ...
```

will switch on pump Plant.P201, while

```
# TASK SwitchPumpOn

    PARAMETER
      Pump AS MODEL GenericPump

    SCHEDULE
      RESET
        Pump.Status := Pump.Open ;
      END # Reset
```

Figure 8.2: `TASK` to switch on a pump

```
SCHEDULE
  ...
  SEQUENCE
    SwitchPumpOn(Pump IS Plant.P205)
    SwitchPumpOn(Pump IS Plant.P206)
    SwitchPumpOn(Pump IS Plant.P207)
  END # Sequence
  ...
```

will, in sequence, switch on pumps `Plant.P205`, `Plant.P206` and `Plant.P207`.

Note that, when executing a `TASK` that contains parameters, the proper list of arguments must be given along with the name of the `TASK`. `TASK`s that contain parameters can be thought of as the equivalent of subroutines or functions in high-level programming languages. Variables declared in the `VARIABLE` section are the equivalent of local subroutine variables. On the other hand, the `PARAMETER` section is the equivalent of a function prototype. It defines the number and type of arguments that a `TASK` accepts as arguments. A "call" to the `TASK` then includes a list of items which must be in the same order, and of the same number and type as the ones in the `TASK`'s parameter list.

Figure 8.3 presents the correct version of the digital controller `TASK` of figure 8.1. The parameters include real constants that determine the various tuning parameters and the sampling interval, a logical expression that determines the termination of control, and model parameters that determine the sensor/valve pair on which the controller is used.

This `TASK` is much more reusable. It can be used for any sensor/valve pair in a simulation experiment and different tuning parameters for the controller can be specified without rewriting the `TASK`. For instance,

```
SCHEDULE
  ...
  PARALLEL
    DigitalPI
      ( SetPoint            IS 150.0,
        Bias                IS 0.5,
        Gain                IS 1.2,
        IntegralTime        IS 20.0,
        SamplingInterval    IS 5.0,
# TASK DigitalPI

    PARAMETER
      SetPoint, Bias, Gain, IntegralTime  AS REAL
      SamplingInterval                    AS REAL
      TerminationCondition                AS LOGICAL_EXPRESSION
      Sensor                              AS MODEL GenericSensor
      Valve                               AS MODEL GenericValve

    VARIABLE
      Error, IntegralError, ControlSignal AS REAL

    SCHEDULE
      SEQUENCE
        IntegralError := 0 ;
        WHILE NOT TerminationCondition DO
          SEQUENCE
            CONTINUE FOR SamplingInterval
            Error := SetPoint - Sensor.Measurement ;
            IntegralError := IntegralError + SamplingInterval*Error ;
            ControlSignal := Bias + Gain*( Error +
                                 IntegralError/IntegralTime ) ;
            IF ControlSignal > 1.0 THEN
              RESET Valve.Position := 1.0 ; END
            ELSE
              IF ControlSignal < 0.0  THEN
                RESET Valve.Position := 0.0 ; END
              ELSE
                RESET Valve.Position := OLD(ControlSignal) ; END
              END # If
            END # If
          END # Sequence
        END # While
      END # Sequence
```

Figure 8.3: Parameterised TASK for a digital PI control law

```
          TerminationCondition IS Plant.T101.TotalVolume > 5.0,
          Sensor               IS Plant.Sensor101,
          Valve                IS Plant.Valve205 )
      DigitalPI
        ( SetPoint             IS 10.0,
          Bias                 IS 0.8,
          Gain                 IS 2.6,
          IntegralTime         IS 50.0,
          SamplingInterval     IS 1.0,
          TerminationCondition IS Plant.R101.Temperature > 80.0,
          Sensor               IS Plant.Sensor103,
          Valve                IS Plant.Valve207 )
    END # Parallel
    ...
```

will initiate two digital control procedures in parallel, acting on two different sensor/valve pairs. The two procedures also have different controller tuning characteristics and a different logical expression determining their termination.

8.2 Hierarchical sub-task decomposition

A complex operation on one or more items of process equipment can usually be decomposed into lower-level, simpler operations. Each of the lower-level operations may in turn be decomposed in other, more primitive operations, the decomposition continuing until all operations can be described in terms of elementary manipulations of the underlying models made possible by elementary tasks. Similarly to hierarchical sub-model decomposition, this *hierarchical sub-task decomposition* defeats complexity by restricting the scope of the problem considered at any point to a manageable level.

Hierarchical sub-task decomposition in gPROMS is possible because, as was mentioned before, previously declared TASKs may be used within other, higher-level TASKs and is greatly facilitated by the fact that suitably parameterised TASKs may be reused several times in different parts of an operation.

```
# TASK OperateReactor

  PARAMETER
    Reactor              AS MODEL StirredReactor
    SR                   AS REAL
    StartTemperature     AS REAL
    Terminationcondition AS LOGICAL_EXPRESSION

  SCHEDULE
    SEQUENCE
      RESET
        Reactor.SteamRate := SR ;
      END
      CONTINUE UNTIL Reactor.Temperature > StartTemperature
      RESET
        Reactor.SteamRate := 0 ;
      END
      CONTINUE UNTIL TerminationCondition
    END # Sequence
```

Figure 8.4: Low-level TASK to operate a reactor

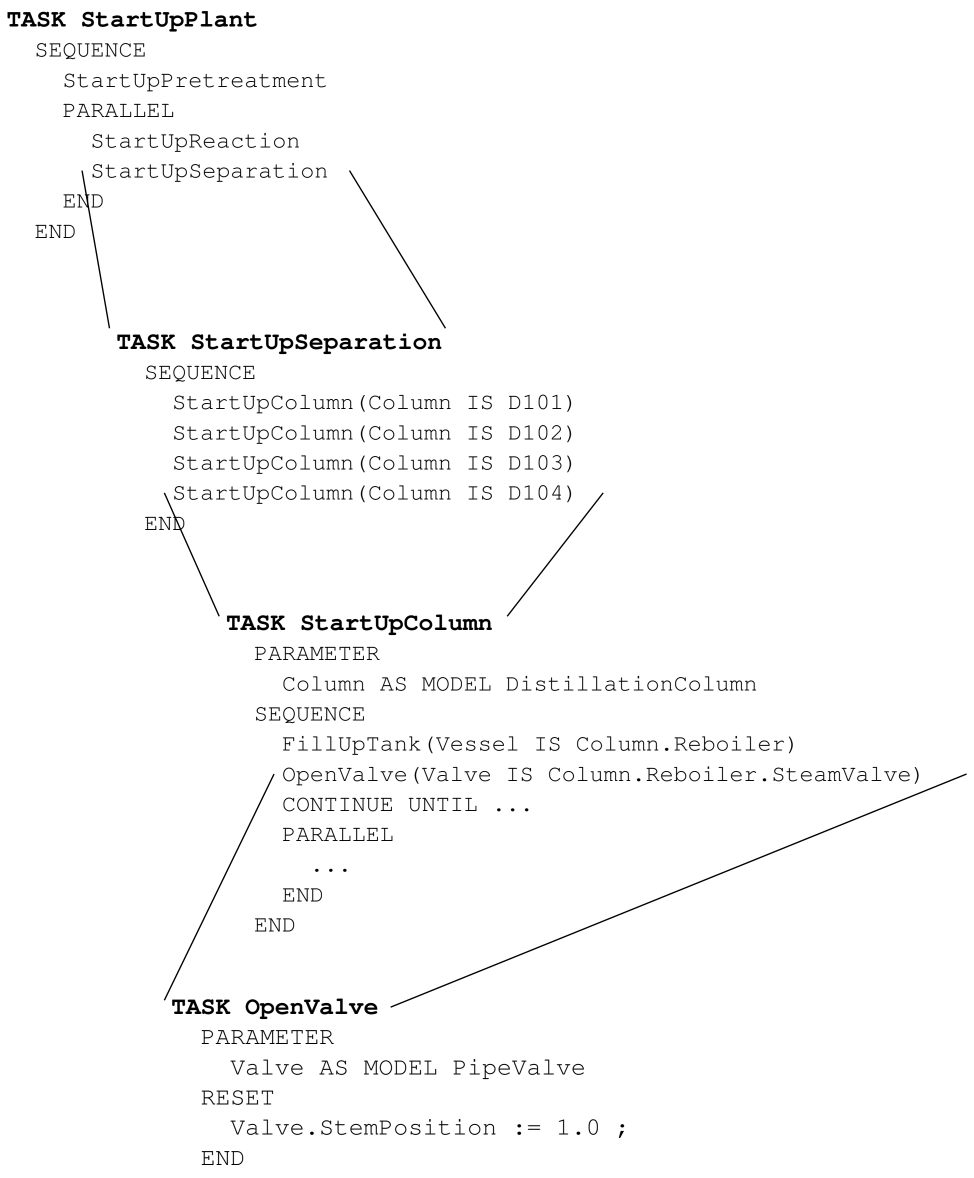
Consider, for instance, the OperateReactor TASK of figure 8.4. It specifies an operating procedure for performing a reaction in a reactor of type StirredReactor (a parameter of the TASK). The operating procedure is simple, involving the execution of four elementary tasks in sequence. First, the steam supply rate to the reactor is set to a value SR. Operation then continues until the temperature in the reactor has exceeded a predefined limit StartTemperature. Finally, the steam supply is cut off and operation continues until a TerminationCondition is satisfied. In addition to the reactor unit, SR, StartTemperature and TerminationCondition are also parameters of the TASK, of type REAL, REAL and LOGICAL_EXPRESSION respectively.

Figure 8.5 then illustrates how the OperateReactor TASK is used to define a higher-level TASK, namely OperateReactionTrain. Two OperateReactor tasks are invoked in parallel to model the operation of two reactors. Parameterisation also permits the specification of different values for the operating parameters of the two reactors.

```
# TASK OperateReactorTrain

  PARAMETER
    Plant AS MODEL ReactorTrain

  SCHEDULE
    PARALLEL
      PerformReaction
        ( Reactor              IS Plant.R1,
          SR                   IS 35.3,
          StartTemperature     IS 70.0,
          TerminationCondition IS Plant.R1.Conversion(1) > 0.95 )
      PerformReaction
        ( Reactor              IS Plant.R2,
          SR                   IS 10.0,
          StartTemperature     IS 40.0,
          TerminationCondition IS Plant.R2.Temperature > 120.0 )
    END # Parallel
```

Figure 8.5: High-level `TASK` to operate a reactor train

A more complicated situation is depicted in figure 8.6 where the startup of an entire plant is modelled by hierarchically decomposing the higher-level operations into lower-level ones through the use of nested and suitably parameterised `TASK`s.



```
TASK StartUpPlant
  SEQUENCE
    StartUpPretreatment
    PARALLEL
      StartUpReaction
      StartUpSeparation
    END
  END
```

```
TASK StartUpSeparation
  SEQUENCE
    StartUpColumn(Column IS D101)
    StartUpColumn(Column IS D102)
    StartUpColumn(Column IS D103)
    StartUpColumn(Column IS D104)
  END
```

```
TASK StartUpColumn
  PARAMETER
    Column AS MODEL DistillationColumn
  SEQUENCE
    FillUpTank(Vessel IS Column.Reboiler)
    OpenValve(Valve IS Column.Reboiler.SteamValve)
    CONTINUE UNTIL ...
    PARALLEL
      ...
    END
  END
```

```
TASK OpenValve
  PARAMETER
    Valve AS MODEL PipeValve
  RESET
    Valve.StemPosition := 1.0 ;
  END
```

Figure 8.6: Hierarchical sub-task decomposition

Chapter 9

Stochastic Simulation in gPROMS

Contents

In this chapter, we discuss how one can perform stochastic simulations in gPROMS.

The first part of the chapter describes the probability functions that are supported and how to assign values sampled from a probability function to `PARAMETER`s and `VARIABLE`s.

The second part of the chapter describes the modelling techniques used to perform stochastic simulations and to generate results in the form of probability density functions.

Function	Arguments	Example
Uniform	*lower*, *upper*	`UNIFORM(0,1)` returns a uniformly distributed number in the range [0,1]. *lower* $<$ *upper*.
Triangular	*lower*, *mode*, *upper*	`TRIANGULAR(1,2,4)` returns a number sampled from a triangular distribution with mode 2, lower limit 1 and upper limit 4. *lower* $<$ *mode* $<$ *upper*.
Normal	*mean*, *stddev*	`NORMAL(3,0.25)` returns a value sampled from a normal distribution with mean 3 and standard deviation 0.25. *stddev* > 0.
Gamma	*alpha*, *beta*	`GAMMA(3,1)` returns a value from the Gamma distribution. *alpha, beta* > 0.
Beta	*alpha*, *beta*, *lower*, *upper*	`BETA(1.5,5,0,1)` returns a value from the Beta distribution. *alpha, beta* > 0; *lower* $<$ *upper*.
Weibull	*alpha*, *beta*	`WEIBULL(4,1)` returns a value from the Weibull distribution. *alpha, beta* > 0.

Table 9.1: Probability distribution functions available in gPROMS.

9.1 Assigning random numbers to `PARAMETER`s and `VARIABLE`s

`PARAMETER`s and `VARIABLE`s can be given random values in the `SET` and `ASSIGN` sections, respectively, of a `PROCESS` in the same way that they are given deterministic values.

Instead of assigning a literal value or expression to the parameter or variable, special functions are used that return values sampled from the distribution. The syntax is shown below.

```
...

  SET
    Identifier   :=  DistributionFunction(ArgList) ;
...

  ASSIGN
    Identifier   :=  DistributionFunction(ArgList) ;
...
```

The functions *DistributionFunction* each require a different set of arguments (*ArgList*). The available functions are described in table 9.1.

Once `PARAMETER`s and `VARIABLE`s have been given stochastic values, they behave exactly as though they had been assigned deterministic values: *i.e.* `PARAMETER`s remain constant and are not calculated by a simulation and the `VARIABLE`s remain at their `ASSIGN`ed values unless reassigned in a `RESET` statement. Any `VARIABLE` may be `RESET`, as usual, using a literal or an expression (using the `OLD` operator if this involves other `VARIABLE`s—see chapter 7) or by assigning a new random number.

Note that `VARIABLE`s and `PARAMETER`s can only ever be assigned "point" values; they are not assigned distributions. Furthermore, each time a variable is `RESET` to a value from a distribution, it is given a different value, even if the distribution function's arguments are the same. For example, in the following segment of a `SCHEDULE` section, the variable `Random` is assigned two different values from the Normal distribution: one time it may be assigned 0.23 then 0.07; another time it may be assigned 0.01 then 0.36.

```
...

SCHEDULE
  SEQUENCE
    ...
    RESET
      Random := NORMAL(0,1) ;
    END
    ...
    RESET
      Random := NORMAL(0,1) ;
    END
    ...
  END
```

One further issue that should be noted is that gPROMS will always seed the random number generator with the same number each time gPROMS is started (*i.e.* each time you run gPROMS from the command prompt; not each time you execute a `PROCESS`). This means that the results of a stochastic simulation can be reproduced if you execute a process directly after starting a gPROMS session, then end gPROMS and start again.

9.2 Plotting results of multiple stochastic simulations

The method outlined above is somewhat limited in use, since the results of the simulations are only for a specific realisation of the random input VARIABLEs. There is no way directly to examine how the distributions of the output VARIABLEs are influenced by the distributions of the input VARIABLEs, which is the prime reason for considering stochastic simulation.

In this section, we will describe methods to do precisely this. In the first part, we will describe how you can combine multiple simulations in a single process and then use the results to evaluate metrics (such as the mean, variance, etc.) of the output VARIABLEs. In the second part, we will outline a method that allows you to plot the probability density functions of the output VARIABLEs.

9.2.1 Combining multiple simulations

Given a model with some uncertain inputs, a series of simulations can be combined into a single process by introducing a new higher-level model. The original model is included in the new one as an array of UNITs, as shown below.

```
# MODEL ModelUncertain

  ...

  VARIABLE
    Input         AS InputVarType    # uncertain input variable
    Output        AS OutputVarType   # important output variable

  ...



# MODEL Combined

  PARAMETER
    NoScenarios   AS                       INTEGER
    InputMean     AS                       REAL
    InputStdDev   AS                       REAL

  UNIT
    Scenarios     AS ARRAY(NoScenarios) OF ModelUncertain
```

The input VARIABLEs for each scenario of ModelUncertain then need to be specified as follows.

```
# PROCESS StochSim

  UNIT
    StSim  AS Combined
  ...

  ASSIGN
    WITHIN StSim DO
      FOR i := 1 TO NoScenarios DO
        Scenarios(i).Input := NORMAL(InputMean,InputStdDev) ;
      END
    END

  ...
```

Note that each input variable is ASSIGNed a different value from the same distribution. This could also have been done with PARAMETERs, but parameter propagation cannot be used in this way: this would result in all PARAMETERs being set the same value, because the parameter in the higher-level model will be assigned randomly and then that particular value will be propagated to the scenarios.

Of course, any inputs that are common to the system (such as design constants or precisely known operating PARAMETERs) can be included in the higher-level model along with equations linking them to the scenarios. This reduces the complexity of the SCHEDULE section.

9.2.2 Plotting probability density functions

Now all of the scenarios are together in one process, but it is not easy to plot them together in one graph. Rather than having to select each variable from each scenario instance, it would be much easier to be able to select the whole distribution. This can easily be done as follows.

```
# MODEL Combined

  PARAMETER
    NoScenarios   AS                       INTEGER
    InputMean     AS                       REAL
    InputStdDev   AS                       REAL

  UNIT
    Scenarios     AS ARRAY(NoScenarios) OF ModelUncertain

  VARIABLE
    Input         AS ARRAY(NoScenarios) OF InputVarType
    Output        AS ARRAY(NoScenarios) OF OutputVarType

    OPmean        AS                       NoType # mean of Output
    OPvariance    AS                       NoType # variance of Output

  EQUATION
    FOR i := 1 TO NoScenarios DO
      Input(i) = Scenarios(i).Input ;
      Output(i) = Scenarios(i).Output ;
    END
    OPmean = SIGMA(Output)/NoScenarios ;
    OPvariance = SIGMA( (Output - OPmean)^2 )/NoScenarios ;
```

Now, all of the scenarios can be plotted on a single graph by selecting a single variable in gRMS. Notice also that the mean and variance of the output can easily be calculated.

Finally, it is often useful to be able to plot the probability density function (pdf) of the output. In general, for each variable this requires two new `VARIABLE`s a distribution domain and three `PARAMETER`s. However, if two `VARIABLE`s are in the same interval they can share the same distribution domain and `PARAMETER`s. This is shown below.

```
# MODEL Combined

  PARAMETER
    NoScenarios   AS  INTEGER
    InputMean     AS  REAL
    InputStdDev   AS  REAL

    NoInt_OP      AS  INTEGER DEFAULT 20 # number of intervals for distribution
    Upper_OP      AS  REAL    DEFAULT 3  # upper bound on distribution
    Lower_OP      AS  REAL    DEFAULT 1  # lower bound on distribution

  DISTRIBUTION_DOMAIN
    Dist_OP       AS (Lower_OP:Upper_OP) # distribution over which
                                         # Output will be plotted
  UNIT
    Scenarios     AS ARRAY(NoScenarios) OF ModelUncertain

  VARIABLE
    Input         AS ARRAY(NoScenarios) OF InputVarType
    Output        AS ARRAY(NoScenarios) OF OutputVarType

    OPmean        AS                       NoType # mean of Output
    OPvariance    AS                       NoType # variance of Output

# temp variable to count occurrences of Output in a particular interval:
    OPacc         AS DISTRIBUTION(Dist_OP,NoScenarios) OF NoType

# pdf function for Output:
    OP_pdf        AS DISTRIBUTION(Dist_OP) OF             NoType

  EQUATION
    FOR i := 1 TO NoScenarios DO
      Input(i) = Scenarios(i).Input ;
      Output(i) = Scenarios(i).Output ;
    END

    OPmean = SIGMA(Output)/NoScenarios ;
    OPvariance = SIGMA( (Output - OPmean)^2 )/NoScenarios ;
    FOR i := Lower_OP TO Upper_OP DO
      FOR j := 1 TO NoScenarios DO
        IF i - (Upper_OP-Lower_OP)/NoInt_OP/2 <= Output(j) AND
           Output(j) < i + (Upper_OP-Lower_OP)/NoInt_OP/2 THEN
          OPacc(i,j) = 1 ;
        ELSE
          OPacc(i,j) = 0 ;
        END
      END
      OP_pdf(i) = SIGMA(OPacc(i,))/NoScenarios ;
    END
```

The three new PARAMETERs introduced are NoInt_OP, Lower_OP and Upper_OP. These PARAMETERs define the distribution over which the output variable will be plotted. NoInt_OP is the number of intervals used for the distribution Dist_OP. This will obviously be used to set Dist_OP:

```
# PROCESS StSim

  ...

  SET
    ...
    Dist_OP := [BFDM, 1, NoInt_OP] ;
```

The first order, backward finite difference method is all that is required because the distribution domain is essentially behaving as an array.

The PARAMETERs Lower_OP and Upper_OP are simply the lower and upper bounds on the domain Dist_OP. The variable being plotted, Output in this case, must lie in the interval [lower, upper] for each scenario; otherwise, the pdf will become distorted. If the lower and upper bounds are set so that a number of VARIABLEs lie in this interval, then all of these VARIABLEs can be plotted using the same distribution. The number of intervals should be set appropriately so that the distribution is not too coarse.

Finally, the two VARIABLEs introduced are OPacc and OP_pdf. Each variable being plotted will need its own pair of VARIABLEs. OPacc(i,j) is set to 1 if the value of Output(j) (the value in scenario j) lies in interval i of the distribution domain. OP_pdf(i) therefore represents the number of scenarios in which Output has a value in interval i. This is divided by the number of scenarios to normalise the pdf.

Plotting the pdf of a state variable can slow down the simulation significantly (due to the many discontinuities, and therefore re-initialisations, encountered as the values of the VARIABLEs switch between intervals). This can be remedied by introducing one further variable, as illustrated below for the example already considered:

```
# MODEL Combined

  ...
  VARIABLE
    ...

# temporary Output variable = 0 until end of simulation, then = Output
    OutputEnd     AS ARRAY(NoScenarios) OF OutputVarType

    ...

  EQUATION
    FOR i := 1 TO NoScenarios DO
      Input(i) = Scenarios(i).Input ;
      Output(i) = Scenarios(i).Output ;
    END

    ...

    FOR i := Lower_OP TO Upper_OP DO
      FOR j := 1 TO NoScenarios DO
        IF i - (Upper_OP-Lower_OP)/NoInt_OP/2 <= OutputEnd(j) AND
           OutputEnd(j) < i + (Upper_OP-Lower_OP)/NoInt_OP/2 THEN
          OPacc(i,j) = 1 ;
        ELSE
          OPacc(i,j) = 0 ;
        END
      END
      OP_pdf(i) = SIGMA(OPacc(i,))/NoScenarios ;
    END



# PROCESS StochSim
  ...

  MONITOR
    StSim.Output(*) ;
    StSim.Input(*) ;
    StSim.OPmean ;
    StSim.OPvariance ;
    StSim.OP_pdf ;

  ...

  ASSIGN
    WITHIN StSim DO
      OutputEnd := 0 ;
      ...
    END

  ...

  SCHEDULE
```

```
SEQUENCE
  ...
  RESET
    StSim.OutputEnd := OLD(StSim.Output) ;
  END
END
```

So, `OutputEnd` is 0 throughout the simulation and the `IF` conditions are only evaluated at initialisation. Only at the end of the simulation is `OutputEnd` changed, at which point all of the `IF` statements are re-evaluated and `OP_pdf` recalculated.

Finally, note that the output has been restricted to only those `VARIABLE`s of importance by using the `MONITOR` section. This reduces the amount of data sent to the output channel (*e.g.* gRMS). Although this is not necessary, it recommended for moderate to large problems (even small problems output large quantities of data when the number of scenarios is large).

9.3 Example

In this section we illustrate the above techniques using a simple model of an isothermal batch reaction. The following reactions occur in the reactor, D being the desired product.

$$\mathrm{A} + \mathrm{B} \rightarrow \mathrm{C} \rightarrow \mathrm{D}$$

The reactor initially contains $10\mathrm{m}^3$ of an equimolar mixture of A and B. The temperature is held constant at 353K and the reaction is allowed to progress for 1 hour. The reaction rates are assumed to follow Arrhenius's law.

A simple, generic model for an isothermal liquid-phase CSTR is used to model the process and is shown overleaf.

```
# MODEL LiquidPhaseCSTR

  PARAMETER
    # Number of components
    NoComp              AS                              INTEGER

    # Number of reactions
    NoReac              AS                              INTEGER

    Density             AS ARRAY(NoComp) OF             REAL

    # Reaction data (Arrhenius law)
    ArrhConstant        AS ARRAY(NoReac) OF             REAL
    ActivationEnergy    AS ARRAY(NoReac) OF             REAL

    # Reaction orders
    Order               AS ARRAY(NoComp,NoReac) OF      INTEGER

    # Component stoichiometric coefficients
    Nu                  AS ARRAY(NoComp,NoReac) OF      INTEGER

    # Gas constant
    R                   AS                              REAL

  VARIABLE
    Fin                 AS                              MolarFlowrate
    Xin                 AS ARRAY(NoComp) OF             MolarFraction
    Fout                AS                              MolarFlowrate
    X                   AS ARRAY(NoComp) OF             MolarFraction
    HoldUp              AS ARRAY(NoComp) OF             Moles
    C                   AS ARRAY(NoComp) OF             MolarConcentration
    T                   AS                              Temperature
    TotalHoldup         AS                              Moles
    TotalVolume         AS                              Volume
    ReactionConstant    AS ARRAY(NoReac) OF             NoType
    Rate                AS ARRAY(NoReac) OF             NoType

  EQUATION

    # Material balance
    FOR i := 1 TO NoComp DO
      $HoldUp(i) = Fin*Xin(i) - Fout*X(i) + TotalVolume*SIGMA(Nu(i,)*Rate) ;
    END

    # Reaction rates
    FOR j := 1 TO NoReac DO
      ReactionConstant(j) = ArrhConstant(j) * EXP(-ActivationEnergy(j)/R/T) ;
      Rate(j) = ReactionConstant(j) * PRODUCT(C^Order(,j)) ;
    END

    # Total volume and total holdup
    TotalVolume = SIGMA(Holdup/Density) ;

    TotalHoldup = SIGMA(HoldUp) ;

    # Molar fractions and concentrations
    Holdup = X * TotalHoldup ;

    Holdup = C * TotalVolume ;
```

The results of a deterministic simulation of the above model is shown in figure 9.1 (the rate `PARAMETER`s were chosen such that the mole fraction of D was about 0.9).

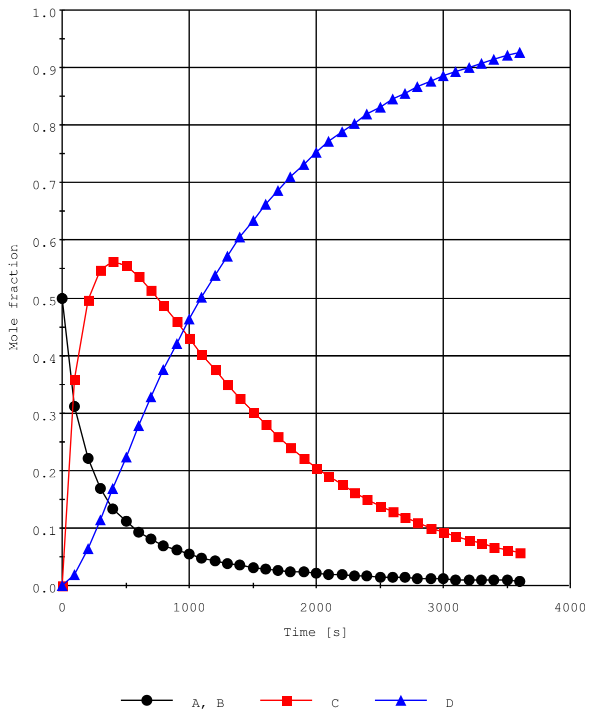


Figure 9.1: Mole fraction profiles for the deterministic simulation.

Now we assume the that temperature of the reaction may change from batch to batch. This can be modelled using a normal distribution with a mean of 353K and a standard deviation of 2K. To see the effect of this, we need to introduce a new model to include a number of scenarios. This is shown overleaf.

```
# MODEL Stochastic_LiquidPhaseCSTR

  PARAMETER

#   define common PARAMETERs here so their values can be propagated
#   to each scenario
# -----------------------------------------------------------------
    NoComp              AS                          INTEGER
    NoReac              AS                          INTEGER
    Density             AS ARRAY(NoComp) OF         REAL
    ArrhConstant        AS ARRAY(NoReac) OF         REAL
    ActivationEnergy    AS ARRAY(NoReac) OF         REAL
    Order               AS ARRAY(NoComp,NoReac) OF  INTEGER
    Nu                  AS ARRAY(NoComp,NoReac) OF  INTEGER
    R                   AS                          REAL
# -----------------------------------------------------------------

    NoScenarios   AS                        INTEGER

    NoInt_PMF     AS                        INTEGER DEFAULT 20
    Upper_PMF     AS                        REAL    DEFAULT 1
    Lower_PMF     AS                        REAL    DEFAULT 0.8

  DISTRIBUTION_DOMAIN
    Dist_PMF      AS (Lower_PMF:Upper_PMF)

  UNIT
    Scenarios     AS ARRAY(NoScenarios) OF LiquidPhaseCSTR

  VARIABLE
    T             AS ARRAY(NoScenarios) OF Temperature
    ProdMolFrac   AS ARRAY(NoScenarios) OF MolarFraction

    PMFmean       AS                        NoType
    PMFvariance   AS                        NoType
    PMFstddev     AS                        NoType

# temp variable to count occurrences of Output in a particular interval:
    PMFacc         AS DISTRIBUTION(Dist_PMF,NoScenarios) OF NoType

# pdf function for Output:
    PMF_pdf        AS DISTRIBUTION(Dist_PMF) OF             NoType

  EQUATION
    FOR i := 1 TO NoScenarios DO
      T(i) = Scenarios(i).T ;
    END

    PMFmean = SIGMA(ProdMolFrac)/NoScenarios ;
    PMFvariance = SIGMA( (ProdMolFrac - PMFmean)^2 )/NoScenarios ;
    FOR i := Lower_PMF TO Upper_PMF DO
      FOR j := 1 TO NoScenarios DO
        IF i - (Upper_PMF-Lower_PMF)/NoInt_PMF/2 <= ProdMolFrac(j) AND
           ProdMolFrac(j) < i + (Upper_PMF-Lower_PMF)/NoInt_PMF/2 THEN
          PMFacc(i,j) = 1 ;
        ELSE
          PMFacc(i,j) = 0 ;
        END
      END
      PMF_pdf(i) = SIGMA(PMFacc(i,))/NoScenarios ;
    END
```

As before, we have defined new distributed `VARIABLE`s to contain the values of the `VARIABLE`s of interest in each scenario. These are `T` for the temperature and `ProdMolFrac` for the mole fraction of the product D (*i.e.* x_4). Again, `PARAMETER`s are defined that describe the upper and lower limits of the distribution and its coarseness. Finally, `VARIABLE`s are defined for the mean, variance and standard deviation of the product mole fraction.

The equations are the same as were described before, except that there is no equation for the standard deviation or to relate the variable `ProdMolFrac` to the `X(4)` `VARIABLE`s in each scenario.

While we could include the equation for the standard devation in this model, by using the equation:

```
    PMFstddev^2 = PMFvariance ;
```

this tends to slow the simulation down. The alternative used here is to `ASSIGN` `PMFstddev` to a temporary value in the `PROCESS` section and to `RESET` it at the end of the simulation using:

```
  SCHEDULE
    SEQUENCE
      ...
      RESET
        xxx.PMFstddev := SQRT(OLD(xxx.PMFvariance)) ;
      END
    END
```

The final difference is the missing equation relating `ProdMolFrac` to `Scenarios().X(4)`. This is because we are plotting a pdf of a dynamic variable and want to avoid slowing the simulation but are demonstrating a different approach to the one described before (where the additional "End" variable was used). Here, we can avoid this additional variable simply by `ASSIGN`ing `ProdMolFrac` itself and then `RESET`ting at the end of the simulation. The disadvantage with this approach is that you cannot plot the mean of the distribution over time; it only contains the correct value at the end of the simulation, when `ProdMolFrac` gets assigned the correct values. In this example, we were not concerned with plotting the mean, *etc.*, over time and so this approach is an appropriate alternative.

The final extract of the gPROMS project, the `PROCESS` entity, is shown below.

```
# PROCESS Stochastic
```

```
UNIT
  R101 AS Stochastic_LiquidPhaseCSTR

SET
  WITHIN R101 DO
    NoScenarios       := 1000                              ;
    Upper_PMF         := 0.95                              ;
    Lower_PMF         := 0.92                              ;
    NoInt_PMF         := 20                                ;
    Dist_PMF          := [ BFDM, 1, NoInt_PMF ]            ;

    NoComp            := 4                                 ;
    NoReac            := 2                                 ;

    Nu                := [ -1,  0,
                           -1,  0,
                            1, -1,
                            0,  1  ]                       ;

    Order              := [ 1, 0,
                            1, 0,
                            0, 1,
                            0, 0  ]                        ;

    R                  := 8.31441                          ; # kJ/kmol/K

    ArrhConstant       := [ 8E-3, 1E-2 ]                   ; # m3/kmol s
    ActivationEnergy   := [ 8000, 6000 ]                   ; # kJ/kmol

    Density            := [ 17.48, 17.15, 10.24, 55.56 ]   ; # kmol/m3
  END

ASSIGN
  WITHIN R101 DO
    PMFstddev          := 0                    ;
    FOR i := 1 TO NoScenarios DO
      ProdMolFrac(i)   := 0                    ;
      WITHIN Scenarios(i) DO
        Fin            := 0                    ;
        Fout           := 0                    ;
        Xin            := [ 0.5, 0.5, 0, 0 ]   ;
        T              := NORMAL(353, 2)       ;
      END
    END
  END

INITIAL
  WITHIN R101 DO
    FOR i := 1 TO NoScenarios DO
      WITHIN Scenarios(i) DO
```

```
        X(2)            = X(1)   ;
        X(3)            = 0      ;
        X(4)            = 0      ;
        TotalVolume     = 10     ;
      END
    END
  END

SCHEDULE
  SEQUENCE
    CONTINUE FOR 3600
    RESET
      FOR i := 1 TO R101.NoScenarios DO
        R101.ProdMolFrac(i) := OLD(R101.Scenarios(i).X(4)) ;
      END
    END
    RESET
      R101.PMFstddev := SQRT(OLD(R101.PMFvariance)) ;
    END
    CONTINUE FOR .01
  END
```

Below are some comments on the `PROCESS`.

`SET` This section illustrates a couple of useful features in gPROMS. The first is that some of the `PARAMETER`s are having thier default values overridden. The second is that all of the `PARAMETER`s in the lower-level model (`LiquidPhaseCSTR`) are being propagated.

`ASSIGN` In this section we assign the dummy values to `ProdMolFrac` and `PMFstddev`. Also, some of the degrees of freedom of the `LiquidPhaseCSTR` model are set, *e.g.* the inlet and outlet flowrates, which are set to zero. Finally, the temperature for each scenario is set a random value from the normal distribution, $N(353, 2)$.

`INITIAL` A typical set of initial conditions are used here.

`SCHEDULE` This sections illustrates the `RESET`ting of the `VARIABLE`s `ProdMolFrac` and `PMFstddev`. Note that because `PMFstddev` depends on `ProdMolFrac`, the latter *must* be `RESET` before the former in a separate `RESET` task. If they are `RESET` in the same task, then `PMFstddev` will be `RESET` based on the values in `ProdMolFrac` from *before* the `RESET` task.

Finally, on some systems gRMS may not be able to plot the pdf `VARIABLE`s correctly (sometimes the value after the `RESET` is ignored by gRMS). A simple solution is to include a short `CONTINUE` at the end of the `SCHEDULE`. This is not an issue with the Excel output channel, although sending the values of a large number of `VARIABLE`s to Excel takes a considerable length of time. It is therefore recommended that you `MONITOR` only the `VARIABLE`s that are necessary.

The results of the stochastic simulation are shown in figures 9.2 to 9.4. Figure 9.2 shows the values assigned to the temperature for each scenario. Figure 9.3 shows the resulting distribution of product mole fractions. Finally, figure 9.4 shows the value of the standard deviation, also illustrating that in this model its value is only correct at the end of the simulation.

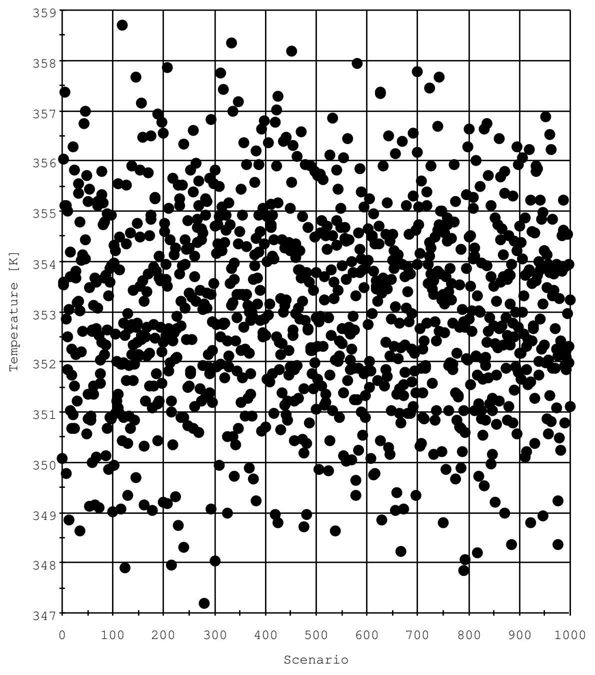


Figure 9.2: Values assigned to the temperature for each scnario.

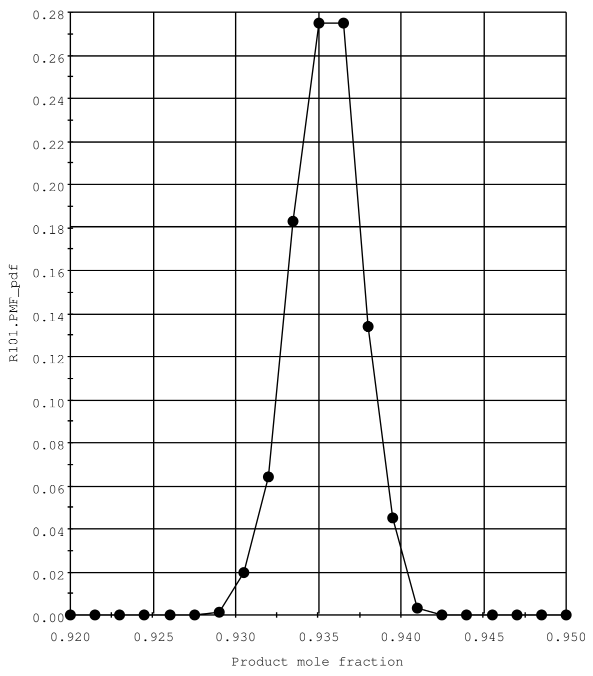


Figure 9.3: Probability density function for the product mole fraction (`X(4)`).

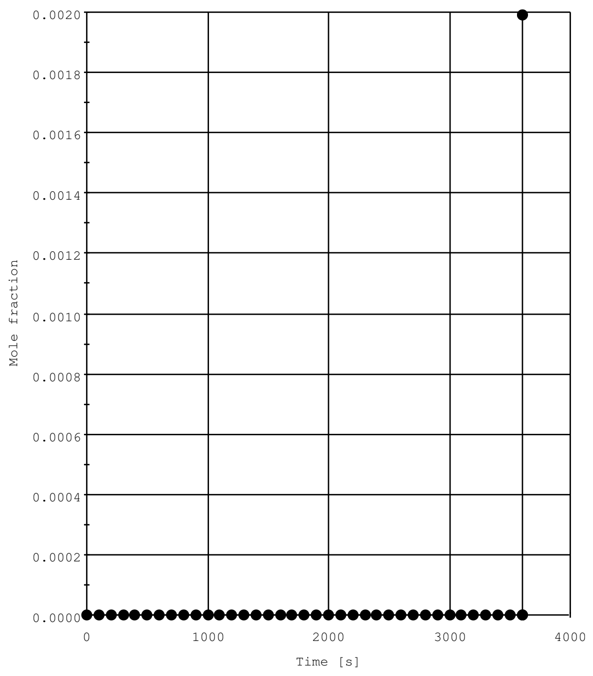


Figure 9.4: Standard deviation of the product mole fraction (`X(4)`).

Chapter 10

Controlling the Execution of Model-based Activities

Contents

As we have seen in section 2.2.5, the `PROCESS` entity is used to describe a simulation activity that is to be carried out by gPROMS using instances of one or more `MODEL` entities. Moreover, as described in chapters 2 and 3 of the gPROMS Advanced User Guide, `PROCESS` entities are central to other types of model-based activities supported by gPROMS, such as optimisation and parameter estimation.

The execution of model-based activities involves the solution of different types of mathematical problems. Typically, these are complex problems due to both their size and their nonlinearity. gPROMS provides a number of state-of-the-art mathematical solvers that employ a combination of symbolic, structural and numerical manipulations for the solution of these problems.

This chapter describes some important features of the `PROCESS` entity that are related with the solution of the underlying mathematical problems and the handling of the results produced by it:

- section 10.1 describes how you can provide initial guesses for the variables that occur in your model;
- section 10.2 describes how you can:
  - choose appropriate solvers for different kinds of problems,
  - specify the destination of any results that the solution may produce;
- the remainder of this chapter provides a detailed description of the mathematical solvers provided as standard within gPROMS; these fall in several categories:
  - solvers for sets of linear algebraic equations (section 10.3);
  - solvers for sets of nonlinear algebraic equations (section 10.4);
  - solvers for mixed sets of nonlinear algebraic and differential equations (section 10.5);
  - solvers for optimisation problems (section 10.6);
  - solvers for parameter estimation problems (section 10.7).

  The description of each solver includes a list of all the parameters that you can use to configure its precise behaviour when applying it to a particular problem.

10.1 The PRESET section

At the start of each simulation, gPROMS has to solve a problem known as *initialisation*. For both steady-state and dynamic simulations, gPROMS must first solve a system of algebraic equations (usually nonlinear). This naturally requires initial guesses for all of the variables in order to provide the solution algorithm with a starting point. These initial guesses (and appropriate bounds on the variables) are specified in the `VARIABLE TYPE` entities (see section 2.2.4). Usually, specifying the initial guesses in this manner *i.e.* the same initial guess and bounds for variables of the same type) is sufficient for gPROMS to solve the initialisation problem. Larger, more complex problems, however, may not be suited to this approach and therefore a more flexible method is needed specifying the initial guesses. This is catered for through the `PRESET` section, which allows the default initial guess and bounds of a variable to be overridden.

The syntax for the `PRESET` section is:

```
PRESET     VariablePath  := InitialValue ;
```

or

```
PRESET     VariablePath  := InitialValue : LowerBound : UpperBound ;
```

`WITHIN` and `FOR` statements may also be used in the `PRESET` section.

Even once a set of suitable initial guesses are found, some problems may take a considerable length of time to solve. This can often be greatly reduced if the solution of the initialisation problem is used to provide the initial guesses. This can be done in gPROMS using *Saved Variable Sets* by `SAVE`-ing the values of all variables after the initialisation and restoring them in the `PRESET` section as shown in figure 10.1. In the first `PROCESS`, a set of initial guesses is used that is sufficient for the initialisation to be solved. The second `PROCESS` then uses the data in the save file to solve the initialisation more quickly. Note that the second `PROCESS` may also save the result of the initialisation problem, so that changes can be made to the problem without having to run the first `PROCESS` again.

Multiple *Saved Variable Sets* can be `RESTORE`d in the `PRESET` section, along with manually specified initial guesses, as shown in the example below. In all cases, any initial guess provided for a particular variable, either via an explicit specification or via a `RESTORE`, will override all earlier initial guesses for the same variable.

```
PRESET
  RESTORE "v_set1", "v_set2" ;"
  RESTORE "v_set3" ;"
  VariablePath  := InitialValue ;
  VariablePath  := InitialValue : LowerBound  : UpperBound;
  RESTORE "v_set4", "v_set5" ;"
  RESTORE "v_set6" ;"
```

```
# PROCESS InitSim1

    ...

    PRESET
      WITHIN aaa DO
        x(1)    := 1            ;
        x(2)    := 1            ;
        x(3:10) := 0            ;
        y()     := 5            ;
        z       := 10 : 5 : 100 ;
        ...
      END

    ...

    SCHEDULE
      SAVE "InitialisationData"
```

(a) PROCESS used to solve the initialisation problem only

```
# PROCESS FullSim

    ...

    PRESET
      RESTORE "InitialisationData"
    ...

    SCHEDULE
      SEQUENCE
        SAVE "InitialisationData"
        ...
      END # sequence
```

(b) Full PROCESS restoring data from the successful initialisation

Figure 10.1: The use of RESTORE in the PRESET section.

10.2 The SOLUTIONPARAMETERS section

The `SOLUTIONPARAMETERS` section allows the specification of parameters that affect:

- the results generated by the execution of a model-based activity;
- the mathematical solvers to be used for the execution of a model-based activity.

The basic syntax for the `SOLUTIONPARAMETERS` section, along with the default values of the parameters, is shown below:

```
SOLUTIONPARAMETERS
  # parameters concerned with output generation
  gExcelOutput        := OFF      ;
  gPLOT               := OFF      ;
  gRMS                := ON       ;
  gUserOutput         := OFF      ;
  Monitor             := ON       ;
  ReportingInterval   := 0.0      ;

  # parameters concerned with mathematical solvers
  DASolver            := "DASOLV" ;
  DOSolver            := "CVP_SS" ;
  LASolver            := "MA48"   ;
  NLSolver            := "NLSOL"  ;
  PESolver            := "MXLKHD" ;
```

Normally, the above default values are sufficient to solve most problems. However, they may be overridden in the `SOLUTIONPARAMETERS` section if and when necessary.

10.2.1 Controlling result generation and destination

The first group of the above parameters allow the user to control the generation of results by the execution of a model-based activity, as well as the destination of these results.

gExcelOutput Enables or disables the Microsoft Excel output channel (see Appendix C).

By default, this parameter is switched `OFF`. When set to `ON`, output is sent to a file whose stem is the `PROCESS` entity name plus an index in square brackets to represent the number of times the process has been executed. For example, if the process name was *MyProcess* the first output file generated would be called `MYPROCESS.xls`; the second would be `MYPROCESS[2].xls`; and so on.

A different file name can be specified directly in the `SOLUTIONPARAMETERS` section using the syntax:

`gExcelOutput := "`*FileName*`" ;`

Note that this automatically implies that the `gExcelOutput` parameter is switched `ON`.

gPLOT Enables or disables the generation of text results files (see Appendix D).

By default, this parameter is switched `OFF`. When set to `ON`, output is sent to a file whose name is the `PROCESS` entity name followed by `.gPLOT`. A different file name can be specified directly in the `SOLUTIONPARAMETERS` section using the syntax:

`gPLOT := "`*FileName*`" ;`

Note that this automatically implies that the `gPLOT` parameter is switched `ON`.

gRMS Enables or disables communication between gPROMS and the gPROMS Results Management System (gRMS, see Appendix B).

By default, this parameter is switched `ON`. When set to `ON`, output is archived under a gRMS process (*cf.* section B.1) with the same name as that of the `PROCESS` entity. A different name can be specified directly in the `SOLUTIONPARAMETERS` section using the syntax:

`gRMS := "`*FileName*`" ;`

Note that this automatically implies that the `gRMS` parameter is switched `ON`.

gUserOutput Enables or disables a user-defined output channel.

The construction of such output channels is described in detail in the gPROMS System Programmer Guide.

By default, this parameter is switched `OFF`.

Monitor Sets the initial state for monitoring of variables.

By default, this parameter is switched `ON`. If set to `OFF`, no results will be collected during the execution of the model-based activity. However, for dynamic simulation activities, monitoring can be enabled at a later stage by inserting the `MONITOR` elementary task in the simulation `SCHEDULE` (*cf.* section 7.3.2).

ReportingInterval Specifies the reporting interval for results.

This is the frequency at which variable values are transmitted to the output channel(s) during a dynamic simulation activity.

This parameter does not have a default value. If it is omitted, the user will be asked to enter a value interactively before the simulation starts (*cf.* section 2.3).

10.2.2 Choosing mathematical solvers for model-based activities

gPROMS supports three main types of model-based activity, namely:

- simulation
- optimisation
- parameter estimation

Each one of these activities can be based on either steady-state or dynamic models.

gPROMS provides a range of state-of-the-art proprietary solvers for the execution of different types of activity. Albeit sufficiently general to handle the dynamic case, these solvers are designed to automatically detect whether a particular problem is, in fact, a steady-state one and to take this into account in its solution.

gPROMS also supports an open software architecture regarding mathematical solvers. This basically means that third-party solvers can be used within gPROMS without any modifications either to the gPROMS software or to the models written in it. Detailed information on this topic can be found in the gPROMS System Programmer Guide.

The `SOLUTIONPARAMETERS` section provides three parameters that can be used to specify which solver (either standard gPROMS or third-party) should be used for each type of activity:

- `DASolver` specifies the solver to be used for simulation activities[1];
- `DOSolver` specifies the solver to be used for optimisation activities[2];
- `PESolver` specifies the solver to be used for parameter estimation activities.

Note that a `PROCESS` entity may contain specifications for all three types of solver irrespective of the kind of activity for which it is actually used.

The value of each of the above three parameters is actually a string identifying the solver to be used, enclosed in double quotes. For example, the syntax:

```
DASolver := "SRADAU";
DOSolver := "DYNOPT";
```

would be used to indicate that:

- dynamic simulation is to be performed with the `SRADAU` solver, one of the standard gPROMS dynamic simulation solvers(*cf.* section 10.5.2);
- dynamic optimisation should use a (hypothetical) third-party dynamic optimisation solver called `DYNOPT`.

Note that the name of the solver is *always* enclosed in double quotes.

10.2.3 Configuring the mathematical solvers

A mathematical solver for a model-based activity, such as dynamic simulation or optimisation, is usually a complex piece of software. Its precise behaviour and performance in solving any particular problem is controlled by a number of *algorithmic parameters*. For example, the quality of the results produced by a dynamic simulation solver (and also the computational effort required) can be controlled by adjusting one or more error tolerances. Each algorithmic parameter will normally have a default value which is chosen to lead to good (if not optimal) performance for a wide range of problems; this default will be used unless the user specifies a different value.

The set of algorithmic parameters recognised by two different solvers – even of the same type – will generally be different. gPROMS provides a general mechanism for specifying algorithmic parameter values of five distinct types:

[1]The “DA” in `DASolver` stands for “differential-algebraic”; this reflects the fact that the main mathematical operation involved in performing dynamic simulation activities is the solution of mixed sets of differential and algebraic equations.

[2]The “DO” in `DOSolver` stands for “dynamic optimisation”; this reflects the fact that all standard optimisation solvers in gPROMS are designed for the general case of optimisation of systems under transient conditions.

- integer algorithmic parameters (*e.g.* the maximum permitted number of iterations);
- real algorithmic parameters (*e.g.* the error tolerances);
- logical algorithmic parameters (*e.g.* whether a certain feature of the solver is to be used or not);
- string algorithmic parameters (*e.g.* the name of a file to receive special output generated by the solver);
- enumerated algorithmic parameters; these are strings (enclosed in double quotes) that can take only certain values (*e.g.* `"OFF"`, `"MEDIUM"`, `"HIGH"`) which are recognised by the solver;
- solver algorithmic parameters; these are strings (enclosed in double quotes) that specify sub-solvers to be used by the solver, as explained in detail in section 10.2.4 below.

For example,the following syntax would be used to specify that a dynamic simulation should be performed using the `SRADAU` solver with an output level of 2, an absolute error tolerance of 10^{-8}, and with the generation of a special diagnostics output file switched on:

```
DASolver := "SRADAU" ["OutputLevel"       := 2;
                      "AbsoluteTolerance" := 1E-8;
                      "Diag"              := TRUE] ;
```

A complete list of all the parameters associated with the `SRADAU` solver is given in section 10.5.2. The important things to note here are:

- the name of the algorithmic parameter is *always* enclosed in double quotes, as is the name of the solver itself;
- the values of algorithmic parameters of type string, enumerated and solver (not shown in the above example) must be enclosed in double quotes;
- any algorithmic parameters not specified here will retain their default values.

10.2.4 Specifying solver-type algorithmic parameters

As mentioned above, some of the algorithmic parameters used to configure solvers may be solvers themselves. For example, solving a set of differential and algebraic equations typically requires the solution of a number of mathematical sub-problems involving sets of either nonlinear or linear algebraic equations. Thus, a differential-algebraic equation solver will normally need to make use of both a nonlinear equation solver and a linear equation solver. We will refer to these as the "sub-solvers" associated with this solver.

Some mathematical solvers have built in sub-solvers that they always use for their operation. On the other hand, more advanced solvers may allow their users to specify the sub-solver to be used. This can be done via an algorithmic parameter. For instance, consider the following extended form of the example specification of the dynamic simulation solver presented in section 10.2.3:

```
   DASolver := "SRADAU" ["OutputLevel"              := 2;
                         "AbsoluteTolerance"        := 1E-8;
                         "Diag"                     := TRUE;
                         "LASolver"                 := "MA28";
                         "InitialisationNLSolver"   := "SPARSE";
                         "ReinitialisationNLSolver" := "SPARSE"] ;
```

This specifies that the `SRADAU` solver should use the `MA28` solver for the solution of any sets of linear algebraic equations that it needs to perform[3]. In addition to a sub-solver for linear equations, the `SRADAU` solver also needs two sub-solvers for nonlinear algebraic equations. One of these is used for the initialisation of the dynamic simulation and the other one for re-initialisation following discontinuities. In the above example, we are specifying that the `SPARSE` solver should be used for both of these tasks[4]. In all cases, note that the value of a solver-type algorithmic parameter (*i.e.* the name of the sub-solver to be used) needs to be enclosed in double quotes.

Of course, a sub-solver is itself a solver and may have its own algorithmic parameters that the user may specify. In the above example, we may wish to specify a tight convergence tolerance for the initialisation solver and a slightly less tight one for re-initialisation. This can be done using the syntax:

```
DASolver := "SRADAU" ["OutputLevel"              := 2;
                      "AbsoluteTolerance"        := 1E-8;
                      "Diag"                     := TRUE;
                      "LASolver"                 := "MA28";
                      "InitialisationNLSolver"   := "SPARSE"
                                                      ["ConvergenceTolerance" := 1E-8];
                      "ReinitialisationNLSolver" := "SPARSE"
                                                      ["ConvergenceTolerance" := 1E-7]] ;
```

In fact, some of the sub-solvers may themselves have solver-type parameters. For example, nonlinear equation solvers, such as `SPARSE`, often need to solve sub-problems that involve sets of linear algebraic equations. Again, this can be accommodated within the general syntax presented above. For example:

```
DASolver := "SRADAU" ["OutputLevel"              := 2;
                      "AbsoluteTolerance"        := 1E-8;
                      "Diag"                     := TRUE;
                      "LASolver"                 := "MA28";
                      "InitialisationNLSolver"   := "SPARSE"
                                                      ["ConvergenceTolerance" := 1E-8;
                                                       "LASolver"             := "MA48"];
                      "ReinitialisationNLSolver" := "SPARSE"
                                                      ["ConvergenceTolerance" := 1E-7;
                                                       "LASolver"             := "MA28"]] ;
```

[3]`MA28` is one of the linear algebraic equation solvers provided as standard within gPROMS (see section 10.3.1).

[4]`SPARSE` is one of the nonlinear algebraic equation solvers provided as standard within gPROMS (see section 10.4.3).

specifies that the SPARSE solver used for initialisation should make use of the MA48 linear algebra solver, while that used for re-initialisation should employ the MA28 solver. Moreover, MA28 will be used by SRADAU to solve any linear equations systems arising outside the initialisation and re-initialisation stages of its operation.

The above syntax for specifying and configuring sub-solvers within solvers is recursive and can be used to define solver hierarchies with any number of levels. For example, a dynamic optimisation solver can use a differential-algebraic equation solver, which in turn can make use of a nonlinear equation solver, which can employ a linear equation solver.

10.2.5 Specifying default linear and nonlinear equation solvers

Most mathematical solvers for simulation, optimisation and parameter estimation need to make use of sub-solvers for the solution of sets of linear and nonlinear algebraic equations. In order to avoid having to specify and configure these low level solvers repeatedly within the same SOLUTIONPARAMETERS section, gPROMS provides two solution parameters that can be used to specify and configure *default* linear and nonlinear algebraic equation solvers. Thus, in addition to the three main solver parameters DASolver, DOSolver and PESolver described in section 10.2.2, gPROMS recognises the following two parameters:

- LASolver specifies the default sub-solver for sets of linear algebraic equations;
- NLSolver specifies the default sub-solver for sets of nonlinear algebraic equations.

Consider, for example, the specification:

```
# default linear algebraic equation solver configuration
LASolver := "MA28" ["PivotStabilityFactor"    := 0.2;
                    "ExpansionFactor"         := 3;
                    "MaxStructures"           := 4] ;

# default nonlinear algebraic equation solver configuration
NLSolver := "SPARSE" [ "OutputLevel"          := 3;
                       "MaxFuncs"             := 1000;
                       "MaxIterNoImprove"     := 5;
                       "NStepReductions"      := 10;
                       "MaxIterations"        := 1000;
                       "ConvergenceTolerance" := 1E-8] ;

DASolver := "DASOLV" ["OutputLevel"           := 1;
                      "AbsoluteTolerance"     := 1E-8] ;

DOSolver := "CVP_MS";
```

This specifies that, whenever the DASOLV solver (*cf.* section 10.5.1) needs to solve sets of linear or nonlinear algebraic equations, it should use, respectively, the MA28 and SPARSE solvers configured as shown above. Also, whenever SPARSE itself requires the solution of a set of linear equations, it should also use MA28 in the same configuration.

The above also specifies that the CVP_MS solver should be used for the execution of dynamic optimisation activities (*cf.* section 10.6.4). This solver will also make use of the specified sub-solver choices and configurations for linear and nonlinear algebraic equations.

Interestingly, `CVP_MS` also requires a differential-algebraic equation solver for its operation. This could be achieved by specifying the value of a solver-type algorithmic parameter called `DASolver`, *e.g.* `DOSolver := "CVP_MS" ["DASolver" := "SuperDAE"];` where `SuperDAE` is a (hypothetical) third-party solver for differential-algebraic equations. However, since no such explicit specification is made above, `CVP_MS` will actually use the `DASolver` choice and configuration shown above for this purpose.

In conclusion, specifying the `DASolver` parameter in `SOLUTIONPARAMETERS` fulfils a dual function as it defines:

- the mathematical solver to be used for simulation activities;
- the default sub-solver to be used by the optimisation and parameter estimation activity solvers whenever they need to solve sets of differential and algebraic equations.

10.3 Standard solvers for linear algebraic equations

There are two standard mathematical solvers for the solution of sets of linear algebraic equations in gPROMS, namely `MA28` and `MA48`. Both of these employ direct LU-factorisation algorithms, designed for large, sparse, asymmetric systems of linear equations. `MA48` is the newer of the two codes.

The `LASolver` solution parameter (*cf.* section 10.2.5) may be used to change and/or configure the default linear algebra sub-solver used by all higher-level solvers. If this parameter is not specified, then the `MA48` solver is used, with the default configuration shown at the start of section 10.3.2 below.

10.3.1 The `MA28` solver

The algorithmic parameters used by `MA28` along with their default values are shown below. This is followed by a detailed description of each parameter.

```
"MA28" [ "OutputLevel"          := 0;
         "PivotStabilityFactor" := 0.1;
         "ExpansionFactor"      := 4;
         "MaxStructures"        := 6;
         "MaxStructuresMemory"  := 100000 ] ;
```

`OutputLevel` An integer in the range [-1, 1].

The amount of information generated by the solver. The following table indicates the lowest level at which different types of information are produced:

-1	(None)
0	Creation and deletion of systems usage statistics on deletion workspace increases
1	Structure analysis messages

`PivotStabilityFactor` A real number in the range [0.0, 1.0].

Controls the balance between minimising the creation of new non-zero elements during the matrix factorisation[5] (`PivotStabilityFactor` = 0) and numerical stability (`PivotStabilityFactor` = 1).

`ExpansionFactor` An integer of value 1 or higher.

The amount of space that gPROMS allocates for the matrix factorisation at the start of a computation is given by:

$$\texttt{ExpansionFactor} \times \text{(Number of Nonzero Elements in Matrix)}$$

gPROMS will automatically expand this storage at a later stage during the computation if the original allocation is found to be insufficient. However, if the amount of storage needed by a particular computation is known *a priori*[6], it will usually be more efficient to allocate it from the start by specifying an appropriate value for `ExpansionFactor`.

[5]And consequently, the amount of storage required by the factorisation.

[6]For example, from experience from earlier similar computations.

`MaxStructures` An integer of value 0 or higher.

The execution of a model-based activity in gPROMS typically involves the factorisation of a number of matrices of several different structures. The gPROMS implementation of `MA28` allows the option of storing information on one or more structures encountered for possible re-use at a later stage of the execution if it is required again to factorise a matrix with one of those structures. This may significantly improve the efficiency of handling discontinuities at the expense of higher memory requirements. The parameter `MaxStructures` is an upper limit on the number of distinct structures that may be stored during any one simulation.

`MaxStructureMemory` An integer of value 0 or higher.

This is an upper bound on the number of integer variable locations that may be used as part of the structure storage scheme described above.

10.3.2 The `MA48` solver

The algorithmic parameters used by `MA48` along with their default values are shown below. This is followed by a detailed description of each parameter.

```
"MA48" [ "OutputLevel"          := 0;
         "PivotStabilityFactor" := 0.1;
         "ExpansionFactor"      := 5;
         "FullSwitchFactor"     := 0.5;
         "PivotSearchDepth"     := 3;
         "BLASLevel"            := 32;
         "MinBlock"             := 1 ] ;
```

`OutputLevel` An integer in the range [-1, 4].

The amount of information generated by the solver. The following table indicates the lowest level at which different types of information are produced:

0	(None)
1	Creation and deletion of systems, usage statistics including CPU, workspace increases, numerical singularity
2	Warning messages, *e.g.* for duplicate entries, which can be ignored
3	Information from the internal Fortran calls: a few entries of the matrix to be factorised and the result
4	More information, including all entries in the factorised matrices, and the right-hand-side and solutions vectors.

`PivotStabilityFactor` As for `MA28`.

`ExpansionFactor` As for `MA28`.

`FullSwitchFactor` A real number in the range [0.0, 1.0].

The `MA48` linear solver has an option of switching to full-matrix linear algebra computations at any stage during the matrix factorisation process if the proportion of non-zero elements in the matrix remaining to be factorised exceeds a specified threshold. The latter can be adjusted by the parameter `FullSwitchFactor`.

`PivotSearchDepth` An integer of value 0 or higher.

The number of columns within which the search for an appropriate pivot element during a factorisation is limited. Generally, a higher number will result in a more numerically stable pivot selection, at the expense of higher computation time. If `PivotSearchDepth` is set to zero, `MA48` will use a special technique for finding the best pivot. Although this may result in reduced fill-in, pivot search in this case is usually slower and occasionally very slow.

`BLASLevel` An integer of value 0 or more.

`MA48` makes use of the Basic Linear Algebra System (BLAS) for vector and matrix operations. BLAS is organised in three different levels, in ascending order of sophistication of the services offered. The `BLASLevel` parameter specifies that BLAS level `BLASlevel`+1 should be used by `MA48`. Additionally, if the value is 2 or more, it is used to set the block column size, a parameter only applicable to level 3. For this reason, the default is 32.

`MinBlock` An integer of value 1 or higher.

`MA48` makes use of block triangularisation as a means of accelerating the factorisation and solution of linear systems. This parameter specifies the minimum block size to be considered in this context.

10.4 Standard solvers for nonlinear algebraic equations

There are three standard mathematical solvers for the solution of sets of nonlinear algebraic equations in gPROMS, namely `BDNLSOL`, `NLSOL` and `SPARSE`:

- `BDNLSOL` stands for "Block Decomposition NonLinear SOLver". It is a new implementation of a general solver for solving sets of nonlinear equations rearranged to block triangular form, and employs a novel algorithm for the handling of equations with reversible symmetric discontinuities (`IF` equations). It also optionally supports "preset propagation", a means of making better use of information provided in the PRESET section. As a modular solver component, `BDNLSOL` can in principle make use of any other nonlinear solver component to solve its individual blocks.
- `NLSOL` is a general purpose nonlinear solver, with and without block decomposition. In reality, it is implemented by providing access to the nonlinear solvers existing within gPROMS prior to version 2.1 as a means of providing continuity and complete backward compatibility with earlier versions.

  Due to its implementation, `NLSOL` is not a true software component, and this implies certain restrictions in its use. In particular, it can be used only as the nonlinear solver for simulation activities.
- `SPARSE` is a true solver component for solution of nonlinear algebraic systems without block decomposition. It provides a sophisticated implementation of a Newton-type method.

All of the above solvers are designed to deal with large, sparse systems of equations in which the variable values are restricted to lie within specified lower and upper bounds. Moreover, they can handle situations in which some of the partial derivatives of the equations with respect to the variables are available analytically while the rest have to approximated[7]. An efficient combination of finite difference approximations and least-change secant updates is used for the latter purpose.

The `NLSolver` solution parameter (*cf.* section 10.2.5) may be used to change and/or configure the default linear algebra sub-solver used by all higher-level solvers. If this parameter is not specified, then:

- simulation activities make use of the `NLSOL` solver with the default configuration shown at the start of section 10.4.2;
- optimisation and parameter estimation activities make use of the `BDNLSOL` solver with the default configuration shown at the start of section 10.4.1.

10.4.1 The `BDNLSOL` solver

The algorithmic parameters used by `BDNLSOL` along with their default values are shown below. This is followed by a detailed description of each parameter.

[7]In gPROMS models, almost all partial derivatives are computed analytically from expressions derived using symbolic manipulations. The main exception is partial derivatives of equations involving any Foreign Object methods that are not capable of returning partial derivatives (see chapter 4 of the gPROMS Advanced User Guide).

```
"BDNLSOL" [OutputLevel                    := 0;
           "LASolver"                     := "MA48";
           "BlockSolver"                  := "SPARSE";
           "UsePresetPropagation"         := TRUE;
           "PresetPropagationOutputLevel" :=  0;
           "IdentityElimination"          := FALSE ];
```

`OutputLevel` An integer in the range [-1, 5].

The amount of information generated by the solver. The following table indicates the lowest level at which different types of information are produced:

0	(None)
1	Numbers of equations in each block in the main block decomposition Failures to solve linear blocks
2	Result of the block decomposition: equation and variable numbers in each block Progress of preset propagation “Solving block *n*” message
3	Changes in variable values due to preset propagation Changed/unchanged variables due to solving single linear equations Final variable values after solving nontrivial blocks
4	Details of block decomposition performed for preset propagation (variable and equation numbers of assignments and blocks) Equation residuals after changing variable values for preset propagation
5	Table of equation names necessary to interpret information from main block decomposition step Tables of variable and equation names necessary to interpret information from block decomposition during preset propagation

`BlockSolver` A quoted string specifying a nonlinear algebraic equation solver.

The solver to be used for the solution of the nonlinear systems representing each block. This can be either `SPARSE` (*cf.* section 10.4.3) or a third-party nonlinear algebraic equation solver (see the gPROMS System Programmer Guide).

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

`LASolver` A quoted string specifying a linear algebraic equation solver.

The solver to be used for the solution of linear algebraic equations arising in the course of the preset propagation algorithm. This can be either one of the standard gPROMS linear algebraic equation solvers (*cf.* section 10.3) or a third-party linear algebraic equation solver (see the gPROMS System Programmer Guide). The default is `MA48`.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

`UsePresetPropagation` A boolean value.

Specifies whether or not the preset propagation algorithm is to be used. Use of this algorithm may assist with difficult initialisations. However there is cost in terms of execution time.

`PresetPropagationOutputLevel` An integer in the range [-1, 5].

Controls the output from preset propagation independently from the output for the solver.

0	(None)
2	Progress of preset propagation
3	Changes in variable values due to preset propagation
4	Details of block decomposition performed for preset propagation (variable and equation numbers of assignments and blocks) Equation residuals after changing variable values for preset propagation
5	Tables of variable and equation names necessary to interpret information from block decomposition during preset propagation

`IdentityElimination` A boolean value.

If set to TRUE, the solver will attempt to internally reduce the size of the problem by removing equations of the form $x = y$, and substituting all occurences of one of these variables with the other. Such equations are often introduced in gPROMS models by stream connectivity equations. This may result in faster solution time although at the current stage of development the costs of creating and using the reduced system sometimes outweigh the benefits, particularly when many IF and CASE conditions are present.

10.4.2 The `NLSOL` solver

The algorithmic parameters used by `NLSOL` along with their default values are shown below. This is followed by a detailed description of each parameter.

```
"NLSOL" ["OutputLevel"            := 0;
         "ConvergenceTolerance"  := 1E-5;
         "MaxFuncs"              := 1000000;
         "MaxIterNoImprove"      := 10;
         "NStepReductions"       := 10;
         "MaxIterations"         := 1000;
         "EffectiveZero"         := 1E-5;
         "FDPerturbation"        := 1E-5;
         "SingPertFactor"        := 1E-2;
         "SLRFactor"             := 50;
         "LASolver"              := "MA48";
         "UseBlockDecomposition" := TRUE ] ;
```

`OutputLevel` An integer in the range [-1, 10].

The amount of information generated by the solver. The following table indicates the lowest level at which different types of information are produced:

-1	(None)
0	Halving of step due to unsatisfactory progress, initial point out of bounds
1	Solution parameters on first use, variables hitting bounds
2	Method and scaling information, number of equation evaluations on convergence, failure to improve in `MaxIterNoImprove` iterations, equation residuals norm, equation with highest residual, variables stuck on bounds, number of variables reset to bounds
3	Variable and equation names of each nonlinear system, call number and condition, step reduction factors
4	Residuals at every evaluation, variables at each iteration, lists of variables being perturbed
5	Variable values before solution, workspace information, steps taken at each iteration
6	Complete Jacobian at each factorisation
10	Solution parameters on every use

`ConvergenceTolerance` A real number in the range $[10^{-20}, 10^{10}]$.

The tolerance used in testing for convergence of the nonlinear system $f(x) = 0$ being solved. A system of n equations $f(x)$ in n unknowns x is assumed to have converged when the norm of the equations:

$$||f(x)|| \equiv \max_{i \in [1,n]} |f_i(x)|$$

falls below the `ConvergenceTolerance`. This is equivalent to the absolute value of the difference between the left and right hand sides of each and every equation in the system being below this tolerance. Note that no automatic scaling is applied by the solver.

`MaxFuncs` An integer in the range [0, 1000000].

The maximum number of evaluations of the vector of equations $f(x)$ that is permitted during solution. This includes the equation evaluations required for approximating any elements of the Jacobian matrix $\partial f/\partial x$ that are not available analytically, using finite difference perturbations.

`MaxIterNoImprove` An integer in the range [0, 1000000].

The maximum number of iterations without a reduction in the norm of the equation vector (see above) before the solver takes corrective action. For convergence to be achieved, this norm must eventually decrease to below the `ConvergenceTolerance`. However, it may actually increase between two consecutive iterations.

The solver monitors the norm at each iteration. It also keeps a record of the best (*i.e.* lowest) norm obtained so far in the solution, the values of the unknowns x^{best} at this point, and the value of the step length restriction factor β^{best} that was applied at the corresponding iteration (see parameter `SLRFactor` below). If no improvement over this best norm is observed within `MaxIterNoImprove` consecutive iterations, then the solver attempts to take corrective action, as follows:

- the unknowns are reset to x^{best};
- the Jacobian matrix is recomputed, using finite differences for any elements not available analytically;
- the step length restriction factor β (see parameter `SLRFactor` below) is halved.

`NStepReductions` An integer in the range [0, 1000000].

The maximum number of consecutive corrective actions that the solver is allowed to attempt. As explained in the context of parameter `MaxIterNoImprove` above, the solver attempts to take certain corrective actions if no improvement in the equation norm is achieved within a certain number of consecutive iterations. If such corrective action is attempted more than `NStepReductions` times in a row (*i.e.* having to return to the same x^{best} in all cases), then the solver terminates its operation unsuccessfully.

`MaxIterations` An integer in the range [0, 1000000].

The maximum number of iterations that the solver is allowed to take. Note that, unlike `MaxFuncs` (see above), this does not include any evaluations of the equations for the purpose of estimating elements of the Jacobian matrix using finite difference perturbations.

`EffectiveZero` A real number in the range $[10^{-20}, 10^{10}]$.

The magnitude of a variable below which absolute rather than relative perturbations are used – see parameters `FDPerturbation`, `SingPertFactor` and `SLRFactor` below.

`FDPerturbation` A real number in the range $[10^{-20}, 10^{10}]$.

Finite difference perturbation factor. If finite difference calculation of partial derivatives with respect to a variable x is required, x is perturbed by:

$$\texttt{FDPerturbation} \times |x|$$

unless $|X|$ is less than `EffectiveZero` (see above), in which case it is perturbed by `FDPerturbation`.

`SingPertFactor` A real number in the range $[10^{-20}, 10^{10}]$.

The perturbation factor used for escaping from local singularities. If, at a certain iteration, the Jacobian matrix is found to be singular (with a rank r that is less than the size of the system n), the solver attempts to escape from such a point by applying a perturbation to $n - r$ of the system variables. For a variable x, the size of this perturbation is:

$$\texttt{SingPertFactor} \times |x|$$

unless $|x|$ is less than `EffectiveZero` (see above), in which case it is perturbed by `SingPertFactor`.

`SLRFactor` A real number in the range $[10^{-20}, 10^{10}]$.

The step length restriction factor. In the interests of improving convergence from poor initial guesses, the solver automatically limits the step taken in any iteration by a fraction $\alpha \in (0, 1]$ so that the magnitude of the change in any variable x does not exceed:

- $\beta|x|$ if x is equal to, or exceeds the `EffectiveZero` (see above);
- β otherwise.

The factor β is set to `SLRFactor` at the start of the solution. Thereafter, it is adjusted automatically to reflect the difficulty of solving the system:

- β is reduced if corrective action needs to be taken (see parameter `MaxIterNoImprove` above);
- β is increased if fast reduction in the equation norm is observed from one iteration to the next.

`LASolver` A quoted string specifying a linear algebraic equation solver.

The solver to be used for the solution of linear algebraic equations at every iteration. This can be either one of the standard gPROMS linear algebraic equation solvers (*cf.* section 10.3) or a third-party linear algebraic equation solver (see the gPROMS System Programmer Guide). The default is `MA48`.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

`UseBlockDecomposition` A boolean value.

Specifies whether block decomposition should be applied to the nonlinear system. If it is, then all the parameters mentioned above are applied to the solution of each block individually.

10.4.3 The `SPARSE` solver

The algorithmic parameters used by `SPARSE` along with their default values are shown below. This is followed by a detailed description of each parameter.

```
"SPARSE" ["OutputLevel"          := 0;
          "ConvergenceTolerance" := 1E-5;
          "MaxFuncs"             := 1000000;
          "MaxIterNoImprove"     := 10;
          "NStepReductions"      := 10;
          "MaxIterations"        := 1000;
          "EffectiveZero"        := 1E-5;
          "FDPerturbation"       := 1E-5;
          "SingPertFactor"       := 1E-2;
          "SLRFactor"            := 50;
          "LASolver"             := "MA48";
          "IdentityElimination"  := FALSE;
          "IterationsWithoutNewJacobian" := 0 ] ;
```

`OutputLevel` An integer in the range [-1, 10].

The amount of information generated by the solver. The following table indicates the lowest level at which different types of information are produced:

-1	(None)
0	Halving of step due to unsatisfactory progress, initial point out of bounds
1	Solution parameters on first use, variables hitting bounds
2	Method and scaling information, residual and call number on convergence, failure to improve in MaxIterNoImprove iterations, residual and worst equation number at each call to driver, variables stuck on bounds, number of variables reset to bounds
3	Variable and equation names of each nonlinear system, call number and condition on each call to driver, step reduction factors
4	Residuals at every evaluation, variables at each iteration, function value on each call to driver, lists of variables being perturbed
5	Variable values before solution, workspace information, solutions of linear systems (*i.e.* steps)
6	Complete Jacobian at each factorisation
10	Solution parameters on every use

`ConvergenceTolerance` A real number in the range $[10^{-20}, 10^{10}]$.

The tolerance used in testing for convergence of the nonlinear system $f(x) = 0$ being solved. A system of n equations $f(x)$ in n unknowns x is assumed to have converged when the norm of the equations:

$$||f(x)|| \equiv \max_{i \in [1,n]} |f_i(x)|$$

falls below the `ConvergenceTolerance`. This is equivalent to the absolute value of the difference between the left and right hand sides of each and every equation in the system being below this tolerance. Note that no automatic scaling is applied by the solver.

`MaxFuncs` An integer in the range [0, 1000000].

The maximum number of evaluations of the vector of equations $f(x)$ that is permitted during solution. This includes the equation evaluations required for approximating any elements of the Jacobian matrix $\partial f/\partial x$ that are not available analytically, using finite difference perturbations.

`MaxIterNoImprove` An integer in the range [0, 1000000].

The maximum number of iterations without a reduction in the norm of the equation vector (see above) before the solver takes corrective action. For convergence to be achieved, this norm must eventually decrease to below the `ConvergenceTolerance`. However, it may actually increase between two consecutive iterations.

The solver monitors the norm at each iteration. It also keeps a record of the best (*i.e.* lowest) norm obtained so far in the solution, the values of the unknowns x^{best} at this point, and the value of the step length restriction factor β^{best} that was applied at the corresponding iteration (see parameter `SLRFactor` below). If no improvement over this best norm is observed within `MaxIterNoImprove` consecutive iterations, then the solver attempts to take corrective action, as follows:

- the unknowns are reset to x^{best};
- the Jacobian matrix is recomputed, using finite differences for any elements not available analytically;
- the step length restriction factor β (see parameter `SLRFactor` below) is halved.

`NStepReductions` An integer in the range [0, 1000000].

The maximum number of consecutive corrective actions that the solver is allowed to attempt. As explained in the context of parameter `MaxIterNoImprove` above, the solver attempts to take certain corrective actions if no improvement in the equation norm is achieved within a certain number of consecutive iterations. If such corrective action is attempted more than `NStepReductions` times in a row (*i.e.* having to return to the same x^{best} in all cases), then the solver terminates its operation unsuccessfully.

`MaxIterations` An integer in the range [0, 1000000].

The maximum number of iterations that the solver is allowed to take. Note that, unlike `MaxFuncs` (see above), this does not include any evaluations of the equations for the purpose of estimating elements of the Jacobian matrix using finite difference perturbations.

`EffectiveZero` A real number in the range $[10^{-20}, 10^{10}]$.

The magnitude of a variable below which absolute rather than relative perturbations are used – see parameters `FDPerturbation`, `SingPertFactor` and `SLRFactor` below.

`FDPerturbation` A real number in the range $[10^{-20}, 10^{10}]$.

Finite difference perturbation factor. If finite difference calculation of partial derivatives with respect to a variable x is required, x is perturbed by:

$$\texttt{FDPerturbation} \times |x|$$

unless $|X|$ is less than `EffectiveZero` (see above), in which case it is perturbed by `FDPerturbation`.

`SingPertFactor` A real number in the range $[10^{-20}, 10^{10}]$.

The perturbation factor used for escaping from local singularities. If, at a certain iteration, the Jacobian matrix is found to be singular (with a rank r that is less than the size of the system n), the solver attempts to escape from such a point by applying a perturbation to $n - r$ of the system variables. For a variable x, the size of this perturbation is:

$$\texttt{SingPertFactor} \times |x|$$

unless $|x|$ is less than `EffectiveZero` (see above), in which case it is perturbed by `SingPertFactor`.

`SLRFactor` A real number in the range $[10^{-20}, 10^{10}]$.

The step length restriction factor. In the interests of improving convergence from poor initial guesses, the solver automatically limits the step taken in any iteration by a fraction $\alpha \in (0, 1]$ so that the magnitude of the change in any variable x does not exceed:

- $\beta|x|$ if x is equal to, or exceeds the `EffectiveZero` (see above);
- β otherwise.

The factor β is set to `SLRFactor` at the start of the solution. Thereafter, it is adjusted automatically to reflect the difficulty of solving the system:

- β is reduced if corrective action needs to be taken (see parameter `MaxIterNoImprove` above);
- β is increased if fast reduction in the equation norm is observed from one iteration to the next.

`LASolver` A quoted string specifying a linear algebraic equation solver.

The solver to be used for the solution of linear algebraic equations at every iteration. This can be either one of the standard gPROMS linear algebraic equation solvers (*cf.* section 10.3) or a third-party linear algebraic equation solver (see the gPROMS System Programmer Guide). The default is `MA48`.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

`IdentityElimination` A boolean value.

If set to TRUE, the solver will attempt to internally reduce the size of the problem by removing equations of the form $x = y$, and substituting all occurences of one of these variables with the other. Such equations are often introduced in gPROMS models by stream connectivity equations. This may result in faster solution time although at the current stage of development the costs of creating and using the reduced system sometimes outweigh the benefits, particularly when many IF and CASE conditions are present.

If using `SPARSE` as the `BlockSolver` parameter of `BDNLSOL`, it is better to set this parameter in `BDNLSOL` rather than `SPARSE`, so that the eliminations are carried out on the complete system.

`IterationsWithoutNewJacobian` An integer in the range [0, 1000000].

If set to 0 `SPARSE` computes the Jacobian at every iteration. Otherwise, `SPARSE` will use a simple form of Modified Newton keeping the Jacobian for a set number of iterations. In some cases this can be used to speed up the solution.

10.5 Standard solvers for differential-algebraic equations

There are two standard mathematical solvers for the solution of mixed sets of differential and algebraic equations in gPROMS, namely `DASOLV` and `SRADAU`:

- `DASOLV` is based on variable time step/variable order Backward Differentiation Formulae (BDF). This has been proved to be efficient for a wide range of problems. However, BDF solvers suffer from loss of stability for certain types of problems (*e.g.* highly oscillatory ones) and they are not very efficient for problems with frequent discontinuities.
- `SRADAU` implements a variable time step, fully-implicit Runge-Kutta method. It has been proved to be efficient for the solution of problems arising from the discretisation of PDAEs with strongly advective terms (in general, highly oscillatory ODEs), and models with frequent discontinuities.

Both of the above solvers are designed to deal with large, sparse systems of equations in which the variable values are restricted to lie within specified lower and upper bounds. Moreover, they can handle situations in which some of the partial derivatives of the equations with respect to the variables are available analytically while the rest have to approximated[8]. Efficient finite difference approximations are used for the latter purpose.

Both solvers automatically adjust each time step taken so that the following criterion is satisfied:

$$\sqrt{\frac{1}{n_d}\sum_{i=1}^{n_d}\left(\frac{\epsilon_i}{a+r|x_i|}\right)^2} \leq 1$$

where:

- n_d is the number of differential variables in the problem (*i.e.* those that appear as `$x` in the gPROMS model);
- ϵ_i is the solver's estimate for the local error in the i^{th} differential variable;
- x_i is the current value the i^{th} differential variable;
- a is an absolute error tolerance;
- r is a relative error tolerance.

In rough terms, this means that the error ϵ_i incurred in a particular variable x_i over a single time step is not allowed to exceed $a + r|x_i|$. The default values for a and r (10^{-5} in both cases) are usually adequate since:

- they control the error in variables x_i of size 0.01 or higher to within acceptable ranges;
- smaller variable values are often not important from an engineering point of view[9].

[8] In gPROMS models, almost all partial derivatives are computed analytically from expressions derived using symbolic manipulations. The main exception is partial derivatives of equations involving any Foreign Object methods that are not capable of returning partial derivatives (see chapter 4 of the gPROMS Advanced User Guide).

[9] For example, a liquid level in a processing vessel of 10^{-4}m is practically indistinguishable from one of 10^{-5}m.

However, for problems in which small variable values may have an important effect on system behaviour, it is advisable to specify a smaller absolute tolerance[10].

The `DASolver` solution parameter may be used to change and/or configure the solver used for simulation activities (*cf.* section 10.2.2), as well as the default DAE sub-solver used by all higher-level solvers (*cf.* section 10.2.5). If this parameter is not specified, then the `DASOLV` solver is used, with the default configuration shown at the start of section 10.5.1 below.

10.5.1 The `DASOLV` solver

The algorithmic parameters used by `DASOLV` along with their default values are shown below. This is followed by a detailed description of each parameter.

```
"DASOLV" ["OutputLevel"                          := 0;
          "AbsoluteTolerance"                    := 1E-5;
          "RelativeTolerance"                    := 1E-5;
          "EventTolerance"                       := 1E-5;
          "EffectiveZero"                        := 1E-5;
          "FDPerturbation"                       := 1E-6;
          "SenErr"                               := FALSE;
          "Absolute1stTimeDerivativeThreshold"   := 0.0;
          "Relative1stTimeDerivativeThreshold"   := 0.0;
          "Relative2ndTimeDerivativeThreshold"   := 0.0;
          "VariablesWithLargestCorrectorSteps"   := 5;
          "LASolver"                             := MA48;
          "InitialisationNLSolver"               := "BDNLSOL";
          "ReinitialisationNLSolver"             := "BDNLSOL"]
```

However, `NLSOL` is used as the default `InitialisationNLSolver` and `ReinitialisationNLSolver` when `DASOLV` is used for *simulation* activities.

`OutputLevel` An integer in the range [-1, 7].

The amount of information generated by the solver. The following table indicates the lowest level at which different types of information are produced:

[10]For example, in a problem involving free radicals or ions, it may be important to distinguish between mole fraction of 10^{-6} and 10^{-7}.

0	(None)
1	(Re-)initialisation times, projection of predictor onto bounds, variables hitting bounds
2	Successful initialisation, change of branch in `IF` conditional equations, location of discontinuities, step failures, repeated convergence failures, predictor outside bounds, predictor step reduction, variables stuck on bounds
3	Detail of convergence failures, values of derivatives on commencing integration, number of perturbation groups, step length reduction due to bounds violation
4	Variable causing discontinuity, detail of perturbation groups
5	Entry to main integrator routines, all error test values, nonfatal singularities during integration, greatest changes in variables at each corrector iteration
6	Convergence values at every corrector iterations, step change factors
7	Time, step, variables, derivatives and residuals at every corrector iteration

`AbsoluteTolerance` A real number in the range $[10^{-20}, 10^{10}]$.

The absolute integration tolerance. Together with the parameter `RelativeTolerance` (see below), they determine whether or not a time step taken by the solver is sufficiently accurate. See the introduction to section 10.5 for more details.

`RelativeTolerance` A real number in the range $[10^{-20}, 10^{10}]$.

The relative integration tolerance. Together with the parameter `AbsoluteTolerance` (see above), they determine whether or not a time step taken by the solver is sufficiently accurate. See the introduction to section 10.5 for more details.

`EventTolerance` A real number in the range $[10^{-20}, 10^{10}]$.

The event tolerance, *i.e.* the maximum time interval within which discontinuities during integration are located.

`EffectiveZero` A real number in the range $[10^{-20}, 10^{10}]$.

The magnitude of a variable below which absolute rather than relative finite difference perturbation is used – see parameter `FDPerturbation` below.

`FDPerturbation` A real number in the range $[10^{-20}, 10^{10}]$.

Finite difference perturbation factor. If finite difference calculation of partial derivatives with respect to a variable X is required, X is perturbed by:

$$\texttt{FDPerturbation} \times |X|$$

unless $|X|$ is less than `EffectiveZero`, in which case it is perturbed by `FDPerturbation`.

`Diag` A boolean value.

Specifies whether very detailed diagnostic information is to be generated during integration.

`SenErr` A boolean value.

For optimisation type activities: specifies whether the sensitivity error test is to be applied at each step of the integration.

`Absolute1stTimeDerivativeThreshold` A real number in the range [0, 10^{10}].

Unless this parameter has the value zero, it represents the value θ_A in the condition used to determine reporting of potential "runaway" derivatives — see Figure 10.2.

If it is zero (the default), no runaway derivatives will be reported.

`Relative1stTimeDerivativeThreshold` A real number in the range [0, 10^{10}].

Represents the value θ_R in the condition used to determine reporting of potential "runaway" derivatives — see Figure 10.2.

`Relative2ndTimeDerivativeThreshold` A real number in the range [0, 10^{10}].

Represents the value θ_2 in the conditions used to determine reporting of potential "runaway" derivatives: see Figure 10.2.

If a subset of the system being solved by DASOLV becomes unstable, then DASOLV may fail, issuing a "repeated error test failure" message. This indicates that the code is no longer able to control the error of integration.
The variables associated with such instabilities often exhibit excessively large values of their first and second time derivatives immediately preceding the instability. DASOLV exploits this fact to help in the diagnosis of such problems. More specifically, DASOLV will report any variable X which satisfies **all three** of the following tests:

1. *Minimum magnitude*: The value of $|X|$ must be greater than 10^{-5}
2. *First derivative*:
   - EITHER
     $$|\dot{X}| > \theta_A$$
     (specified with the `AbsoluteDerivativeThreshold` parameter)
   - OR
     $$\frac{|\dot{X}|}{|X|} > \theta_R$$
     (specified with the `RelativeDerivativeThreshold` parameter)
3. *Second derivative*:
   $$\frac{|\ddot{X}|}{|\dot{X}|} > \theta_2$$
   (specified with the `RelativeSecondDerivativeThreshold` parameter)

Figure 10.2: Diagnosing error test failures in DASOLV

`VariablesWithLargestCorrectorSteps` A non-negative integer.

On rare occasions, DASOLV fails with a "corrector step failure" message. This indicates that the code is unable to establish a set of variable values that satify the system equations at a particular point. It is often caused by errors or bad scaling in some modelling equations which results in the corrector iterations taking excessively large steps in some of the variables.

To help with the diagnosis of such problems, DASOLV can report the variables with the largest relative change at each corrector iteration. The relative change for a variable X is defined as:

$$\frac{\delta X}{a + r|X|}$$

where:

- δX is the step in the variable at this corrector iteration;
- a is the absolute tolerance;
- r is the relative tolerance.

The parameter `VariablesWithLargestCorrectorSteps` specifies the number of variables to be reported in this manner. Note that such reporting takes place only if the parameter `OutputLevel` is set to a value of 5 or higher.

`LASolver` A quoted string specifying a linear algebraic equation solver.

The solver to be used for the solution of linear algebraic equations at each step of the integration. This can be either one of the standard gPROMS linear algebraic equation solvers (*cf.* section 10.3) or a third-party linear algebraic equation solver (see the gPROMS System Programmer Guide). The default is `MA48`.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

`InitialisationNLSolver` A quoted string specifying a nonlinear algebraic equation solver.

The solver to be used for the solution of nonlinear algebraic equations occurring at the initialisation stage of the integration. This can be either one of the standard gPROMS nonlinear algebraic equation solvers (*cf.* section 10.4) or a third-party nonlinear algebraic equation solver (see the gPROMS System Programmer Guide). The default is `NLSOL` when `DASOLV` is used for simulation activities, and `BDNLSOL` when it is used for optimisation or parameter estimation activities.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

`ReinitialisationNLSolver` A quoted string specifying a nonlinear algebraic equation solver.

The solver to be used for the solution of nonlinear algebraic equations that is necessary for re-initialisation following discontinuities. This can be either one of the standard gPROMS nonlinear algebraic equation solvers (*cf.* section 10.4) or a third-party nonlinear algebraic equation solver (see the gPROMS System Programmer Guide). The default is `SPARSE`.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

10.5.2 The `SRADAU` solver

The algorithmic parameters used by `SRADAU` along with their default values are shown below. This is followed by a detailed description of each parameter.

```
"SRADAU" ["OutputLevel"               := 0;
          "AbsoluteTolerance"         := 1E-5;
          "RelativeTolerance"         := 1E-5;
          "EventTolerance"            := 1E-5;
          "Diag"                      := FALSE;
          "LASolver"                  := MA48;
          "InitialisationNLSolver"    := "NLSOL";
          "ReinitialisationNLSolver" := "NLSOL"]
```

`OutputLevel` An integer in the range [0, 4].

The amount of information generated by the solver. The following table indicates the lowest level at which different types of information are produced:

0	(None)
1	(Re-)initialisation times, projection of predictor onto bounds, variables hitting bounds
2	Successful initialisation, change of branch in `IF` conditional equations, location of discontinuities, step failures, repeated convergence failures, predictor outside bounds, predictor step reduction, variables stuck on bounds
3	Detail of convergence failures, values of derivatives on commencing integration, number of perturbation groups, step length reduction due to bounds violation
4	Variable causing discontinuity, detail of perturbation groups

`AbsoluteTolerance` A real number in the range $[10^{-20}, 10^{10}]$.

The absolute integration tolerance. Together with the parameter `RelativeTolerance` (see below), they determine whether or not a time step taken by the solver is sufficiently accurate. See the introduction to section 10.5 for more details.

`RelativeTolerance` A real number in the range $[10^{-20}, 10^{10}]$.

The relative integration tolerance. Together with the parameter `AbsoluteTolerance` (see above), they determine whether or not a time step taken by the solver is sufficiently accurate. See the introduction to section 10.5 for more details.

`EventTolerance` A real number in the range $[10^{-20}, 10^{10}]$.

The event tolerance, *i.e.* the maximum time interval within which discontinuities during integration are located.

`Diag` A boolean value.

Specifies whether very detailed diagnostic information is to be generated during integration.

`LASolver` A quoted string specifying a linear algebraic equation solver.

The solver to be used for the solution of linear algebraic equations at each step of the integration. This can be either one of the standard gPROMS linear algebraic equation solvers (*cf.* section 10.3) or a third-party linear algebraic equation solver (see the gPROMS System Programmer Guide). The default is `MA48`.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

`InitialisationNLSolver` A quoted string specifying a nonlinear algebraic equation solver.

The solver to be used for the solution of nonlinear algebraic equations occurring at the initialisation stage of the integration. This can be either one of the standard gPROMS nonlinear algebraic equation solvers (*cf.* section 10.4) or a third-party nonlinear algebraic equation solver (see the gPROMS System Programmer Guide). The default is `NLSOL`.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

`ReinitialisationNLSolver` A quoted string specifying a nonlinear algebraic equation solver.

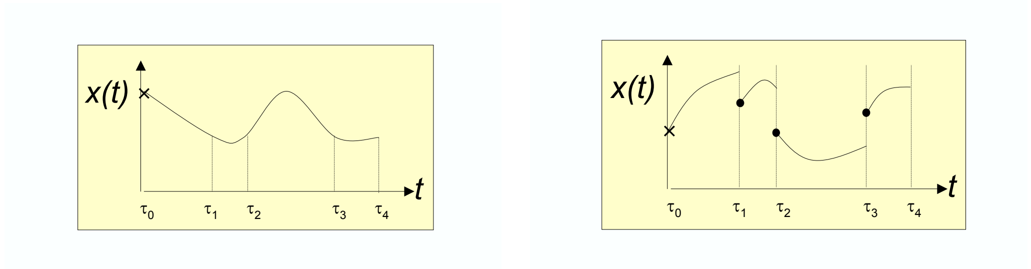
The solver to be used for the solution of nonlinear algebraic equations that is necessary for re-initialisation following discontinuities. This can be either one of the standard gPROMS nonlinear algebraic equation solvers (*cf.* section 10.4) or a third-party nonlinear algebraic equation solver (see the gPROMS System Programmer Guide). The default is `SPARSE`.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

10.6 Standard solvers for optimisation

There are two standard mathematical solvers for optimisation in gPROMS, namely `CVP_SS` and `CVP_MS`. `CVP_SS` can solve optimisation problems with both discrete and continuous decision variables ("mixed integer optimisation"). Both steady-state and dynamic problems are supported. `CVP_MS` can solve dynamic optimisation problems with continuous decision variables.

For dynamic optimisation problems, both `CVP_SS` and `CVP_MS` are based on a control vector parameterisation (CVP) approach which assumes that the time-varying control variables are piecewise constant (or piecewise linear) functions of time over a specified number of control intervals. The precise values of the controls over each interval, as well as the duration of the latter, are generally determined by the optimisation algorithm[11]. As the number of control variables is usually a small fraction of the total number of variables in the problem, the optimisation algorithm has to deal only with a relatively small number of decisions, which makes the CVP approach applicable to large problems.



(a) Single-shooting algorithm (b) Multiple-shooting algorithm

Figure 10.3: Single- *vs.* multiple-shooting algorithms

The `CVP_SS` solver implements a "single-shooting" dynamic optimisation algorithm. This involves the following steps (see figure 10.3(a)):

1. the optimiser chooses the duration of each control interval, and the values of the control variables over it;
2. starting from the initial point at time $t = 0$ (shown as a cross on the vertical axis in figure 10.3(a)), the dynamic system model is solved over the entire time horizon to determine the time-variation of all variables $x(t)$ in the system;
3. the above information is used to determine the values of[12]:
   - the objective function to be optimised;
   - any constraints that have to be satisfied by the optimisation;

[11]In addition, as explained in chapter 2 of the gPROMS Advanced User Guide, many dynamic optimisation problems involve time-invariant parameters that also have to be chosen by the optimiser.

[12]In practice, the solution of the model also needs to determine the values of the partial derivatives (sensitivities) of the objective function and constraints with respect to all the quantities specified by the optimiser.

4. based on the above, the optimiser revises the choices it made at the first step, and the procedure is repeated until convergence to the optimum is achieved.

The term "single-shooting" arises from the second step in the above algorithm which involves a single integration of the dynamic model over the entire horizon.

The CVP_MS solver implements a "multiple-shooting" dynamic optimisation algorithm with the following steps (see figure 10.3(b)):

1. the optimiser chooses the duration of each control interval, the values of the control variables over it, and, additionally, the values of the differential variables $x(t)$ at the start of each control interval other than the first one (shown as solid circles in figure 10.3(b));
2. for each control interval, starting from the initial point that is either known (for the first interval) or is chosen by the optimiser (for all subsequent intervals), the dynamic system model is solved over this control interval to determine the time-variation of all variables $x(t)$ in the system;
3. the above information is used to determine the values of:
    - the objective function to be optimised;
    - any constraints that have to be satisfied by the optimisation;
    - the discrepancies between the computed values of the variables $x(t)$ at the end of each interval and the corresponding values chosen by the optimiser at the start of the next interval;
4. based on the above, the optimiser revises the choices it made at the first step, and repeats the above procedure until it obtains a point that:
    - optimises the objective function;
    - satisfies all constraints;
    - ensures that all differential variables $x(t)$ are continuous at the control interval boundaries.

The "multiple-shooting" term reflects the fact that each control interval is treated independently at the second step above.

Both solvers, by default, employ the DASOLV code (*cf.* section 10.5.1) for the solution of the underlying DAE problem and the computation of its sensitivities. In principle, this can be replaced by a third-party solver with similar capabilities.

The choice between the CVP_SS and CVP_MS solvers for any dynamic optimisation problem depends primarily on the number of optimisation decision parameters that the algorithm has to deal with in computing the sensitivities of the model variables. In principle:

- CVP_MS should normally be preferred for problems with many time-varying control variables and/or many control intervals, but with relatively few differential ("state") variables;
- CVP_SS should normally be preferred for large problems (potentially involving several hundreds or thousands of differential ("state") variables) but with relatively few time-varying control variables and control intervals.

In practice, some experimentation may be required to determine the better algorithm for any particular application.

The `DOSolver` solution parameter may be used to change and/or configure the solver used for optimisation activities (*cf.* section 10.2.2). If this parameter is not specified, then the `CVP_SS` solver is used, with the default configuration shown at the start of section 10.6.1 below.

10.6.1 The `CVP_SS` solver

`CVP_SS` can solve steady-state and dynamic optimisation problems with both continuous and discrete optimisation decision variables. The algorithmic parameters used by `CVP_SS` along with their default values are shown below. This is followed by a detailed description of each parameter.

```
"CVP_SS"  [ "DASolver"    := "DASOLV";
            "MINLPSolver" := "OAERAP"];
```

`DASolver` A quoted string specifying a differential-algebraic equation solver.

The solver to be used for integrations of the model equations and their sensitivity equations at each iteration of the optimisation. This can be either the standard `DASOLV` solver (*cf.* section 10.5.1) or a third-party differential-algebraic equation solver (see the gPROMS System Programmer Guide). The default is `DASOLV`.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

`MINLPSolver` A quoted string specifying a mixed integer optimisation solver.

The solver to be used for mixed integer optimisation problems. This can be either the standard `OAERAP` solver (*cf.* section 10.6.2) or a third-party mixed integer optimisation solver (see the gPROMS System Programmer Guide). For optimisation problems that do not involve any discrete decision variables, this can be any `CAPE-OPEN` compliant solver that is capable of solving NLPs but not MINLPs, *e.g.*the standard NLP solver `SRQPD` (*cf.* section 10.6.3). The default is `OAERAP`.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

10.6.2 The `OAERAP` solver

The `OAERAP` solver employs an outer approximation (OA) algorithm for the solution of the MINLP. As outlined in Figure 10.4, this involves solving a sequence of simpler optimisation problems, including nonlinear programs (NLPs) at steps 1 and 3 and mixed integer linear programs (MILPs) at step 2. The OAERAP code has been designed so that it can make direct use of any CAPE-OPEN compliant NLP and MILP solvers (see the gPROMS System Programmer Guide) without the need for any additional interfacing or modification.

Given initial guesses for all optimisation decision variables, both discrete (y) and continuous (x):

Step 0: Initialisation

- Set the objective function of the best solution that is currently available, $\phi^{best} := +\infty$.
- Set the objective function of the best solution that may be obtained, $\phi^{LB} := +\infty$.

Step 1: Solve fully relaxed problem

- Solve a continuous optimisation problem (NLP) treating all discrete variables as continuous (i.e. allow them to take any value between their lower and upper bounds) to determine optimal values x^{FR}, y^{FR} of the optimisation decision variables and of the objective function, ϕ^{FR}.
- If above problem is infeasible, <u>terminate</u>: original problem is infeasible as posed.
- If all discrete optimisation decision variables have discrete values at the solution of the above problem, then <u>terminate</u>: optimal solution of original problem is (x^{FR}, y^{FR}) with an objective function value of ϕ^{FR}.

Step 2: Solve master problem

- Construct a mixed integer linear programming (MILP) problem which:
  - involves appropriate linearisations of the objective function and the constraints carried out at the solutions of all continuous optimisation problems solved so far,
  - excludes all combinations of discrete variable values that have been considered at step 2 so far.
- Solve the above MILP problem to determine optimal values of both the continuous and discrete variables x^{MP}, y^{MP}, and the corresponding value of the objective function ϕ^{MP}.
- If the above problem is infeasible or if $\phi^{best} - \phi^{MP} \leq \varepsilon \max(1, |\phi^{best}|)$, then <u>terminate</u>: there are no more combinations of discrete variables that can be usefully considered.
  - If $\phi^{best} = +\infty$, then original problem was infeasible.
  - Otherwise, the optimal solution is (x^{best}, y^{best}) with a corresponding objective function value of ϕ^{best}.
- The MILP provides an improved bound on the best solution that may be obtained; therefore, update $\phi^{LB} := \phi^{MP}$.

Step 3: Solve primal optimisation problem

- Fix all discrete optimisation decision variables to their current values.
- Solve continuous optimisation problem (NLP) to determine:
  - optimal value of objective function, ϕ^{PR};
  - optimal values of continuous optimisation decision variables, x^{PR}.
- If the above NLP is feasible and $\phi^{PR} < \phi^{best}$, then an improved solution to the original problem has been found; record its details by setting $\phi^{best} := \phi^{PR}$; $x^{best} := x^{PR}$; $y^{best} := y^{PR}$.

Step 4: Iterate

- Set the next set of values of the discrete optimisation decision variables to be considered $y^{PR} := y^{MP}$.
- Repeat from step 2.

Figure 10.4: Outline of the `OAERAP` algorithm for the solution of a `MINLP` problem (minimisation case)

The `OAERAP` solver also includes an equality relaxation (ER) scheme for handling equality constraints. It should be emphasised that, in the case of optimisation problems defined in gPROMS, this relaxation is applied only to any `ENDPOINT_EQUALITY` constraints that may appear in the `OPTIMISATION` Entity.

The algorithm described in Figure 10.4 is guaranteed to obtain the globally optimal solution to the optimisation problem posed only if the latter is convex. This is unlikely to be the case in many problems of engineering interest. An augmented penalty (AP) strategy is employed in order to increase the probability of a global solution being obtained.

The algorithmic parameters used by `OAERAP` along with their default values are shown below. This is followed by a detailed description of each parameter.

```
"OAERAP"  [ "OutputLevel"           := 0;
            "MaxIterations"         := 100000;
            "OptimisationTolerance" := 1E-4;
            "MILPSolver"            := "GLPK";
            "NLPSolver"             := "SRQPD"];
```

`OutputLevel` An integer in the range [-1, 0].

The amount of information generated by the solver. The following table indicates the lowest level at which different types of information are produced:

-1	(None)
0	Solution of fully relaxed point, solution of master problem, solution of primal optimisation problem, final solution

`MaxIterations` An integer in the range [1, 100000].

The maximum number of iterations involving step 2-4 of the algorithm described in Figure 10.4. This is essentially the maximum number of distinct alternatives to be considered by the algorithm.

`OptimisationTolerance` A real number in the range $[0.0, 1.0]$.

The optimisation tolerance ε used in the termination criterion at step 2 of the algorithm described in Figure 10.4.

`MILPSolver` A quoted string specifying a mixed integer linear programming solver.

Specifies a `CAPE-OPEN` compliant solver to be used for the solution of the mixed integer linear programming (MILP) problems at step 2 of the algorithm described in Figure 10.4.

`NLPSolver` A quoted string specifying a nonlinear programming solver.

Specifies a `CAPE-OPEN` compliant solver to be used for the solution of the nonlinear programming (NLP) problems at steps 1 and 3 of the algorithm described in Figure 10.4.

10.6.3 The `SRQPD` solver

The `SRQPD` solver employs a sequential quadratic programming (SQP) method for the solution of the nonlinear programming (NLP) problem. The algorithmic parameters used by `SRQPD` along with their default values are shown below. This is followed by a detailed description of each parameter.

```
"SRQPD"  [ "OutputLevel"           := 0;
                           "MaxFun" :=  10000,
            "MaximumLineSearchSteps" :=  20,
        "MinimumLineSearchStepLength" :=  1e-005,
        "InitialLineSearchStepLength" :=  1.0,
             "OptimisationTolerance" :=  0.001,
                          "Scaling" :=  0];
```

`OutputLevel` An integer in the range [-1, 4].

The amount of information generated by the solver. The following table indicates the lowest level at which different types of information are produced:

-1	(None)
0	Failed integrations and initialisations, optimisation failure, summary information from the SRQPD nonlinear programming code, final solution point and constraint values, best available point after failure
1	Values of optimisation decision variables, objective function and constraints in each major optimisation iteration
2	Start and end times of each interval of integration, optimisation decision variables and objective function at each line search trial
3	Derivatives of objective function and constraints

`MaxFun` An integer in the range [0, 100000].

The maximum number of *optimiser* function evaluations (*i.e.* solutions of the underlying steady-state or dynamic model) to perform before halting the solution process (if no optimum has been found by that point).

`MaximumLineSearchSteps` An integer in the range [1, 100].

The maximum number of line search steps in one optimisation iteration.

`MinimumLineSearchStepLength` A real number in the range $[10^{-10}, 1.0]$.

The minimum length of a line search step.

`InitialLineSearchStepLength` A real number in the range $[10^{-10}, 1.0]$.

The length of the line search step for the first optimisation iteration. An initial line search step length less than 1 is recommended when the initial approximation of the Hessian (*i.e.* identity matrix) is very different to the actual values in the Hessian. This could result in a very large initial step and therefore several line search trials before the optimiser finds a better point.

`Scaling` An integer in the range [0, 3].

The form of scaling to be applied to the optimisation decision variables, including control variables, time-invariant parameters, the length of the time horizon and the lengths of individual control intervals. These decision variables may vary significantly in magnitude, which may adversely affect the performance of the optimisation algorithms. Consequently, appropriate scaling of the optimisation decision variables is strongly recommended[13].

The scaling performed is of the general mathematical form:

[13] A useful indication as to whether scaling is necessary is the condition number estimate that is printed out at each iteration of the optimisation calculation. It is recommended that scaling be undertaken for problems with condition numbers exceeding 10^{10}.

$$\tilde{q}_j \equiv \frac{q_j - c_j}{d_j} \tag{10.1}$$

where q_j is the jth original optimisation decision variable and $\tilde{q}_j$ is the corresponding scaled decision variable. The constants c_j and d_j are determined automatically depending on the value of `Scaling`, as described below:

- **`Scaling` = 0:** No scaling (default).

$$\begin{aligned} d_j &= 1, \\ c_j &= 0. \end{aligned}$$

- **`Scaling` = 1:** Scaling according to the ranges of the optimisation decision variables so that the scaled variables vary between -1 and 1.

$$\begin{aligned} d_j &= \frac{1}{2}\left(q_j^{\max} - q_j^{\min}\right), \\ c_j &= \frac{1}{2}\left(q_j^{\max} + q_j^{\min}\right) \end{aligned}$$

- **`Scaling` = 2:** Scaling according to the initial guesses of the optimisation variables.

$$\begin{aligned} d_j &= \begin{cases} q_j^0 & \text{if } |q_j^0| > \varepsilon, \\ \frac{1}{2}\left(q_j^{\max} - q_j^{\min}\right) & \text{otherwise} \end{cases} \\ c_j &= 0 \end{aligned}$$

  where q_j^0 is the initial guess for the jth optimisation variable and ε is a small constant (currently set at 10^{-8}).

- **`Scaling` = 3:** Scaling according to the value and the gradients of the objective function Φ at the initial guess.

$$\begin{aligned} d_j &= \begin{cases} \frac{1+|\Phi(q^0)|}{2\left|\frac{\partial\Phi}{\partial q_j}\right|_{q^0}} & \text{if } \left|\frac{\partial\Phi}{\partial q_j}\right|_{q^0} > \varepsilon, \\ \frac{1}{2}\left(q_j^{\max} - q_j^{\min}\right) & \text{otherwise} \end{cases} \\ c_j &= 0 \end{aligned}$$

  where q^0 is the vector of initial guesses of the optimisation decision variables and ε is a small constant (currently set at 10^{-8}).

`OptimisationTolerance`[14] A real number in the range [0.0, 1.0].

The solution tolerance for the optimisation. Convergence is deemed to occur when a linear combination of the gradients of the Lagrangian function on one hand, and the violation of the constraints on the other, drops below this tolerance. More specifically, the convergence criterion used is:

$$\frac{1}{|\Phi^*| + 1.0}\left(\left|\sum_j \frac{\partial\Phi^*}{\partial q_j}\delta q_j\right| + \sum_{j\in\mathcal{E}} |\lambda_j(x_j^* - \tilde{x}_j)| + \sum_{j\in\mathcal{I}} |\mu_j| \max(0, x_j^L - x_j^*, x_j^* - x_j^U)\right)$$

[14]The alternative spelling `OptimizationTolerance` is also recognized as well as the old spelling `OptTol`.

$$+\sum_{j\in\mathcal{E}}|x_j^* - \tilde{x}_j| + \sum_{j\in\mathcal{I}}\max(0, x_j^L - x_j^*, x_j^* - x_j^U) \leq \texttt{OptTol}$$

where:

- q_j is the jth optimisation decision variable (including time-invariant parameters, control variable parameterisations, and control interval durations);
- Φ^* is the final value of the objective function;
- δq_j is the step taken in variable q_j at the last iteration of the optimisation calculation;
- x_j^* are the final values of the quantities x_j that are subject to equality and inequality constraints in the optimisation[15];
- $\mathcal{E}$ is the subset of x_j that are constrained to be equal to specified values $\tilde{x}_j$;
- $\mathcal{I}$ is the subset of x_j that are constrained to lie between lower and upper bounds x_j^L and x_j^U respectively;
- λ_j is the Lagrange multiplier that corresponds to the equality constraint imposed on variable $x_j, j \in \mathcal{E}$;
- μ_j is the Lagrange multiplier that corresponds to the bound constraints imposed on variable $x_j, j \in \mathcal{I}$.

10.6.4 The CVP_MS solver

The algorithmic parameters used by CVP_MS along with their default values are shown below. This is followed by a detailed description of each parameter[16].

```
"CVP_MS" [ "OutputLevel"            := 0;
           "MaxFun"                 := 10000;
           "SQPMinAlpha"            := 1E-10;
           "SQPHeLa"                := "BFGS";
           "SQPHeLaEps"             := 1E-10;
           "SQPHeLaScale"           := TRUE;
           "SQPHeLaEigenCtrl"       := TRUE;
           "SQPMaxIters"            := 500;
           "SQPMaxInfIters"         := 10;
           "SQPWatchdogStart"       := 10;
           "SQPWatchdogCredit"      := 0;
           "SQPWatchdogLogging"     := FALSE;
           "SQPSolver"              := "Powell";
           "SQPQPSolver"            := "Franke";
           "QPEps"                  := 1E-10;
           "QPMatSolver"            := "RedSpBKP";
           "QPMaxIters"             := 250;
           "OptTol"                 := 1E-3;
           "InfDefault"             := 1E10;
           "NumSen"                 := FALSE;
```

[15] This set normally includes all of the optimisation decision variables q, as well as any model variables which are subject to equality and inequality constraints.

[16] For full details, please consult the information at http://www.systemtechnik.tu-ilmenau.de/~fg_opt/omuses/omuses.html.

```
                              "SetBounds"                := FALSE;
                              "NeedLagrangeMultipliers" := FALSE;
                              "DASolver"                 := "DASOLV"];
```

`OutputLevel` An integer in the range [0, 4].

The amount of information generated by the solver. The following table indicates the lowest level at which different types of information are produced:

0	Failed integrations and initialisations, optimisation failure, summary information from the HQP nonlinear programming code, final solution point and constraint values, best available point after failure
1	Values of optimisation decision variables, objective function and constraints in each major optimisation iteration
2	For each multiple-shooting interval in each major optimisation iteration: the values of the optimisation decision variables, the values and derivatives of the matching conditions, the constraints and the objective function.

`MaxFun` An integer in the range [0, 100000].

The maximum number of *optimiser* function evaluations (*i.e.* solutions of the underlying dynamic model) to perform before halting the solution process (if no optimum has been found by that point).

`SQPMinAlpha` A real number in the range $[0, 10^5]$.

Lower limit for the step length in the line search of the sequential quadratic programming (SQP) sub-solver.

`SQPHeLa` A quoted string.

The method to be used for constructing the approximation of the Hessian matrix of the Lagrangian. Permitted values are:

- `"BFGS"`: partitioned BFGS update with Powell's damping
- `"DScale"`: numerical approximation with a diagonal matrix

`SQPHeLaEps` A real number in the range $[0, 10^5]$.

ϵ parameter used in SQP algorithm, see the Omuses document.

`SQPHeLaScale` A boolean value.

Use "`DScale`" approach to initialise Hessian rather than setting it to the identity matrix.

`SQPHeLaEigenCtrl` A boolean value.

Control of positive definite Hessian blocks based on eigenvalues (only used with the `BFGS` method).

`SQPMaxIters` An integer in the range [0, 100000].

Total number of SQP iterations allowed.

`SQPMaxInfIters` An integer in the range [0, 100000].

Number of infeasible SQP iterations (*i.e.* points where the integration fails) allowed before failure.

`SQPWatchdogStart` An integer in the range [0, 100000].

Iteration at which to start watchdog algorithm if using `"Powell"` algorithm (see `SQPSolver` parameter below).

`SQPWatchdogCredit` An integer in the range [0, 100000].

Number of "bad" iterations until backtracking and regular step are performed (0 means disable watchdog).

`SQPWatchdogLogging` A boolean value.

Specifies whether watchdog log output should be produced.

`SQPSolver` A quoted string.

The type of sequential quadratic programming (SQP) algorithm to be used for the optimisation. Permitted values:

- `"Powell"`
- `"Schittkowski"`

`SQPQPSolver` A quoted string.

The type of quadratic programming (QP) solver to be used at each iteration of the optimisation. Permitted values are:

- `"Franke"`
- `"Mehrotra"`

`QPEps` A real number in the range $[0, 10^5]$.

The tolerance to which the quadratic programming sub-problems are to be solved.

`QPMatSolver` A quoted string.

The matrix solver to use for the solution of the quadratic programming sub-problems in the optimisation. Permitted values are (refer to Omuses document for details):

- `"SpBKP"`
- `"RedSpBKP"`
- `"SpSC"`
- `"LQDOCP"`

`QPMaxIters` An integer in the range [0, 100000].

Maximum number of QP iterations to attempt.

`OptTol` A real number in the range [0.0, 1.0].

The solution tolerance for the optimisation.

`InfDefault` A real number in the range $[0, 10^{35}]$.

Upper and lower bounds greater than this value in magnitude are treated as $\pm\infty$ (as appropriate).

`NumSen` A boolean value.

Specifies whether sensitivities should be calculated numerically – *i.e.* by repeated "normal" integrations with perturbed values – rather than 'analytically', *i.e.* with a special sensitivity integration. **Not recommended** except perhaps for large problems with very few parameters per interval.

`DASolver` A quoted string specifying a differential-algebraic equation solver.

The solver to be used for integrations of the model equations and their sensitivity equations at each stage and each iteration of the optimisation. This can be either the standard `DASOLV` solver (*cf.* section 10.5.1) or a third-party differential-algebraic equation solver (see the gPROMS System Programmer Guide). The default is `DASOLV`.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

10.7 Standard solvers for parameter estimation

There is currently only one standard mathematical solver for parameter estimation in gPROMS, namely `MXLKHD`. This is based on a general maximum likelihood approach. More details are given in chapter 3 of the gPROMS Advanced User Guide.

The `PESolver` solution parameter may be used to configure the parameter estimation solver (*cf.* section 10.2.2). The default configuration is shown at the start of section 10.7.1 below.

10.7.1 The `MXLKHD` solver

The algorithmic parameters used by `MXLKHD` along with their default values are shown below. This is followed by a detailed description of each parameter.

```
        "MXLKHD" [ "OutputLevel"  := 0;
                   "OptTol"       := 1E-3;
                   "MaxFun"       := 10000;
         "MaximumLineSearchSteps" :=  20,
    "MinimumLineSearchStepLength" :=  1e-005,
                   "Scaling"      := 0;
                   "Statistics"   := 0;
                   "DASolver"     := "DASOLV" ]
```

`OutputLevel` An integer in the range [-1, 4].

The amount of information generated by the solver. The following table indicates the lowest level at which different types of information are produced:

-1	(None)
0	Failed integrations and initialisations, optimisation failure, summary information from the SQP nonlinear programming code, final solution point and constraint values, best available point after failure
1	values of optimisation decision variables, objective function and constraints in each major optimisation iteration
3	The derivatives of the objective function and constraints

`OptTol` A real number in the range [0.0, 1.0].

The solution tolerance for the parameter estimation. Convergence is deemed to occur when the following convergence criterion is satisfied:

$$\frac{1}{|\Phi^*|+1.0}\left(\left|\sum_j \frac{\partial \Phi^*}{\partial \theta_j}\delta\theta_j\right| + \sum_j |\mu_j| \max(0, \theta_j^L - \theta_j^*, \theta_j^* - \theta_j^U)\right)$$

$$\sum_j \max(0, \theta_j^L - \theta_j^*, \theta_j^* - \theta_j^U) \leq \texttt{OptTol}$$

where:

- θ_j is the jth parameter to be estimated (including both model parameters and variance model parameters);
- θ_j^* is the final value of parameter θ_j;
- θ_j^L is the lower bound imposed on parameter θ_j;
- θ_j^U is the upper bound imposed on parameter θ_j;
- Φ^* is the final value of the maximum likelihood objective function;
- $\delta\theta_j$ is the step taken in parameter θ_j at the last iteration of the parameter estimation calculation;
- μ_j is the Lagrange multiplier that corresponds to the bound constraints imposed on parameter θ_j;

`MaxFun` An integer in the range [0, 100000].

The maximum number of *optimiser* function evaluations (*i.e.* solutions of the underlying dynamic model) to perform before halting the solution process (if no optimum has been found by that point).

`MaximumLineSearchSteps` An integer in the range [1, 100].

The maximum number of line search steps in one optimisation iteration.

`MinimumLineSearchStepLength` A real number in the range $[10^{-10}, 1.0]$.

The minimum lenght of a line search step.

`Scaling` An integer in the range [0, 3].

The parameters to be determined in the context of a single parameter estimation problem may vary significantly in magnitude, which may adversely affect the performance of the optimisation algorithms. Consequently, appropriate scaling of these parameters is strongly recommended[17].

The scaling performed is of the general mathematical form:

$$\tilde{\theta}_j = \frac{\theta_j - c_j}{d_j} \tag{10.2}$$

where θ_j is the jth original parameter to be estimated and $\tilde{\theta}_j$ the corresponding scaled parameter. The constants c_j and d_j are determined automatically depending on the value of `Scaling`, as described below:

- **`Scaling` = 0:** No scaling (default).

$$\begin{aligned} d_j &= 1, \\ c_j &= 0. \end{aligned}$$

- **`Scaling` = 1:** Scaling according to the ranges of the parameters so that the scaled parameters vary between −1 and 1.

$$\begin{aligned} d_j &= \frac{1}{2}\left(\theta_j^{\max} - \theta_j^{\min}\right), \\ c_j &= \frac{1}{2}\left(\theta_j^{\max} + \theta_j^{\min}\right) \end{aligned}$$

[17] A useful indication as to whether scaling is necessary is the condition number estimate that is printed out at each iteration of the optimisation calculation. It is recommended that scaling be undertaken for problems with condition numbers exceeding 10^{10}.

- `Scaling` = **2:** Scaling according to the initial guesses of the parameters.

$$
\begin{aligned}
d_j &= \begin{cases} \theta_j^0 & \text{if } |\theta_j^0| > \varepsilon, \\ \frac{1}{2}\left(\theta_j^{\max} - \theta_j^{\min}\right) & \text{otherwise} \end{cases} \\
c_j &= 0
\end{aligned}
$$

  where θ_j^0 is the initial guess for the jth parameter and ε is a small constant (currently set at 10^{-8}).

- `Scaling` = **3:** Scaling according to the value and the gradients of the objective function at the initial guess.

$$
\begin{aligned}
d_j &= \begin{cases} \frac{1+|\Phi(\theta^0)|}{2\left|\frac{\partial\Phi}{\partial\theta_j}\right|_{\theta^0}} & \text{if } \left|\frac{\partial\Phi}{\partial\theta_j}\right|_{\theta^0} > \varepsilon, \\ \frac{1}{2}\left(\theta_j^{\max} - \theta_j^{\min}\right) & \text{otherwise} \end{cases} \\
c_j &= 0
\end{aligned}
$$

  where θ^0 is the vector of initial guesses of the parameters and ε is a small constant (currently set at 10^{-8}).

`Statistics` An integer in the range [0, 2].

Controls the information produced by the solver in the machine-readable estimation statistics (`.stat-mr`) file at the end of the computation (*cf.* section 3.6.1.4 of the gPROMS Advanced User Guide):

0 The parameter vector used for the calculation of the variance/covariance matrices includes both the model parameters and the variance model parameters.

1 The parameter vector used for the calculation of the variance/covariance matrices includes only the model parameters.

2 The variance/covariance matrices are not calculated.

`DASolver` A quoted string specifying a differential-algebraic equation solver.

The solver to be used for integrations of the model equations and their sensitivity equations at each iteration of the parameter estimation. This can be either the standard `DASOLV` solver (*cf.* section 10.5.1) or a third-party differential-algebraic equation solver (see the gPROMS System Programmer Guide). The default is `DASOLV`.

This parameter can be followed by further specifications aimed at configuring the particular solver by setting values to its own algorithmic parameters (*cf.* section 10.2.4).

Appendix A

Model Analysis and Diagnosis

Contents

A.1 Introduction

At the start of each simulation, gPROMS analyses the mathematical model so as to assist the user in identifying structural problems and errors in the modelling and/or the problem specification. In particular, gPROMS attempts to determine:

- if the model is well-posed and whether alternative specifications are required for the degrees-of-freedom;
- if the underlying set of differential and algebraic equations is of index exceeding 1; and
- if the initial conditions are inconsistent.

These structural problems are considered in more detail below.

A.2 Well-posed models and degrees-of-freedom

A.2.1 Case I: over-specified systems

An over-specified system is one which either itself consists of more equations than unknown variables, or involves an over-specified sub-set of equations and unknowns.

Mathematically, it can be shown that *a*ny over-specified system will contain at least one sub-system involving k equations in only $(k-1)$ distinct unknowns. gPROMS identifies this sub-system and, where appropriate, offers informed suggestions on which `ASSIGN`ments may be responsible for the over-specification.

As a simple example of this, consider the gPROMS input shown in figure A.1. It is easy to see that `MODEL mod1` consists of 4 equations in 4 variables, one of which, `y2`, is ASSIGNed in the `PROCESS proc`. Execution of the `PROCESS proc` leads to the following diagnostic message:

```
Executing process PROC...

All 4 variables will be monitored during this simulation!

Building mathematical problem description took 0.014 seconds.

Loaded MA48 library
Execution begins....

   Variables
       Known            :      1
       Unknown          :      3
          Differential  :      2
          Algebraic     :      1

   Model equations      :      4
   Initial conditions   :      2


Checking consistency of model equations and ASSIGN specifications...

   ERROR: Part of your problem is over-specified.
          The following 3 equation(s) involve only 2 unknown variable(s).
              Model Equation         1: MYMOD.$X1 = MYMOD.X1 * MYMOD.Y1 ;
              Model Equation         3: MYMOD.X1^2 = MYMOD.Y2 ;
              Model Equation         4: 0 = MYMOD.Y1 - MYMOD.Y2 ;

          The 2 unknown(s) occuring in these 3 equations are:

              MYMOD.Y1 (ALGEBRAIC)
              MYMOD.X1 (STATE)

          The problem may have been caused because you ASSIGNed the
          following variable(s):
              MYMOD.Y2 (INPUT)
```

```
Initialisation calculation failed.

Execution of PROC fails prematurely.
```

gPROMS identifies the over-specified sub-system of 3 equations in 2 unknown variables and suggests that the `ASSIGN`ment of the variable `y2` is causing the problem. Un-`ASSIGN`ing this variable leads to a working simulation.

A.2.2 Case II: under-specified systems

An under-specified system has more unknown variables than equations. In this case, gPROMS diagnoses the problem and provides a list of candidate variables for `ASSIGN`ment (while advising against `ASSIGN`ing differential variables). This is illustrated by the gPROMS v1.x example shown in figure A.2. gPROMS issues the following message upon execution of the `PROCESS proc`:

```
Executing process PROC...

All 5 variables will be monitored during this simulation!

Building mathematical problem description took 0.001 seconds.

Loaded MA48 library
Execution begins....

   Variables
       Known            :      0
       Unknown          :      5
          Differential  :      2
          Algebraic     :      3

   Model equations      :      4
   Initial conditions   :      2


Checking consistency of model equations and ASSIGN specifications...

   ERROR: Your problem is underspecified.
          You need to ASSIGN 1 of the following unknown variables:
             MYMOD.X2   *** Not recommended ***
             MYMOD.Y3
Initialisation calculation failed.

Execution of PROC fails prematurely.
```

`ASSIGN`ing the algebraic variable `y3` leads to a well-posed system.

```
DECLARE
  TYPE
    NoType       = 0.5 :   -1E30 :  1E30
END #declare
#================================================================
MODEL mod1

   VARIABLE
      x1, x2, y1, y2 AS NoType

   EQUATION

      $x1 = x1*y1 ;

      $x2 = x1 + x2*y1 + y2 ;

      x1^2 = y2 ;

      0 = y1 - y2 ;

END #model
#================================================================
PROCESS proc

   UNIT
      mymod    AS mod1

   ASSIGN
      WITHIN mymod DO
         y2 := 3 ;
      END #within

   INITIAL
      WITHIN mymod DO
         x1 = 0 ;
         x2 = 0 ;
      END #within

   SOLUTIONPARAMETERS
      ReportingInterval := 1 ;

   SCHEDULE
      CONTINUE FOR 10

END #process
```

Figure A.1: Illustrative example: over-specified system

```
DECLARE
  TYPE
    NoType       = 0.5 :   -1E30 :  1E30
END #declare
#================================================================
MODEL mod1

   VARIABLE
      x1, x2, y1, y2, y3                AS NoType

   EQUATION

      $x1 = x1*y1 ;

      $x2 = x1 + x2*y1 + y2 + 3*y3 ;

      x1^2 = y2 ;

      0 = y1 - y2 ;

END #model
#================================================================
PROCESS proc

   UNIT
      mymod    AS mod1

   INITIAL
      WITHIN mymod DO
         x1 = 0 ;
         x2 = 0 ;
      END #within

   SOLUTIONPARAMETERS
      OutputLevel := 2 ;
      gRMS := OFF ;
      ReportingInterval := 1 ;


   SCHEDULE
      CONTINUE FOR 10

END #process
```

Figure A.2: Illustrative example: under-specified system

A.3 High-index DAE systems

Consistent initialisation of DAE systems is often related to their *index*. The index of a DAE system is defined as the minimum number of differentiations with respect to time that are necessary in order to obtain the time derivatives of *all* variables, *i.e.* to reduce the system to a set of ordinary differential equations (ODEs). Index-1 systems are generally very similar to ODEs in that the number of initial conditions that can be specified arbitrarily is equal to the number of differential variables in the system, all the differential variables may be given arbitrary initial values, and similar numerical methods can be used for the solution of the system. On the other hand, in "high-index" DAEs (index > 1), the number of initial conditions that can be specified arbitrarily may be *less than* the number of differential variables, the differential variables are not independent and ODE-type numerical methods may fail. For the latter case, gPROMS:

- lists the subset of differential variables that cannot be given arbitrary initial values;
- indicates the sub-system of k equations in $(k-1)$ algebraic variables and time derivatives of differential variables that is responsible; and
- lists any variables that are `ASSIGN`ed in the above sub-system, since it is often the choice of `ASSIGN`ments that causes high-index problems.

In order to illustrate this, consider the gPROMS input shown in figure A.3. Executing the `PROCESS proc` gives:

```
Executing process PROC...

All 5 variables will be monitored during this simulation!

Building mathematical problem description took 0.009 seconds.

Loaded MA48 library
Execution begins....

Performing initialisation calculation at time:    0.000
   Variables
       Known            :      1
       Unknown          :      4
          Differential  :      2
          Algebraic     :      2

   Model equations      :      4
   Initial conditions   :      2


Checking consistency of model equations and ASSIGN specifications... OK!

Checking index of differential-algebraic equations (DAEs)...

   ERROR: Your problem is a DAE system of index greater than 1.
```

```
          Your differential variables ("states") are not independent: for
          example, you cannot specify arbitrary initial values for the
          differential variable(s):
              MYMOD.X1 (STATE)
          since the following 1 equation(s) will then be left with only 0
          unknown(s):
              Model Equation         3: MYMOD.X1^2 = MYMOD.Y2 ;

          The problem may have been caused because you ASSIGNed the
          following variable(s):
              MYMOD.Y2 (INPUT)
Initialisation calculation failed.

Execution of PROC fails prematurely.
```

The DAE system in this example consists of 4 equations in 4 unknown variables, and, as gPROMS confirms, is well-posed. However, with the variable `y2` `ASSIGN`ed, the system is high-index because the initial value of `x1` cannot be specified arbitrarily due to the algebraic relationship $x_1^2 = y_2$.

```
DECLARE
  TYPE
    NoType       = 0.5 :   -1E30 :  1E30
END #declare
#================================================================
MODEL mod1

VARIABLE
  x1, x2, y1, y2, y3                 AS NoType

EQUATION

  $x1 = x1*y1 ;

  $x2 = x1 + x2*y1 + y2 + 3*y3 ;

  x1^2 = y2 ;

  x2 = x1 + y1 + y2*y3 ;

END #model
#================================================================
PROCESS proc

   UNIT
      mymod    AS mod1

   ASSIGN
      WITHIN mymod DO
         y2 := 1 ;
      END #within

   INITIAL
      WITHIN mymod DO
         x1 = 0 ;
         x2 = 0 ;
      END #within

   SOLUTIONPARAMETERS
      ReportingInterval := 1 ;


   SCHEDULE
      CONTINUE FOR 10

END #process
```

Figure A.3: Illustrative example: high-index system

A.4 Inconsistent initial conditions

Once gPROMS has checked that the system is well-posed, square and of index 1, it checks the consistency of the initial conditions and identifies sub-systems that are over- or under-specified at $t = 0$. For example, consider the system shown in figure A.4. In this case, it is clear from inspection that the initial conditions, $x_1(0) = 0$ and $y_2(0) = 1$, are inconsistent due to the relationship $x_1^2 = y_2$. This is confirmed by the gPROMS output:

```
Executing process PROC...

All 5 variables will be monitored during this simulation!

Building mathematical problem description took 0.001 seconds.

Loaded MA48 library
Execution begins....

   Variables
       Known            :      1
       Unknown          :      4
          Differential  :      2
          Algebraic     :      2

   Model equations      :      4
   Initial conditions   :      2


Checking consistency of model equations and ASSIGN specifications... OK!

Checking index of differential-algebraic equations (DAEs)...         OK!

Checking consistency of initial conditions...

   ERROR: Your initial conditions are inconsistent.
          At time t=0, the following 3 equation(s) involve only 2 unknown
          variable(s).
              Model Equation         3: MYMOD.X1^2 = MYMOD.Y2 ;
              Initial Condition      1: MYMOD.X1 = 0 ;
              Initial Condition      2: MYMOD.Y2 = 1 ;

          The 2 unknown(s) occuring in these 3 equations are:

              MYMOD.Y2 (ALGEBRAIC)
              MYMOD.X1 (STATE)

Initialisation calculation failed.

Execution of PROC fails prematurely.
```

Note that using the initial conditions:

```
INITIAL
   WITHIN mymod DO
      $x1 = 0 ;
       y2 = 1 ;
   END #within
```

for example, rectifies the problem.

```
DECLARE
  TYPE
    NoType       = 0.5 :   -1E30 :  1E30
END #declare
#================================================================
MODEL mod1

VARIABLE
  x1, x2, y1, y2, y3                 AS NoType

EQUATION

  $x1 = x1*y1 ;

  $x2 = x1 + x2*y1 + y2 + 3*y3 ;

  x1^2 = y2 ;

  x2 = x1 + y1 + y2*y3 ;

END #model
#================================================================
PROCESS proc

   UNIT
      mymod    AS mod1

   ASSIGN
      WITHIN mymod DO
         y3 := 1 ;
      END #within

   INITIAL
      WITHIN mymod DO
         x1 = 0 ;
         y2 = 1 ;
      END #within

   SOLUTIONPARAMETERS
      ReportingInterval := 1 ;


   SCHEDULE
      CONTINUE FOR 10

END #process
```

Figure A.4: Illustrative example: system with inconsistent initial conditions

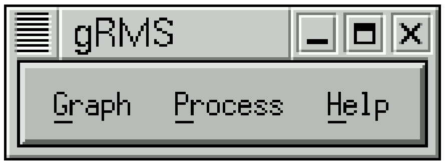
Appendix B

gRMS Output Channel

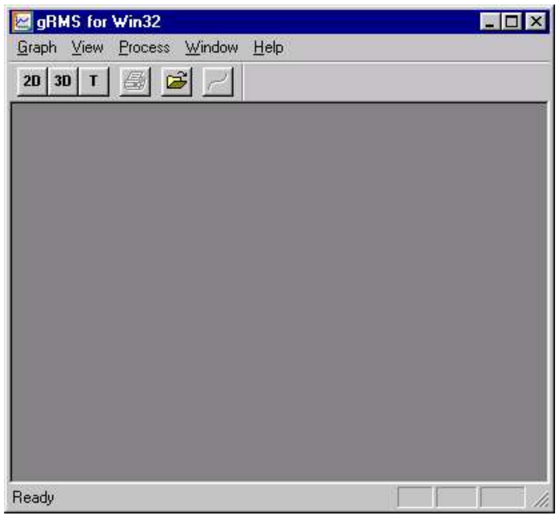
Contents

The gRMS (gPROMS Results Management Service) application provides facilities for plotting and printing gPROMS results as 2D and 3D graphs. It is normally started automatically by gPROMS.

You will find an introduction to the use of gRMS in sections 2.4.1 and 2.4.2 for MS Windows-based and Unix systems respectively. This appendix goes into some more detail that you may find useful in making the most out of this powerful results presentation tool.



The gRMS Toolbar (UNIX).



The empty gRMS frame (Windows).

UNIX The UNIX version of gRMS starts up displaying the gRMS Toolbar which contains menu-items for loading/saving data and for creating new plots. Each plot is displayed in a separate window with its own menu-bar containing items specific to that plot.

Windows The Windows version of gRMS starts up displaying an empty frame. Each plot is displayed in a sub-window within that frame. All menu-items appear on the main frame's menu and plot specific menu-items always operate on the currently active plot. If there is no active plot, then plot specific menu-items cannot be used and will be grayed out.

B.1 gRMS processes

gRMS organises all its results in Processes. A gRMS Process:

- is created when a gPROMS `PROCESS` starts being `execute`d (*cf.* section 2.3);
- receives data from gPROMS including information on the variables in the problem as well as their values during the simulation;
- remains in existence even after the gPROMS simulation has terminated or, indeed, gPROMS has been exited;
- can be saved as a permanent file on disk; such files normally have a `.gRMS` file extension;
- may be reloaded by gRMS from the above file at a later time in order to display results *etc.*. This can be done using the "Open..." menu-item.

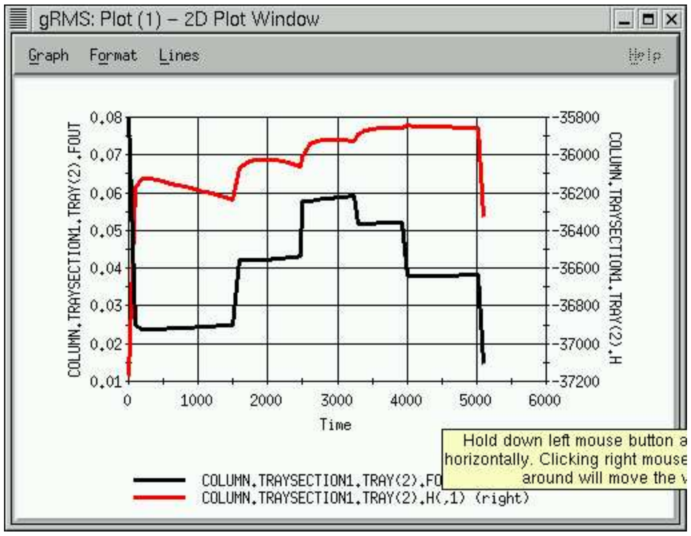
gRMS manages its processes using the "Process" menu. Initially this menu contains the "Open..." "Save All" and "Close All" items. Each new Process that is created (either by an executing gPROMS `PROCESS` or by loading in a `.gRMS` file) appears as an additional item on this menu.

The menu-item for each Process in the "Process" menu is a pull-right menu containing the following items:

Close	Close the Process. If the Process has not been saved, then the data that it contains will be lost.
Save	Save the process using its current name. This menu-item is disabled if the process has already been saved.
Save As...	Display a standard file dialog allowing you to choose a destination directory and filename to which the Process will be saved.
Properties...	Displays statistics about the Process including the number of variables, domains and time-intervals.

B.2 Plotting 2D graphs

To plot a new 2D graph, select the “Graph→New 2D Plot” menu-item. A new “2D Plot Window” is displayed containing an empty 2D plot.


gRMS: Plot (1) – 2D Plot Window
Graph
Format
Lines
Help
COLUMN.TRAYSECTION1.TRAY(2).FOUT
COLUMN.TRAYSECTION1.TRAY(2).H
Time
Hold down left mouse button a
horizontally. Clicking right mouse
around will move the v
COLUMN.TRAYSECTION1.TRAY(2).FO
COLUMN.TRAYSECTION1.TRAY(2).H(,1) (right)

A 2D Plot Window (UNIX).

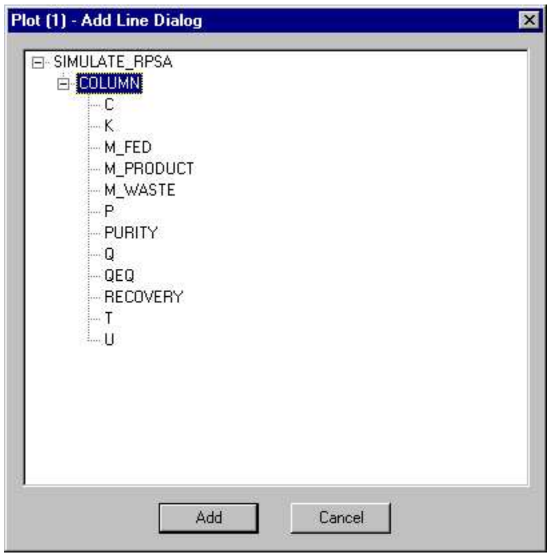
B.2.1 Adding lines to a plot

To add a new line to a plot select the “Line→Add...” menu-item. The “Add Line Dialog” is displayed which allows you to navigate the model hierarchy and choose a variable to be plotted.

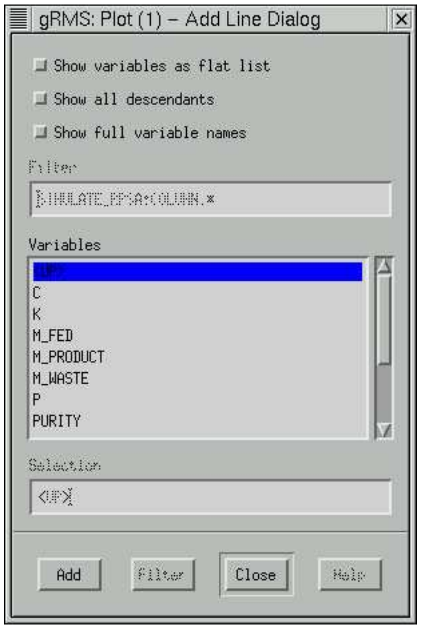
When a line is added to a plot, a “Line Properties Dialog” will be displayed so that the line can be formatted (see section B.2.2). The line will be drawn on the plot only if it has a single free domain *i.e.* the corresponding variable is a function of a single independent variable (usually Time).

Windows On Windows, navigation of the model hierarchy is achieved using a tree-style mechanism. To add a line corresponding to a certain variable:

- *either* double-click on the variable,
- *or* click on the variable to select it and then click “OK”.



The Add Line Dialog (Windows).



The Add Line Dialog (UNIX).

UNIX On UNIX, navigation of the model hierarchy is achieved using a list.

Selecting (single-clicking) an item in the list causes it to be listed in the “Selection” field.

The action of the “OK” button depends on what type of item is in the “Selection” field:

`<UP>`	Moves up the hierarchy by one level.
`xxx*`	Moves down the hierarchy into model instance `xxx`.
`xxx`	Adds a line to the plot using the variable `xxx`.

The behaviour of this dialog can be altered by the three toggle buttons which, by default, are all set off.

Show variables as flat list – setting this toggle flattens the model hierarchy and activates the “Filter” field which can then be used to filter the contents of the list using standard UNIX wild-cards (“?” to represent one character, “*” to represent any number of characters.)

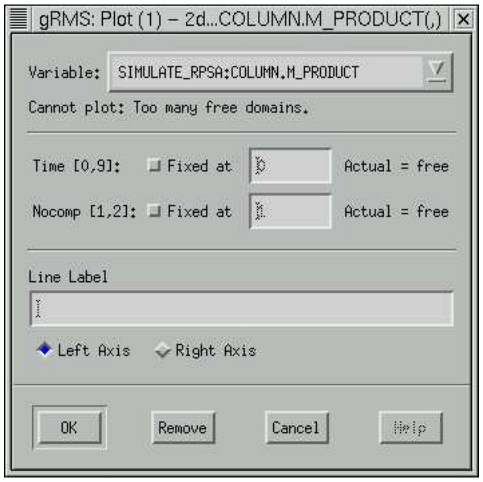
Show all descendants – by default, the list only displays the models/variables at the current level of the hierarchy. With this toggle set, the list also includes the names of all models/variables deeper in the hierarchy.

Show full variable names – by default, the list only displays the next element in the name of each model/variable. Setting this toggle causes the full names to be listed.

B.2.2 Formatting lines

For each line on the plot, the “Line” menu contains a pull-right menu-item (if the line cannot be plotted, then on Windows its name is preceded with “*” and on UNIX it is highlighted in red.) The pull-right menu contains the following items:

Properties...	Displays a "Line Properties Dialog" for the line. This dialog presents you with a list of all the domains of the line (3 in the example shown, "Time", "Axial" and "Radial".) In order to be plottable a line must have only one free-domain which is achieved by fixing the other domains to a point. In addition, this dialog allows you to specify a label for the line that will appear in the plot legend, and to specify which of the y-axis the line should be plotted against. *N.B.* If a plot contains only one line it will be plotted against the Left axis even if you specify otherwise. A drop-down list on this dialog allows you to change the data-source to any similar variable or to convert the line into a "Template" (both of these concepts are explained in section B.7).
Style...	Pops up a "Line Style Dialog" for the line to allow you to change its appearance on the plot.
Copy	Adds an identical copy of the line to the plot.
Remove	Removes the line from the plot.


gRMS: Plot (1) – 2d...COLUMN.M_PRODUCT(,)

Variable: SIMULATE_RPSA:COLUMN.M_PRODUCT

Cannot plot: Too many free domains.

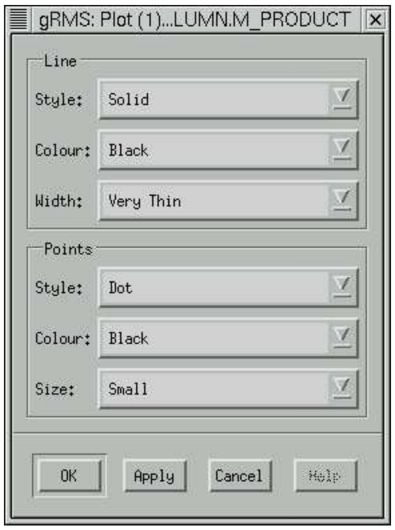
Time [0,9]: Fixed at 0 Actual = free

Nocomp [1,2]: Fixed at 1 Actual = free

Line Label

[✓ Left Axis] Right Axis

Line Properties Dialog (UNIX).


gRMS: Plot (1)...LUMN.M_PRODUCT

Line

Style: Solid

Colour: Black

Width: Very Thin

Points

Style: Dot

Colour: Black

Size: Small

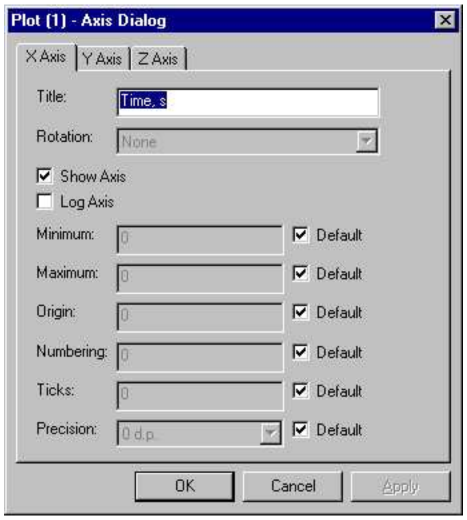
Line Style Dialog (UNIX).

B.2.3 Formatting 2D plots

The format of the plot is controlled via the items in the plot's "Format" menu.

B.2.3.1 Axes

Windows The format of the axes can be changed using the "Axis Format Dialog" which is displayed when the "Format→Axis..." menu-item is selected.



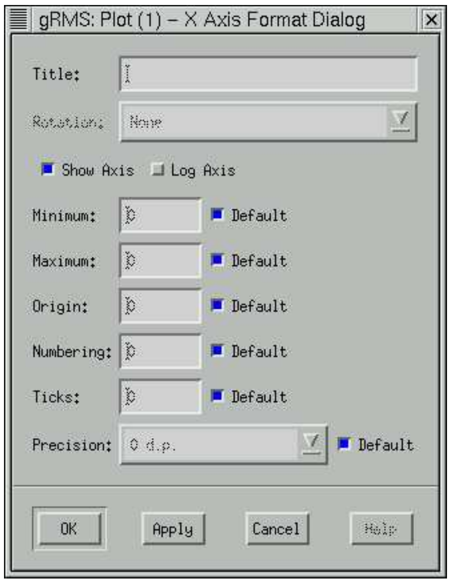
Title: Time, s
Rotation: None
[✓] Show Axis
[ ] Log Axis
Minimum: 0 [✓] Default
Maximum: 0 [✓] Default
Origin: 0 [✓] Default
Numbering: 0 [✓] Default
Ticks: 0 [✓] Default
Precision: 0 d.p. [✓] Default

Axis Format Dialog (Windows).

Title	Title to display on the axis. By default, this is the name of the first line plotted against that axis.
Rotation	Orientation of the axis title. This cannot be changed for the X-Axis.
Show Axis	Select to display the axis.
Log Axis	Select to display the axis with a logarithmic scale.
Minimum	Minimum value of the data on this axis.
Maximum	Maximum value of the data on this axis.
Origin	Specifies where the axis should be drawn. For example, setting the x origin to 4.0 will cause the left y-axis to cross the x-axis at 4.0.
Numbering	Increment between axis numbers.
Ticks	Increment between axis ticks.
Precision	Number of decimal places the axis numbering should use.

A bounding box can be displayed around the axes by selecting the "Format→Bounding Box" menu-item.

UNIX On UNIX, the "Axis Format Dialog" can be used only to alter the format of one axis at a time and "Format→Axis" menu-item is a pull-right menu containing an item for each axis.



Title:

Rotation: None

[✓ Show Axis] Log Axis

Minimum: 0 [✓ Default]

Maximum: 0 [✓ Default]

Origin: 0 [✓ Default]

Numbering: 0 [✓ Default]

Ticks: 0 [✓ Default]

Precision: 0 d.p. [✓ Default]

Axis Format Dialog (UNIX).

The bounding box is displayed by selecting the "Format→Axis→Bounding Box" menu-item.

B.2.3.2 Default line styles

When lines are first created on 2D plots, gRMS chooses an unused line style from the following list of default styles:

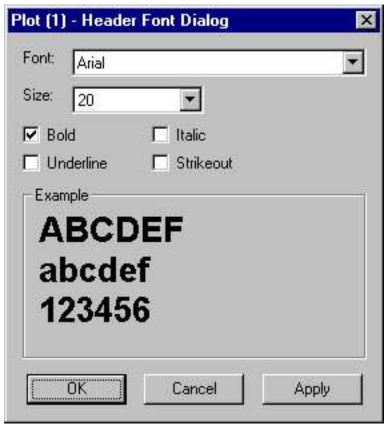
Name	Line Style	Colour	Data Points Style
Black	Solid	Black	Dot
Red	Solid	Red	Square
Blue	Solid	Blue	Triangle
Green	Solid	Green	Diamond
Magenta	Solid	Magenta	Star
Dashed Black	Dashed	Black	Dot
Dashed Red	Dashed	Red	Square
Dashed Blue	Dashed	Blue	Triangle
Dashed Green	Dashed	Green	Diamond
Dashed Magenta	Dashed	Magenta	Star

The list of default styles can be edited using the "Format→Default Line Style" menu-item.

As changes to the default style list are lost when gRMS is shut down, this facility is only really useful when used in conjunction with Plot Templates (see section B.7).

B.2.3.3 Fonts

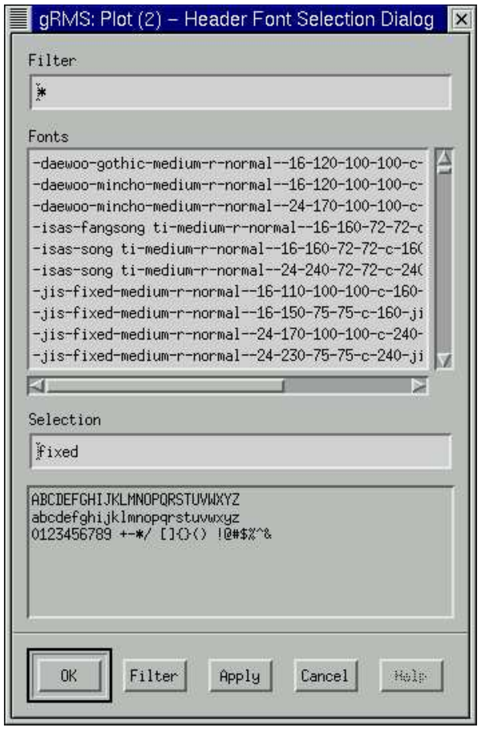
The fonts used to display text on the plot can be changed using the items in the "'Format→Fonts" menu. These can be used to display a "Font Selection Dialog" which allows a font to be picked from a system dependent list.



Font: Arial
Size: 20
[✓] Bold
[ ] Italic
[ ] Underline
[ ] Strikeout

ABCDEF
abcdef
123456

Font Selection Dialog (Windows).



Filter: *

Fonts:

```
-daewoo-gothic-medium-r-normal--16-120-100-100-c-
-daewoo-mincho-medium-r-normal--16-120-100-100-c-
-daewoo-mincho-medium-r-normal--24-170-100-100-c-
-isas-fangsong ti-medium-r-normal--16-160-72-72-c
-isas-song ti-medium-r-normal--16-160-72-72-c-16(
-isas-song ti-medium-r-normal--24-240-72-72-c-24(
-jis-fixed-medium-r-normal--16-110-100-100-c-160-
-jis-fixed-medium-r-normal--16-150-75-75-c-160-ji
-jis-fixed-medium-r-normal--24-170-100-100-c-240-
-jis-fixed-medium-r-normal--24-230-75-75-c-240-ji
```

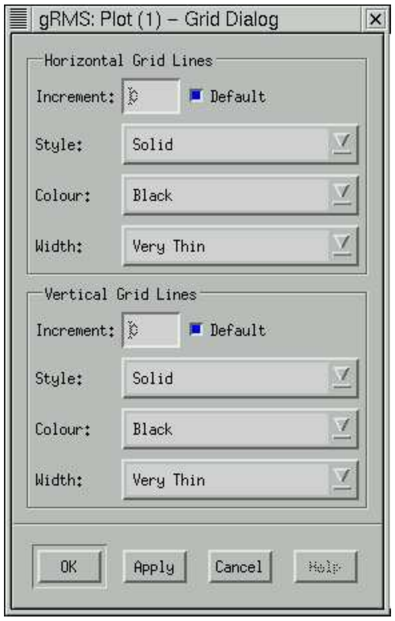
Selection: fixed

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
0123456789 +-*/ []{}() !@#$%^&
```

Font Selection Dialog (UNIX).

B.2.3.4 Grid

The format of the plot's grid can be changed using the "Grid Dialog" which is displayed when the "Format→Grid..." menu-item is selected.



Horizontal Grid Lines
Increment: 0 [✓ Default]
Style: Solid
Colour: Black
Width: Very Thin

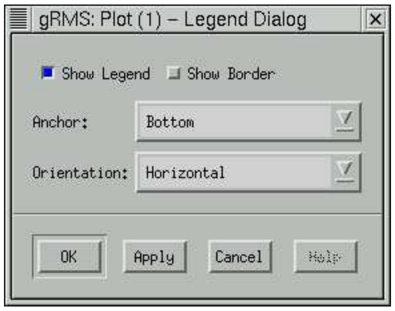
Vertical Grid Lines
Increment: 0 [✓ Default]
Style: Solid
Colour: Black
Width: Very Thin

Grid Dialog (UNIX).

By default, the "Increment" of a grid-line is the same as the axis-numbering. To remove the grid-lines set the "Increment" to 0.

B.2.3.5 Legend

The format of the plot's legend can be changed using the "Legend Dialog" which is displayed when the "Format→Legend..." menu-item is selected.

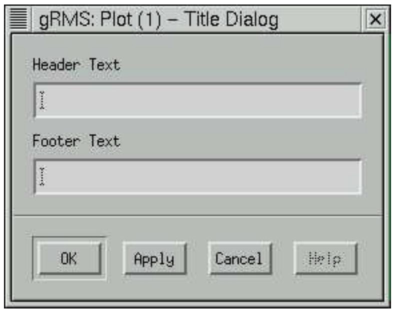


Legend Dialog (UNIX).

N.B. The values of the "Anchor" and "Orientation" are only hints, if the window is too small then the legend may not appear as specified.

B.2.3.6 Title

The plot can be supplied with a header and footer using the "Title Dialog" which is displayed when the "Format→Title..." menu-item is selected.



Title Dialog (UNIX).

B.2.3.7 Scaling, zooming and translation

Scaling	With `Ctrl` pressed and the middle mouse button depressed, moving the mouse up and down zooms the plot in and out.
Translation	With `Shift` pressed and the middle mouse button depressed, moving the mouse translates the plot.
Zooming	With `Shift` pressed and the left mouse button depressed the mouse can be used to select an area to zoom into.

Pressing `r` resets the scaling, translation and zooming.

N.B. For mice with only two buttons pressing the middle button is simulated by pressing both buttons simultaneously.

B.3 Plotting 3D graphs

To plot a new 3D graph select the "Graph→New 3D Plot" menu-item. A new "3D Plot Window" is displayed containing an empty 3D Plot. The appearance of this window is the same as for a 2D Plot except the "Line" menu is replaced by the "Surface" menu.

B.3.1 Adding a surface to a plot

Adding a surface to a 3D Plot is achieved in the same way as adding a line to a 2D Plot.

Only one 3D surface can be plotted at a time. If the plot already contains a surface, then the "Surface→Add..." menu-item is renamed "Surface→Change..."; otherwise, the behaviour is the same.

B.3.2 Formatting surfaces

The "Surface" menu contains the following items for formatting the surface displayed on a 3D Plot.

Properties...	Displays a "Surface Properties Dialog" for the surface. This dialog is functionally identical to the "Line Properties Dialog".
Remove	Removes the surface from the plot.
Draw Mesh	When set, the surface is plotted in 3D with the X-Y grid projected onto the surface.
Draw Shade	When set, the surface is plotted in 3D with flat shading.
Draw Contour	When set, contour lines are automatically drawn between distribution levels in the data.
Draw Zones	When set, each distribution level in the data is displayed in a different solid colour.

B.3.3 Formatting 3D plots

The format of the plot is controlled via the items in the plot's "Format" menu.

B.3.3.1 Axes

Windows Same as for 2D plots though there are fewer controllable parameters.

UNIX The format of the axes can be changed using the "3D Axis Format Dialog" which is displayed when the "Format→Axis..." menu-item is selected.

Functionally, this is the same as for 2D plots except there are fewer controllable parameters.

B.3.3.2 Fonts

Windows Same as for 2D plots except that only the style (Bold, Italic, ...) and not the face can be selected for the Axis font.

UNIX Same as for 2D plots except the Axis font is picked from a pull-right menu rather than the “Font Selection Dialog”.

B.3.3.3 Legend

Same as for 2D plots except that there is an option to display the legend as either “Stepped” or “Continuous”.

B.3.3.4 Rotation

The default rotation of the plot about the (X,Y,Z) axes is (45,0,45).

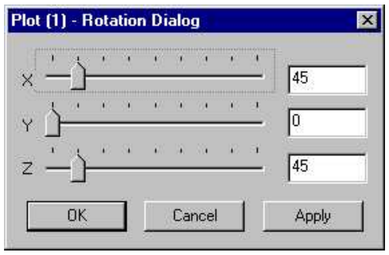
With the middle mouse button (or both buttons for 2-button mouse) depressed, moving the mouse rotates the plot.

If you hold down `x`, `y`, or `z`, then the rotation is restricted to being around that axis.

If your hold down `e`, then the rotation is restricted to being perpendicular to the screen.

Alternatively you can change the rotation from the “Format” menu.

Windows The rotation of the plot can be changed using the “Rotation Dialog” which is displayed when the “Format→Rotation...” menu-item is selected.


Plot (1) - Rotation Dialog

X: 45
Y: 0
Z: 45

Rotation Dialog (Windows).

UNIX The rotation of the plot can be changed to a limited set of preset values using the sub-items in the “Format→Rotation” pull-right menu.

B.3.3.5 Title

Same as for 2D plots.

B.3.3.6 Scaling, zooming and translation

Same as for 2D plots except you need to hold down `Ctrl` and not `Shift` when zooming.

N.B. You can actually hold down `Ctrl` when zooming 2D plots, but in this case the axes will only be displayed if they lie within the selected zoom area.

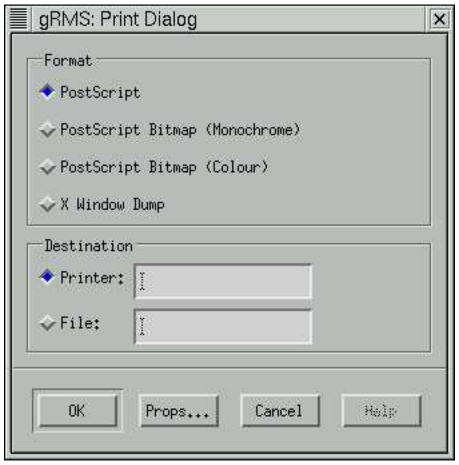
B.4 Printing gRMS plots

2D and 3D Plots can be printed by selecting the “File→Print...” menu-item.

Windows gRMS uses the standard Windows print dialog.

The printed plot will be scaled to fit the page whilst maintaining the same aspect ratio as displayed on the screen.

UNIX The “Print Dialog” will be displayed allowing you to choose the format of the printed plot, and the name of the printer or file you wish to print to.

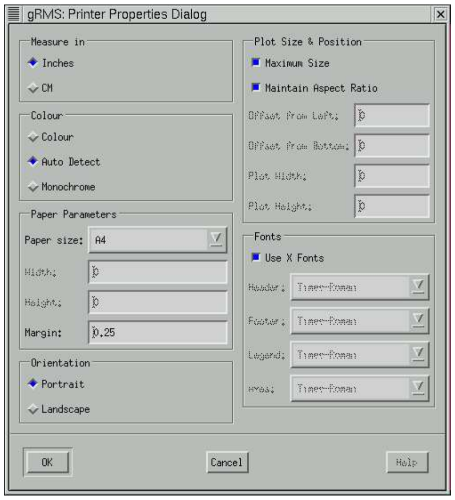


gRMS: Print Dialog

Format
- [✓ PostScript]
- PostScript Bitmap (Monochrome)
- PostScript Bitmap (Colour)
- X Window Dump

Destination
- [✓ Printer:]
- File:

The Print Dialog (UNIX).



gRMS: Printer Properties Dialog

Measure in
- [✓ Inches]
- CM

Colour
- Colour
- [✓ Auto Detect]
- Monochrome

Paper Parameters
- Paper size: A4
- Width: 0
- Height: 0
- Margin: 0.25

Orientation
- [✓ Portrait]
- Landscape

Plot Size & Position
- [✓ Maximum Size]
- [✓ Maintain Aspect Ratio]
- Offset From Left: 0
- Offset From Bottom: 0
- Plot Width: 0
- Plot Height: 0

Fonts
- [✓ Use X Fonts]
- Header: Times-Roman
- Footer: Times-Roman
- Legend: Times-Roman
- Axes: Times-Roman

The Printer Properties Dialog (UNIX).

PostScript The default format which outputs an encapsulated PostScript (EPSF-2.0) image of the graph using device-independent PostScript operators.

Clicking the “Props...” button pops-up the “Printer Properties Dialog” which contains additional formatting options for “PostScript” output.

The “Fonts” option requires a little explaining, by default gRMS tries to “Use X Fonts” which means that the plot is printed using the closest available font to that displayed on the screen. If this does not print correctly then you can disable this option and select fonts from the four menus.

N.B. To obtain the best WYSIWYG (what you see is what you get) output filling the whole of the printed page, stretch the Plot Window so that it has the aspect ratio of your paper in the orientation you are using and then set the "Maintain Aspect Ratio" toggle button to be off.

Three additional output formats are also available, however the options in the "Printer Properties Dialog" are not available for them:

PostScript Bitmap (Monochrome) An encapsulated PostScript (EPSF-2.0) image of the graph created by taking the pixels on the screen and outputting them using the PostScript image operator. The resolution is not as good as with the standard PostScript format, and the file size is much larger. The only reason to use this format is if the plot uses especially unusual fonts which are not reproduced correctly by the standard PostScript format.

PostScript Bitmap (Colour) Same as the above, but in Colour.

X Window Dump A standard X Windows Dump representation of the graph.

B.5 Viewing and exporting data

Data from 2D and 3D plots can be viewed in a window or exported as a “tab/space/comma delimited” ASCII text file suitable for importing into a spreadsheet.

N.B. “comma delimited” format is only available in the Windows version of gRMS.

B.5.1 2D plots

The data can be viewed by selecting the “Graph→View Data...” menu-item.

The data is displayed in a table with the values of the free-domain in the first column and the values of lines plotted against that domain in subsequent columns. If the plot contains lines from variables in different Processes, or lines plotted against different free-domains then multiple tables are displayed.

Windows The data can be exported to a file by selecting
Graph→Export Data...”. This displays a Windows file dialog for you to specify the file name and type.

UNIX The data can be exported to a file by selecting either the “Graph→Export Data→Tab Delimited Table...” or “Graph→Export Data→Space Delimited Table...” menu-item. This displays a standard Motif file dialog for you to specify a file name.

B.5.2 3D plots

The data can be viewed by selecting either the “Graph→View Data→Table...” or “Graph→View Data→Matrix...” menu-item.

In ‘Matrix’ format the data is exported in a table with the x-values labelling the columns, the y-values labelling the rows and the z-values in the table. The ‘Table’ format exports the data in a three column table (x,y,z).

Windows The data can be exported to a file by selecting “Graph→Export Data...”. This displays a Windows file dialog for you to specify the file name and type.

UNIX The data can be exported to a file by selecting one of the “Graph→Export Data→Tab Delimited Table...”, “Graph→Export Data→Space Delimited Table...”,
“Graph→Export Data→Tab Delimited Matrix...” or “Graph→Export Data→Space Delimited Matrix...” menu-items. This displays a standard Motif file dialog for you to specify a file name.

B.6 Exporting images

Graphical images of plots can be exported from gRMS for inclusion in documents and presentations.

Windows Select the "Graph→Export Image..." menu-item. This displays a standard Windows file dialog for you to specify a file name and image type from the following:

Enhanced Metafile (emf)
Aldus Placeable Windows Metafile (wmf)
Windows Bitmap (bmp)
Standard PNG (png)
Interlaced PNG (png)
JPEG (jpg)

UNIX Select the "Graph→Print..." menu-item and use the "Printer Dialog" to select output to a file.

B.7 Templates

To simplify the use of gRMS when creating many similar plots, *e.g.* for multiple runs of the same process, gRMS allows plot and line 'Templates' to be defined.

A template is a 'description' of a plot (or line) that contains everything needed to display the plot except for the data itself.

Using a template requires sources of data for the plot (or line) to be specified; this is known as instantiation.

There are two types of Templates, 'Line/Surface Templates' and the much more useful 'Plot Templates'.

All references to lines on 2D plots in the following discussion also apply to a surface on a 3D plot.

B.7.1 Line templates

Line templates are 'descriptions' of lines; they contain everything needed to display a line except for the data itself. Line templates contain information about the line colour, line style, line width, point style (what symbol is used), point colour, point size and the line label.

Like lines, line templates appear on the "Lines" menu of the Plot Window. They can be manipulated exactly like lines (see section B.2.2). They can be instantiated from their Line Properties Dialog, by selecting a variable from the "Variable" drop-down list (this contains a list of all variables with the same name, distributed over domains of the same name and in the same order).

Lines can be converted into templates from their Line Properties Dialog by selecting the "Template: *.xxx" item from the "Variable" drop-down list.

B.7.2 Plot templates

Plot Templates are 'descriptions' of plots; they contain everything needed to display a plot except for the data itself. Plot templates contain information about each axis (scale, origin, minimum value, maximum value, tick parameters, title, *etc.*), all fonts used, grid lines, the legend, the title, the properties of each line (as in a line template) and the set of variables to be plotted. One can imagine that setting all of this information each time a process is executed will become extremely tedious. Also note that a plot template can be instantiated with processes that are not identical to the one used to create the template: gRMS will search through the data to find as many matches to the variables that it has in the template.

Plot templates are created from normal 2D and 3D plots (see sections B.7.2.1–B.7.2.3).

N.B. It is possible to create a template from a plot that contains no lines or surfaces; just line templates and axes, font, grid, legend, title and default line style formatting. Such a template requires no instantiation.

B.7.2.1 Saving a plot as a template

To save a plot as a template select the "Graph→Save As Template..." menu-item. This displays a standard file dialog allowing you to choose a name for the template file. We suggest that plot templates are saved with the file extension `.gpt`.

B.7.2.2 Creating a new plot from a saved template

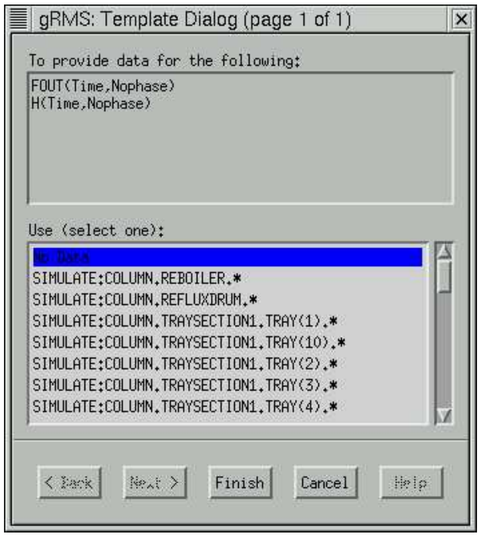
To create a new plot from a previously saved template select the "Graph→Open Template..." menu-item. This displays a standard file dialog allowing you to choose the template to open.

B.7.2.3 Using an existing plot as a template

If you want to quickly use a plot as a template without going through the Save/Open procedure then select the "Graph→Use As Template..." menu-item.

B.7.2.4 Instantiating a plot template

The 'Plot Template Dialog' is displayed when you use any plot template that requires instantiation (*i.e.* was created from a plot containing lines or surfaces). This dialog is used to instantiate the template.


gRMS: Template Dialog (page 1 of 1)

To provide data for the following:

```
FOUT(Time,Nophase)
H(Time,Nophase)
```

Use (select one):

```
No Data
SIMULATE:COLUMN.REBOILER.*
SIMULATE:COLUMN.REFLUXDRUM.*
SIMULATE:COLUMN.TRAYSECTION1.TRAY(1).*
SIMULATE:COLUMN.TRAYSECTION1.TRAY(10).*
SIMULATE:COLUMN.TRAYSECTION1.TRAY(2).*
SIMULATE:COLUMN.TRAYSECTION1.TRAY(3).*
SIMULATE:COLUMN.TRAYSECTION1.TRAY(4).*
```

Template Dialog (UNIX).

The dialog displays the details for one required data-source at a time. If more than one data-source needs to be instantiated then the "Back" and "Next" buttons can be used to move through the required data-sources.

The dialog contains two lists. The top list contains all the variables that are required from the data-source whilst the bottom list contains all the possible data-sources meeting these requirements (from those processes loaded into gRMS).

A particular data-source is chosen by selecting it from the bottom list.

When you have specified all the data-sources click "Finish".

By default all the data-sources are instantiated as "No Data" which means that lines depending on those data-sources will become 'Line Templates' on the finished plot.

B.7.2.5 Common templates

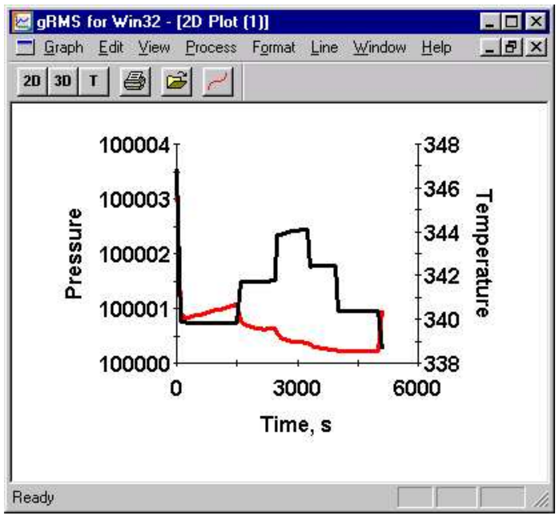
Plot templates saved in the `oc` directory of the gPROMS installation directory (as identified by the value of the `GPROMSHOME` environment variable) are known as common templates.

These appear in the "Graph→Open Common Template" menu for easy access.

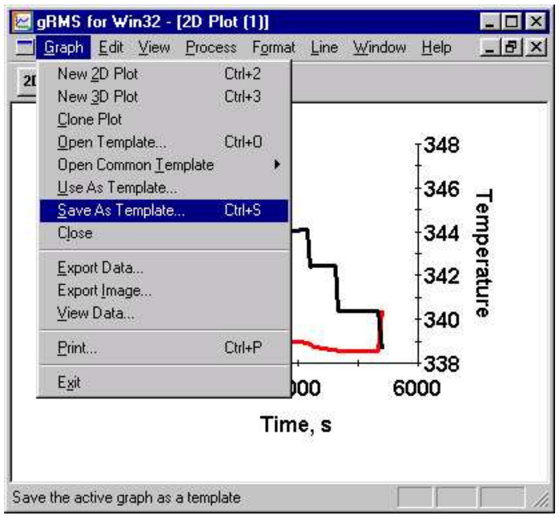
If you have common templates called `plot2d.gpt` and `plot3d.gpt` then gRMS uses these when you select the "Graph→New 2D Plot" and "Graph→New 3D Plot" menu-items. Usually you would create these templates from plots with no lines or surfaces.

B.7.2.6 Plot template example

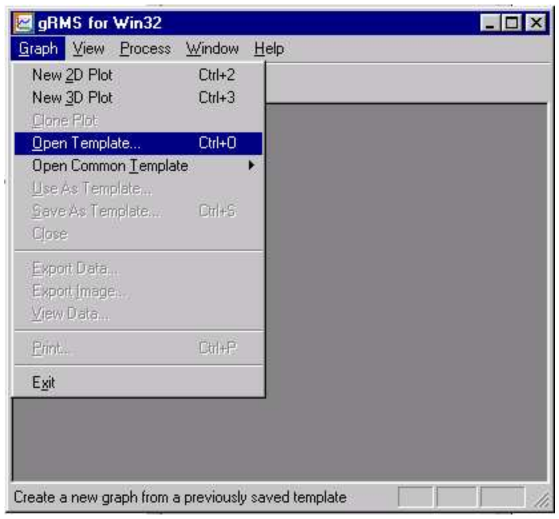
This example demonstrates how plot templates can be used to simplify viewing of the results for similar sub-models within the same gPROMS process. The example shown creates and used a template for models with P (pressure) and T (temperature) variables.



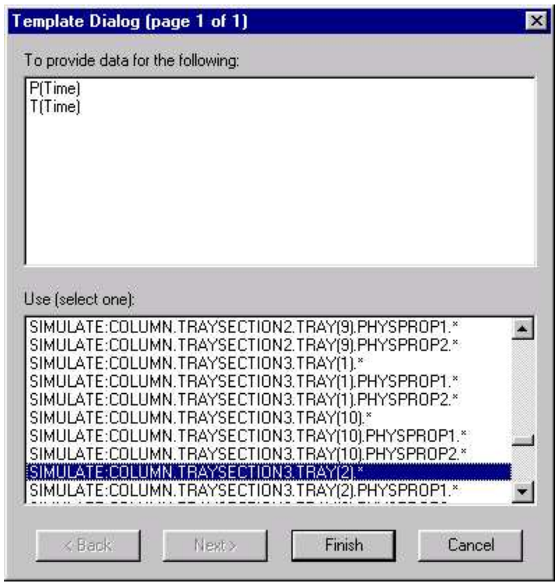
1. Create a plot of P and T from one model.



2. Save the plot as a template.



3. Open the template.


Template Dialog (page 1 of 1)

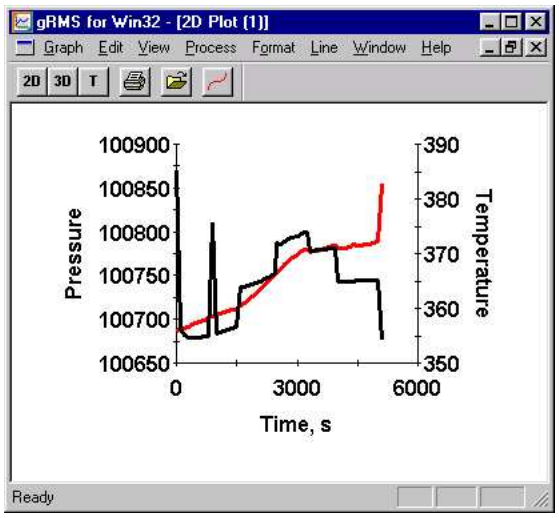
To provide data for the following:

```
P(Time)
T(Time)
```

Use (select one):

```
SIMULATE:COLUMN.TRAYSECTION2.TRAY(9).PHYSPROP1.*
SIMULATE:COLUMN.TRAYSECTION2.TRAY(9).PHYSPROP2.*
SIMULATE:COLUMN.TRAYSECTION3.TRAY(1).*
SIMULATE:COLUMN.TRAYSECTION3.TRAY(1).PHYSPROP1.*
SIMULATE:COLUMN.TRAYSECTION3.TRAY(1).PHYSPROP2.*
SIMULATE:COLUMN.TRAYSECTION3.TRAY(10).*
SIMULATE:COLUMN.TRAYSECTION3.TRAY(10).PHYSPROP1.*
SIMULATE:COLUMN.TRAYSECTION3.TRAY(10).PHYSPROP2.*
SIMULATE:COLUMN.TRAYSECTION3.TRAY(2).*
SIMULATE:COLUMN.TRAYSECTION3.TRAY(2).PHYSPROP1.*
```

4. Instantiate the template from a different model with P and T.



5. View the resulting plot.

B.8 Advanced use of gRMS

In all likelihood you will never need to use any of the features described in this section, but for the adventurous herein may lie some items of interest.

B.8.1 Preventing gRMS from starting automatically with gPROMS

To prevent gPROMS automatically starting gRMS set the `NOGRMS` environment variable.

N.B. gRMS will also not be started automatically if the `GRMSPORT` or `GRMSHOST` environment variables are set, or if the UNIX version of gPROMS is started with the `-port` or `-host` flags.

B.8.2 Starting gRMS independently from gPROMS

If you wish to start gRMS independently of gPROMS then you can start it with the command line:

`gRMS.exe [-port` *number*`] [-dir` *directory*`] [-print` *printer*`] [-lpr]`

Where the contents of [square brackets] are optional command line switches for the following:

`-port`	gRMS is a server application that receives data over a network from gPROMS. This command line switch tells gRMS to listen for connections from gPROMS on a given port number. When gRMS is started by gPROMS this is set automatically. If not specified gRMS checks to see if `GRMSPORT` is set and if not uses port 9876.
`-dir`	This is the directory which gRMS will try to open and save files to by default. When gRMS is started by gPROMS this is set automatically. If not specified gRMS check to see if `GRMSDIR` is set. If `GRMSDIR` is not set then it uses `GPROMSDIR/output` and if that is not set it uses the directory it is started in.
`-print`	This switch is only for the UNIX version and can be used to specify the name of the default printer that gRMS should print plots to. If not specified gRMS checks to see if `GRMSPRINTER` is set.
`-lpr`	This switch is only for the UNIX version. By default gRMS uses the UNIX system program “lp” to print plots, some earlier versions of UNIX do not have this program and use one called “lpr” instead, this switch tells gRMS to use the “lpr” program. If not specified gRMS checks to see if `GRMSLPR` is set..

B.8.3 Running gPROMS and gRMS on different machines

Because gPROMS and gRMS communicate using the TCP/IP protocol they can be run on separate machines and communicate over a local area network, or the internet. This is best demonstrated by example: if we want to run gRMS on a machine called marzipan.psenterprise.com (the name of the machine gPROMS is running on is not important) and communicate using port 9999 then gRMS is started like this:

```
gRMS.exe -port 9999 -dir ~/gPROMS/output
```

and gPROMS is started like this:

```
gPROMS -port 9999 -host marzipan.psenterprise.com
```

N.B. The Windows version of gPROMS does not currently accept command line arguments so you would have to set `GRMSPORT` and `GRMSHOST` instead.

B.8.4 Multiple gPROMS runs communicating with a single gRMS

More than one gPROMS run can communicate their results to a single instance of gRMS at a time.

Again we demonstrate by example, and use port 9999 for communication. gRMS is started like this:

```
gRMS.exe -port 9999 -dir ~/gPROMS/output
```

the first gPROMS is started like this:

```
gPROMS -port 9999
```

and the second gPROMS is started in the same way:

```
gPROMS -port 9999
```

Both gPROMS runs will now communicate their results to gRMS.

N.B. The Windows version of gPROMS does not currently accept command line arguments so you would have to set `GRMSPORT` instead.

B.8.5 gRMS resources under UNIX

X-Windows provides a mechanism to customise applications using Resource Files. It is beyond the scope of this manual to discuss the eccentricities of this mechanism other than to refer the user to a book (X-Window System User's Guide, OSF/Motif 1.2 Edition, O'Reilly & Associates, Inc., ISBN 1-56592-015-5) and to list the resources that can be used to customise gRMS. If you would like help in creating a gRMS resource file on your system then please contact `support.gPROMS@psenterprise.com`. The resources provided for customisation can be split into three sets:

- Those mimicking the command line switches. If gRMS is started from the command line using switches then these override the associated resource settings.
- Those that control the overall appearance of gRMS.
- Those that control the appearance of individual gRMS windows and dialogs.

Resource	Function
Grms.directory	Duplicates function of `-dir` command line switch.
Grms.lpr	Duplicates function of `-lpr` command line switch.
Grms.port	Duplicates function of `-port` command line switch.
Grms.printer	Duplicates function of `-print` command line switch.

Table B.1: Resources mimicking the command line switches.

Resource	Function	Default
height	The height of the dialog or window.	Not set, except for Grms.PlotWindow.height that is set to 600.
width	The width of the dialog or window.	Not set, except for Grms.PlotWindow.width that is set to 500.
foreground	The foreground colour of the dialog or window.	Not set, so defaults to value of Grms*.foreground.
background	The background colour of the dialog or window.	Not set, so defaults to value of Grms*.background.
x	The initial x position of the top left corner of the dialog or window.	Not set, so position depends on Window Manager.
y	The initial y position of the top left corner of the dialog or window.	Not set, so position depends on Window Manager.

Table B.2: Resources controlling individual windows and dialogs.

Grms	Grms*.PlotWindow
Grms*.AddDialog	Grms*.PrintDialog
Grms*.AxisDialog	Grms*.PrintPropsDialog
Grms*.ErrorDialog	Grms*.PropsDialog
Grms*.FontDialog	Grms*.QuestionDialog
Grms*.GridDialog	Grms*.StyleDialog
Grms*.InformationDialog	Grms*.TitleDialog
Grms*.LegendDialog	

Table B.3: Names of individual windows and dialogs.

Resource	Function	Default
Grms*.background	Default background colour for gRMS windows.	grey
Grms*.foreground	Default foreground colour for gRMS windows.	black
Grms*.fontList	Comma separated list of three fonts, the standard font, the bold font and the italic font. N.B. Due to bugs in the DEC/OSF1 version of Motif, the bold and italic fonts do not have any effect.	fixed, fixed=BOLD_TAG, fixed=ITALIC_TAG
Grms*.selectColor	Colour for selected toggle buttons.	blue
Grms*.DirList.background	Background colour for list in File Dialog, should be set the same as Grms*.Text.background.	light grey
Grms*.DirList.foreground	Foreground colour for list in File Dialog, should be set the same as Grms*.Text.foreground.	black
Grms*.FilterText.background	Background colour for "Filter" text fields, should be set the same as Grms*.Text.background.	light grey
Grms*.FilterText.foreground	Foreground colour for "Filter" text fields, should be set the same as Grms*.Text.foreground.	black
Grms*.ItemsList.background	Background colour for list in Font Dialog and Add Dialog, should be set the same as Grms*.Text.background.	light grey
Grms*.ItemsList.foreground	Foreground colour for list in Font Dialog and Add Dialog, should be set the same as Grms*.Text.foreground.	black
Grms*.Text.background	Background colour for text fields.	light grey
Grms*.Text.foreground	Foreground colour for text fields.	black

Table B.4: Resources controlling general appearance.

Appendix C

Microsoft Excel Output Channel

Contents

C.1 Introduction

The Microsoft Excel[1] output channel is a method for storing the results of a simulation in spreadsheet form. Each time a gPROMS simulation is executed, the output channel creates a new Microsoft Excel workbook containing the values of the variables arranged in worksheets. These data can then be plotted or manipulated using Excel's existing facilities or exported to other common data management systems.

[1]The facilities described in this Appendix are supported by Microsoft Excel 97 and later versions.

C.2 Enabling the Microsoft Excel Output Channel

The Microsoft Excel output channel is enabled via a specification in the `SOLUTIONPARAMETERS` section of the `PROCESS` entity. The default specification is written

```
SOLUTIONPARAMETERS
  gExcelOutput := ON ;
```

With the above specification, gPROMS will generate a temporary file called `ProcessName.xls`[2]. However, it is recommended that the default filename is overridden using the following specification:

`gExcelOutput := "`*FullFileName*`" ;`

In this case, the results will be stored in `FullFileName.xls`. Where `FileName.xls` corresponds to the full pathname of the new Excel file (e.g. C:\My Documents\MyResults.xls).

In the latter case, if the file already exists (*e.g.* the process has already been executed at least once), the file will be called `FileName[2].xls`. If this file also exists, gPROMS will increment the number in brackets until a new file can be generated without overwriting an existing one.

[2]This will be created in the `output` directory of the gPROMS execution directory used during the execution of the activity

C.3 Format of the Microsoft Excel output

The output of the simulation is written into several worksheets within the Microsoft Excel workbook. The first worksheet, called "Details", contains a list of units and variables. The data for each variable are stored in individual worksheets.

When gPROMS executes the process, it automatically opens Excel and the output file along with a macro file (`Output.xls` in the `OC` directory). You can use the macros to select the worksheet that contains the data for a specific variable. Simply select the cell in `Details` that contains the name of the variable that you want and then press `CTRL+SHIFT+g`.

C.4 Using the graph generation macro

A second macro is available that can generate simple 2-D plots. From anywhere in the workbook, press `CTRL+SHIFT+n` to start the macro. A window will then appear with a list of available units and variables. Selecting a variable and pressing the `Add` button either generates a new worksheet containing a copy of the data and the 2-D plot (for scalar variables) or brings up an option window (for arrays and distributions). If a variable depends on a number of different domains, one must be selected to be plotted on the abscissa by pressing the corresponding radio button. For the remaining domains, appropriate values must be entered into the corresponding boxes. Once all domains have been specified, pressing the `Add` button will generate the graph.

You can plot multiple instances of the same variable in the same graph (*e.g.* the mole fractions of all components) simply by entering a new value for the component index and pressing `Add`. However, at present, variables with different names cannot be plotted on the same graph. It is, of course, a simple procedure to copy the required data into a new worksheet and to create the graph manually.

Appendix D

gPLOT Output Channel

Apart from the gRMS and Microsoft Excel output channels, gPROMS provides a general purpose ASCII output channel, called gPLOT, that can be used to interface with any other software.

If `gPLOT := ON ;` is specified in the `SOLUTIONPARAMETERS` section of a `PROCESS` entity, then gPROMS will create an ASCII text file called

```
ProcessName.gPLOT
```

where `ProcessName` is the name of the `PROCESS` being executed.

gPROMS ModelBuilder will automatically load the gPLOT file into the "Results" Entity group within the Case folder. The file can easily be exported for use in applications outside ModelBuilder using the Export tool explained in the ModelBuilder User Guide.

Alternately, the name of the gPLOT file can be specified explicitly by using the alternative declaration in the `SOLUTIONPARAMETERS` section: `gPLOT := "`*FileName*`" ;`.

In this file, gPROMS will generally record results:

- at the initial time;
- just *before* each discontinuity;
- just *after* each discontinuity;
- at the regular recording interval specified by the user.

Each such result set will contain the corresponding value of the simulation time, followed by the values of all recorded variables. Each entry is on a separate line.

The precise format of the gPLOT file is as follows:

Line 1	The number of variables being monitored (n)
Next n lines	The names of variables in order
Next $n+1$ lines	"0" on the first line (the initial simulation time), followed by the initial values of the variables on the subsequent n lines
Next $n+1$ lines	A value of the simulation time on the first line, followed by the values of the variables at this time on the subsequent n lines
	< The above repeated at every reporting time, and before and after every discontinuity >
Last line	A terminator (-1.00000E+09)