MTT: Model Transformation Tools

August 2003
For version 5.0.

Peter Gawthrop, Geraint Bevan

General information about MTT is available at URL www.sf.net. MTT is distributed under the GNU GENERAL PUBLIC LICENSE (see Section A.2 [Copying], page 95). This manual is distributed under the GNU Free Documentation License (see Section A.1 [GNU Free Documentation License], page 88).

`http://mtt.sourceforge.net`

1 Introduction

MTT is a set of Model Transformation Tools based on bond graphs. **MTT** implements the theory to be found in the book "Metamodelling: Bond Graphs and Dynamic Systems" by Peter Gawthrop and Lorcan Smith published by Prentice Hall in 1996 (ISBN 0-13-489824-9).

It implements two features not discussed in that book:

- bicausal bond graphs and
- hierarchical bond graphs.

In the context of software, it has been said that one good tool is worth many packages. UNIX is a good example of this philosophy: the user can put together applications from a range of ready made tools. This manual describes the application of this philosophy to dynamic system modeling embodied in **MTT** - a set of Model Transformation Tools each of which implements a single transformation between system representations.

System representations have two attributes.

- A Form: e.g. acausal bond graph, differential algebraic, linear state-space etc.
- A Language: e.g. Fig, Matlab, LaTeX, Reduce, postscript etc.

Transformations in **MTT** are accomplished using appropriate software (e.g. Octave/Matlab, Reduce) encapsulated in UNIX Bourne shell scripts. The relationships between the tools are encoded in a Make File; thus the user can specify a final representation and all the necessary intermediate transformations are automatically generated.

1.1 What is a representation?

Physical systems have many representations. These include

- a schematic diagram,
- a block diagram,
- a bunch of equations,
- a single differential(-algebraic) equation,
- simulation code,
- linearised state-space (or descriptor) equations,
- transfer function (of the linearised system),
- frequency response (of the linearised system),
- etc...

Each of these representations is related to other representations by an appropriate transformation (see Section 1.2 [What is a Transformation?], page 2. In many cases, a modeler is presented with a physical system and needs to make a model. In particular, a model, in this context, is a representation of the system appropriate to a particular use, for example:

- simulation,
- control system design,
- optimisation

- etc.

Indeed, for a given physical system, the modeler would need to derive a number of models. This process can be viewed as a series of steps; each involving a transformation between representations (see Section 1.2 [What is a Transformation?], page 2.

In this context, the following considerations are relevant.

- There is a unique 'core' representation of any system. There are many routes from this core representation, each leading to an appropriate model. There are many possible routes to this core representation from the physical system: the route chosen is a matter of convenience.
- Because the core representation is unique, it is easy to expand the tool-box to include additional transformations from the physical system to the core representation and additional transformations from the core representation to the mode.
- Transformation_1 probably cannot, and certainly should not, be completely automated. Engineering insight, knowledge and experience is essential to capture the essence (with respect to the particular use) of the physical system whilst discarding irrelevant form.
- Representation_1 should be 'close' in some sense to the Physical system.
- The core representation, and hence the representations leading to it, must contain enough information to generate all of the required models.
- Representations must be easily extensible: it must be possible to add extra components or attributes without restructuring the representation.

I happen to believe that Bond graphs (see Section 1.3 [Bond graphs], page 2) provide the most convenient and powerful basis for the core representation.

1.2 What is a transformation?

Each system representation (see Section 1.1 [What is a Representation?], page 1 is related to other representations by an appropriate transformation as follows:

- Physical system
- Transformation_1 —> Representation_1
- Transformation_2 —> Representation_2
- ...
- Transformation_N —> Core representation
- Transformation_N+1 —> Representation_N+1
- Transformation_N+2 —> Representation_N+2
- ...
- Transformation_N+M —> Model

Thus modeling is seen as a sequence of transformations between representations.

1.3 What is a bond graph?

Bond graphs provide a graphical high-level language for describing dynamic systems in a precise and unambiguous fashion. They make a clear distinction between structure (how

components are connected together), and behavior (the particular constitutive relationships, or physical laws, describing each component.

They can describe a range of physical systems including:

- Electrical systems
- Mechanical systems
- Hydraulic systems
- Chemical process systems

More importantly, they can describe systems which contain subsystems drawn from all of these domains in a uniform manner.

Bond graphs are made up of components (see Section 1.6 [Components], page 4) connected by bonds (see Section 1.5 [Bonds], page 4) which define the relationship between variables (see Section 1.4 [Variables], page 3).

1.4 Variables

In bond graph terminology there are four sorts of variables:

- *effort* variables
- *flow* variables
- *integrated effort* variables
- *integrated flow* variables

Examples of *effort* variables are

- voltage
- pressure
- force
- torque
- temperature

Examples of *flow* variables are

- current
- volumetric flow rate
- velocity
- angular velocity
- heat flow

Examples of integrated *flow* variables are

- charge
- volume
- momentum
- angular momentum
- heat

1.5 Bonds

Bonds connect components (see Section 1.6 [Components], page 4) together. Each bond carries two variables:

- an effort (see Section 1.4 [Variables], page 3) variable and
- a flow (see Section 1.4 [Variables], page 3) variable.

Each bond has three notations associated with it:

- a half-arrow,
- a causal stroke and
- a causal half-stroke.

The half-arrow indicates two things:

- the direction of power (or pseudo power) flow and
- the side of the bond associated with the flow variable.

The causal stroke indicates two things:

- the effort variable is imposed at the same end as the stroke and
- the flow variable is imposed at the opposite end to the stroke.

The causal half-stoke indicates one thing:

- if it is on the effort side of the bond, the effort variable is imposed at the same end as the stroke or
- if it is on the flow side of the bond, the flow variable is imposed at the opposite end to the stroke.

1.6 Components

Components provide the building blocks of a dynamic system when connected by bonds (see Section 6.5.1.2 [bonds], page 28). Components have the following attributes:

`ports` provide the connections to other components (see Section 1.6.1 [Ports], page 4)

`constitutive relationships`
: define how the port-variables are related (see Section 1.6.2 [Constitutive relationship], page 5)

1.6.1 Ports

Components have one or more ports. Each port carries two variables, and effort and a flow variable (see Section 1.4 [Variables], page 3). Any pair of ports can be connected by a bond (see Section 1.5 [Bonds], page 4); this connection is equivalent to saying that the effort variables at each port are identical and that the flow variables at each port are identical.

Ports are implemented in **MTT** using named SS components. (see Section 6.5.1.9 [Named SS components], page 31).

The direction of the named SS components. (see Section 6.5.1.9 [Named SS components], page 31) is coerced (see Section 6.5.1.10 [Coerced bond direction], page 31) to have the same direction as the bons connected to the corresponding port. Thus the direction of the direction of the named SS components has no significance unless the component is at the top level.

1.6.2 Constitutive relationship

The constitutive relationship of a component defines how the port variables are related. This relationship may be linear or non-linear. This typically contains symbolic parameters (see Section 1.6.3 [Symbolic parameters], page 5) which may be replaced, for the purposes of numerical analysis by numeric parameters (see Section 1.6.4 [Numeric parameters], page 5).

1.6.3 Symbolic parameters

The constitutive relationship of a system component (see Section 1.6 [Components], page 4) typically contains symbolic parameters. For example a resistor may have a symbolic resistance r. It is convenient to leave such parameters as symbols when viewing equations or when performing symbolic analysis such as differentiation.

However, **MTT** allows replacement of symbolic parameters by numeric parameters (see Section 1.6.4 [Numeric parameters], page 5) when appropriate.

1.6.4 Numeric parameters

Numerical parameters are needed to give specific values to symbolic parameters (see Section 1.6.3 [Symbolic parameters], page 5) for the purposes of numeric analysis; for example: simulation, graph plotting or use within a numerical package such as Octave (see Section 10.4 [Octave], page 79).

1.7 Algebraic loops

Following Chapter 3 of the book, algebraic loops appear as under-causal components in the bond graph. It is up to the modeler to indicate how these loops are to be resolved by adding appropriate SS elements.

In particular if zero junction is undercausal an SS:loop component (with effort output indicated by a causal stroke) with the following label file entry:

```
loop SS unknown,zero
```

For more information, refer to: “Metamodelling: Bond Graphs and Dynamic Systems” by Peter Gawthrop and Lorcan Smith published by Prentice Hall in 1996 (ISBN 0-13-489824-9).

1.8 Switched systems

Some systems contain switch-like components. For example an electrical system may contain on-off switches and diodes and a hydraulic system may shut-off valves and non-return valves.

Such systems are sometimes called hybrid systems. The modelling an simulation of such systems is the subject of current research. **MTT** implements a simple pragmatic approach to the modelling and simulation of such systems via two new Bond Graph components:

`ISW` a switched `I` component

`CSW` a switched `C` component

These switches are user controlled through the logic representation (see Section 4.4 [Simulation logic], page 21).

2 User interface

There are two user interfaces to **MTT**: a command line interface (see Section 2.2 [Command line interface], page 6) and a menu-driven interface (see Section 2.1 [Menu-driven interface], page 6).

2.1 Menu-driven interface

The Menu-driven interface for **MTT** is invoked as:

```
xmtt
```

This will bring up a menu which should be self explanatory :-). Various messages will be echoed in the window from whence **xMTT** was invoked.

2.2 Command line interface

The command line interface for **MTT** is of the form:

```
mtt [options] <system_name> <representation> <language>
```

`[options]`
: the (optional) option switches (see Section 2.3 [Options], page 6)

`<system_name>`
: the name of the system being transformed

`<representation>`
: the mnemonic for the system representation (see Section 6.1 [Representation summary], page 25)

`<language>`
: the mnemonic for language for the representation (see Chapter 9 [Languages], page 78)

for example

```
mtt rc rep view
```

creates a view of the report describing system rc and

```
mtt rc sm m
```

creates an m file (suitlable for Octave or Matlab) containing state matrices describing the system rc.

2.3 Options

MTT has a number of optional switches to control its operation. These are invoked immediately after 'mtt' on the command line; for example:

```
mtt -o -ss -cc syst cbg view
```

invokes the `-o`, `-ss`, and `-cc` options.

If you wish to use an option all the time, use the alias function appropriate to the shell you are using. For example, using bash:

```
alias mtt='mtt -o -ss -cc'
```

Means that the previous example can be executed using

```
mtt syst cbg view
```

2.3.1 Available options

`-A` solve algebraic equations symbolically

`-abg` start at abg.m representation

`-ae <method>`
algebraic equation solver: reduce|hybrd|dassl|hooke

`-cc` C++ code generation

`-cr` use cr before resolving equations

`-D` debug – leave log files etc

`-d <dir>` use directory <dir>

`-dc` maximise derivative (not integral) causality

`-dr <dir>` use files contained in <dir>

`-I` prints more information

`-i <method>`
integration method: implicit|euler|rk4|dassl

`-o` ode is same as dae

`-oct` use oct files in place of m files where appropriate

`-opt` optimise code generation (equivalent to -optl)

`-optl` locally optimise code generation (line-by-line)

`-ntmpvar <N>`
declare <N> temporary variables (default=$num_tmp_var)

`-p` print environment variables

`-partition`
partition hierachical system

`-pdf` generate pdf in place of ps

`-q` quiet mode – suppress MTT banner

`-r` reset time stamp on representation

`-s` generate sensitivity BG (use mtt -s sSys rep lang)

`-ss` use steady-state info to initialise simulations

`-stdin` read input data from standard input for simulations

`-sub <subsys>`
operate on subsystem <subsys>

`-t` tidy mode (default)

`-trace` just indicate what mtt will do - but do not do it

`-u` untidy mode (leaves files in current dir)

`-v` verbose mode (multiple uses increment the verbosity)

`--version`
print version and exit

`--versions`
print version of mtt and components and exit

`-viewlevel <N>`
view <N> levels of hierachy

2.3.1.1 Experimental options

There are some experimental options. These have not yet been heavily tested and should be used with caution.

`-optg` globally optimise code generation (full vector)

`-make-sort`
use sorted equations (sese, generated by make)

`-no-reduce`
try not to use symbolic algebra

`-sort` use sorted equations (sese, generated by seqn)

2.3.1.2 Deprecated options

C code generation is now deprecated in favour of C++ generation

`-c` c-code generation

2.3.2 Model specific options

It is often desirable to keep the options used to build a model associated with the directory in which the model is contained, along with information about compiler options or paths to component libraries. In these cases, it is convenient to create an executable shell script, say `call_mtt` which sets environment variables and options before calling mtt proper, for example:

```
#! /bin/sh
# call_mtt: sets model specific environment and options

## Processor flags

# use 32 bit Reduce
export SYMBOLIC="reduce"

# set compiler debug and optimization options
export MTT_CXXFLAGS="${MTT_CXXFLAGS}\
        -Wall-Wno-unused -Wuninitialized\
        -O1 -march=pentiumpro -save-temps"

## Model flags
```

```
# use directory ../.. as base directory
export DIR=${DIR:-${PWD}/../..}

# additional components are in directory ../comp
export MTT_COMPONENTS="${MTT_COMPONENTS}:${DIR}/comp"

# additional constitutive relationships are in ../cr
export MTT_CRS="${MTT_CRS}:${DIR}/cr"

# C header files in ../cr/h
export MTT_CXXINCS="${MTT_CXXINCS} -I${DIR}/cr/h"

# call mtt with model specific options
exec ${MTTPATH}/mtt -cc -cr -i euler -D -optl -ntmpvar 1200 $*
```

which may then be used in place of mtt on the command line,

```
./call_mtt sys odeso gnuplot
```

2.4 Utilities

MTT provides some utilities to help you keep track of model building and to keep things clean and tidy. The commands, and there purpose are:

`mtt help` Lists the help/browser commands

`mtt copy <system>`
: Copies the system (ie directory and enclosed files) to the current working directory.

`mtt rename <old_name> <new_name>`
: Renames all of the defining representations (see Section 6.2 [Defining representations], page 27) and textually changes each file appropriately.

`mtt <system> clean`
: Remove all files generated by **MTT** associated with system 'system'.

`mtt clean` Remove all files generated by **MTT** associated with all systems within the current directory.

`mtt system representation vc`
: Apply version control to representation 'representation' of system 'system'.

`mtt system vc`
: Apply version control to all representations (under version control) system 'system'.

These are described in more detail in the following sections.

2.4.1 Help

MTT implements a browser to keep track of all the systems, subsystems and constitutive relationships that you, and others may write. It is invoked in the following ways:

```
mtt help representations
mtt help components
mtt help examples
mtt help crs
mtt help representations <match_string>
mtt help components <match_string>
mtt help examples  <match_string>
mtt help crs <match_string>
mtt help <component_or_example_or_CR_name>
```

2.4.1.1 help representations

The command

```
mtt help representations
```

lists all of the representations (see Chapter 6 [Representations], page 25) available in **MTT**. These may change as the version number of **MTT** increases.

The command

```
mtt help representations <match_string>
```

lists those representation which contain the string `match_string`. This string can be any regular expression (see standard Linux documentation under `awk`). For example

```
mtt help representations descriptor
```

gives all representations containing the word `descriptor`.

2.4.1.2 help components

The command

```
mtt help components
```

lists all of the components (see Section 1.6 [Components], page 4) available in **MTT**. These may change as the version number of **MTT** increases.

The command

```
mtt help components <match_string>
```

lists those component which contain the string `match_string`. This string can be any regular expression (see standard Linux documentation under `awk`). For example

```
mtt help components source
```

gives all components containing the word `component`.

2.4.1.3 help examples

This command provides a good way to get started in **MTT**. having found an interesting example, copy it to your working directory using

```
mtt copy <example_name>
```

(see Section 2.4.2 [Copy], page 11)

```
mtt help examples
```

lists all of the examples available in **MTT**. This list will change as more examples are added.

The command

```
mtt help examples <match_string>
```

lists those component which contain the string `match_string`. This string can be any regular expression (see standard Linux documentation under `awk`). For example

```
mtt help examples pharmokinetic
```

gives all examples containing the word `pharmokinetic`.

2.4.1.4 help crs

The command

```
mtt help crs
```

lists all of the constitutive relationships (see Section 1.6.2 [Constitutive relationship], page 5) available in **MTT**. These may change as the version number of **MTT** increases.

The command

```
mtt help crs <match_string>
```

lists those constitutive relationships which contain the string `match_string`. This string can be any regular expression (see standard Linux documentation under `awk`). For example

```
mtt help crs sin
```

gives all crs containing the word `sin`.

2.4.1.5 help <name>

The command

```
mtt help <name>
```

gives a detailed description of the entity called `name`.

2.4.2 Copy

MTT provides a way of copying examples to your working directory:

```
mtt copy <example_name>
```

Use the command

```
mtt help examples
```

(see Section 2.4.1.3 [help examples], page 10) to find something of interest.

Note that components and constitutive relationships are automatically copied when required.

2.4.3 Clean

MTT generates a lot of representations in a number of languages. Some of these you will edit yourself; others can always be recreated by **MTT**. It makes sense, therefore to have a utility that removes all of these other files when you have finished actively working with a particular system. These are two versions:

1. `mtt system clean`
2. `mtt clean`

The first removes all files that can be regenerated with **MTT** associated with system 'system'; the second removes all such files associated with all systems in the current working directory.

The files which remain after such a clean are the Defining representations (see Section 6.2 [Defining representations], page 27).

2.4.4 Version control

When you are working on a modeling project, it is easy to forget what changes you made to a system and why you made them. Sometimes, you may regret some changes and wish to revert to an earlier version: even if you use .old files this may be difficult to achieve safely.

These are very similar problems to those faced by software developers and can be solved in the same way: using version control.**MTT** provides version control using the standard GNU Revision Control System (RCS). This is hidden from the user, but is fully complementary to direct use of RCS (e.g. via emacs vc commands) to the more experienced user who wishes to do so.

The only files that you should ever change (i.e. the ones never overwritten by **MTT**) are the Defining representations (see Section 6.2 [Defining representations], page 27).

All of the files, with the exception of `system_abg.fig`, are initially created by **MTT** and contain the RCS header for version control.

The **MTT** version control will automatically expand this part of the text to include all change comments that you give it – so will direct use of RCS (e.g. via emacs vc commands)

The **MTT** version commands are as follows:

`mtt system representation vc`
: Apply version control to representation 'representation' of system 'system'.

`mtt system vc`
: Apply version control to all representations (under version control) system 'system'.

The first is appropriate after you have made a revision to a single file. It will prompt you for a change comment; this will be automatically included in the file header. In addition, enough information will be saved to enable any version to be retrieved via RCS.

The second is appropriate to record the state of the entire model. This assumes that all relevant files have been recorded by the first version of the command. Once again, old versions of the entire model can be retrieved using the relevant RCS commands.

A subdirectory 'RCS' is created to hold this information. You need not bother about the contents, except that you must not delete any files within 'RCS'.

3 Creating Models

MTT helps you to analyse and transform system models – ultimately the process of capturing the real world in a model is up to you. This chapter discusses the **MTT** aspects of creating a model. For convenience, this is divided into creating simple models and creating complex models.

3.1 Quick start

It is probably worth a quick skim though **MTT** to get a flavour of what it can do before plunging into the detail of the rest of this document. Here is a series of commands to do this.

Copy an initial set of files describing the bond graph.

```
mtt copy rc
```

Move to it.

```
cd rc
```

View the acausal bond graph (the system is called "rc").

```
mtt rc abg view
```

View the causal bond graph of the system.

```
mtt rc cbg view
```

View the corresponding ordinary differential equations (ode).

```
mtt rc ode view
```

View the system (output) step response

```
mtt rc sro view
```

An alternative (but more general) way of achieving the same result is

```
mtt -cc rc odeso view
```

View the system transfer function

```
mtt rc tf view
```

View the log modulus frequency response of the system.

```
mtt rc lmfr view
```

View the log modulus frequency response of the system for 100 logarithmically spaced frequencies in the range 0.1 to 10 radians per second.

```
mtt rc lmfr view 'W=logspace(-1,1,100);'
```

MTT has a report generation ((see Section 6.17 [Report], page 63) facility which can generate a hypertext description of the system.

```
mtt rc rep hview
```

The report contents are specified by the rep representation (see Section 6.17 [Report], page 63), in this case the corresponding file is:

```
% %% Outline report file for system rc (rc_rep.txt)

mtt rc abg tex
mtt rc struc tex
```

```
mtt rc cbg ps
mtt rc ode tex
mtt rc ode dvi
mtt rc sm tex
mtt rc tf tex
mtt rc tf dvi
mtt rc sro ps
mtt rc lmfr ps
mtt rc odes h
mtt rc numpar txt
mtt rc input txt
mtt -cc rc odeso ps
mtt rc rep txt
```

A non-hypertext version can be viewed using:

```
mtt rc rep view
```

Now have a go at modifying the bond graph.

```
mtt rc abg fig
```

This brings up the bond graph in Xfig (see Section 10.2 [Xfig], page 79). Try creating a system with two rs and 2 cs.

More examples can be found using

```
mtt help examples
```

Details of an example can be found using

```
mtt help <example_name>
```

and copied using

```
mtt copy <example_name>
```

Lots of examples are available.

```
mtt help examples
```

lists them and

```
mtt copy <name>
```

gets you an example.

3.2 Creating simple models

For then purposes of this section, simple models are those which are built up from bond graphs involving predefined components. In contrast, more complex systems (see Section 3.3 [Creating complex models], page 16) need to be built up hierarchically.

The recommended sequence of steps to create a simple model is:

1. Decide on a name for the system; let us call it 'syst' for the purposes of this discussion.
2. Invoke the Bond Graph editor to draw the acausal Bond Graph.

   ```
   mtt syst abg fig
   ```

3. Draw the Bond Graph (see Section 6.5.1 [Language fig (abg.fig)], page 28), including the bonds (see Section 1.5 [Bonds], page 4), the components (see Section 1.6 [Components], page 4) and any artwork (see Section 6.5.1.15 [artwork], page 33) to make the Bond

Graph more readable. The graphical editor xfig is (see Section 10.2 [Xfig], page 79) is self-explanatory. The icon library is helpful here (see see Section 6.5.1.1 [icon library], page 28).

4. Add causal strokes (see Section 6.5.1.3 [strokes], page 29) where needed to define causality. As a general rule, use the minimum number of strokes needed to define the problem; this will often be only on the `SS` components. (see Section 6.5.1.6 [SS components], page 30).

   Save the bond graph.

5. View the corresponding causal bond graph.

   ```
   mtt syst cbg view
   ```

   1. At this stage, **MTT** will warn you that the labeled components do not appear in the label file - this can safely be ignored.
   2. **MTT** will indicate the percentage of components which are causally complete – ideally this will be 100%. Components which are not causally complete will be listed.
   3. A view of the causal bond graph will be created. The added causal strokes are indicated in blue, undercausal components in green and overcausal components in red.
   4. If the bond graph is causally complete, proceed to the next step, otherwise think hard and return to the first step.

6. At this stage, no constitutive relationships have been defined. Nevertheless, **MTT** will proceed in a semi-qualitative fashion by assuming that all constitutive relationships are unity (and therefore linear). It may be useful at this stage to view various derived representations to check the overall model properties before proceeding further. For example:

   1. View the system Differential-algebraic equations

      ```
      mtt syst dae view
      ```

   2. View the system state matrices

      ```
      mtt syst sm view
      ```

   3. View the system transfer function

      ```
      mtt syst tf view
      ```

   4. View the system step response

      ```
      mtt syst sro view
      ```

7. As well as creating the causal bond graph, **MTT** has also generated templates for other text files (see Section 6.2 [Defining representations], page 27) used to further specify the system. These can now be edited using your favorite text editor (see Section 10.3 [Text editors], page 79).

8. **MTT** will now generate the representations (see Section 6.1 [Representation summary], page 25)that you desire. For example the system can be simulated by

   ```
   mtt syst odeso view
   ```

   MTT will complain if a component is named in the bond graph but not in the label file and vice versa. This mainly to catch typing errors.

3.3 Creating complex models

Complex models – in distinction to simple models (see Section 3.2 [Creating simple models], page 14) – have a hierarchical structure. In particular, bond graph components can be created by specifying their bond graph. Typically, such components will have more than one port (see Section 1.6.1 [Ports], page 4); within each component, ports are represented by named SS components (see Section 6.5.1.9 [Named SS components], page 31); outwith each component, ports are unambiguously identified by labels (see Section 6.5.1.11 [Port labels], page 31) and vector labels (see Section 6.5.1.12 [Vector port labels], page 32).

Complex models are thus created by conceptually decomposing the system into simple subsystems, and then creating the corresponding bond graphs. The procedure for simple systems (see Section 3.2 [Creating simple models], page 14) is then followed using the top level system (see Section 3.3.1 [Top level], page 16); **MTT** then recursively operates on the lower level systems.

The report representation (see Section 6.17 [Report], page 63) provides a convenient way of viewing a complex system.

An example of such a system can be created as follows:

```
mtt copy twolink
mtt twolink rep hview
```

3.3.1 Top level

The top level of a complex model contains subsystems but is not, itself, contained by other systems. It has the following special features:

- its name is used in the mtt command as the system name.
- all named SS componenents (see Section 6.5.1.9 [Named SS components], page 31) are treated as ordinary SS components (see Section 6.5.1.6 [SS components], page 30).

4 Simulation

One purpose of modelling is to simulate the modeled dynamic system. Although this is just another transformation (see Section 1.2 [What is a Transformation?], page 2) and therefore is covered in the appropriate chapter (see Chapter 6 [Representations], page 25), it is important enough to be given its own chapter.

Simulation is typically performed using an appropriate simulation language (which is often inappropriately conflated with modelling tools). **MTT** provides a number of alternative routes to simulation based on the following representations (see Chapter 6 [Representations], page 25):

`cse` constrained-state differential equation form

`ode` ordinary differential (or state-space) equations

in each case these equations may be linear or nonlinear.

Special cases of numerical simulation, appropriate to *linear* systems, are:

`ir` impulse response - state

`iro` impulse response - output

`sr` impulse response - state

`sro` impulse response - output

There are a number of languages (see Chapter 9 [Languages], page 78) which can be used to describe these representations for the purposes of numerical simulation:

`m` `octave` a high-level interactive language for numerical computation.

`c` `gcc` a c compiler.

`cc` `g++` a C++ front-end to gcc.

There are a number solution algorithms available:

- explicit solution via the matrix exponential
- backward Euler integration (implicit)
- forward Euler integration (explicit)
- Runge Kutta IV integration (explicit, fixed step)
- Hybrd algebraic solver (MINPACK, Octave fsolve)

However, all combinations of representation, language and solution method are not supported by **MTT** at the moment. Given a system 'system', some recommended commands are:

`mtt system iro view`
creates the impulse response of a *linear* system via the system_sm.m representation using explicit solution via the matrix exponential.

`mtt system sro view`
creates the step response of a *linear* system via the system_sm.m representation using explicit solution via the matrix exponential.

`mtt -cc system odeso view`
: creates the response of a *nonlinear* system via the system_ode.cc representation using implicit integration.

`mtt -cc -i euler system odeso view`
: creates the response of a *nonlinear* system via the system_ode.cc representation using euler integration.

Simulation parameters are described in the system_simpar.txt file (see Section 4.2 [Simulation parameters], page 18).

The steady-state solution of a system can also be "simulated"(see Section 4.1 [Steady-state solutions], page 18).

4.1 Steady-state solutions

4.1.1 Steady-state solutions (odess)

MTT can compute the steady-state solutions of an ordinary differential equation; this used the octave function 'fsolve'. The solution is computed as a function of time using the input specified in the input file. The simulation parameter file (see Section 4.2 [Simulation parameters], page 18) is used to provide the time scales.

For example

```
mtt copy rc
cd rc
mtt rc odess view
```

4.1.2 Steady-state solutions (ss)

A rudimentary form of steady-state solution exists in mtt. The steady states and inouts are supplied by the user in the file system_simpar.r and the corresponding output and sate derivative computed by **MTT** using

```
mtt system ss view
```

For example

```
mtt copy rc
cd rc
mtt rc sspar view
mtt rc ss view
```

4.2 Simulation parameters

Simulation parameters are set in the system_simpar.txt file. At the moment this sets the following variables:

- LAST the last simulation time
- DT the incremental time (for plotting)
- STEPFACTOR the number of integration steps per DT – thus the integration interval is DT/STEPFACTOR
- WMIN Minimum frequency = 10^WMIN

- WMAX Maximum frequency = 10^WMAX
- WSTEPS Number of Frequency steps.
- INPUT The input index for frequency response

There are a number of solution algorithms

- Euler basic Euler integration (see Section 4.2.1 [Euler integration], page 19). This method is simple, but not recommended for stiff systems.
- Implicit semi-implicit integration (see Section 4.2.2 [Implicit integration], page 19) - uses the smx representation to give stability.
- Runge Kutta IV fixed step Runge Kutta fourth order integration (see Section 4.2.3 [Runge Kutta IV integration], page 20).
- Hybrd numerical algebraic equation solver

4.2.1 Euler integration

Euler integration approximates the solution of the Ordinary Differential Equation

```
dx/dt = f(x,u)
```

by

```
x := x + f(x,u)*DDT
```

where

```
DDT = DT/STEPFACTOR
```

If the system is linear, stability is ensured if the integer STEPFACTOR is chosen to be greater than the real number

```
(maximum eigenvalue of -A)*DT/2
```

where A is the nxn matrix appearing in

```
f(x,u) = Ax + Bu
```

If the system is non linear, the linearised system matrix A should act as a guide to the choice of STEPFACTOR.

4.2.2 Implicit integration

Implicit integration approximates the solution of the Ordinary Differential Equation

```
dx/dt = f(x,u)
```

by

```
(I-A*DT)x := (I-A*DT)x + f(x,u)DT
```

where A is the linearised system matrix. This implies the solution of N (=number of states) linear equations at each sample interval. The OCTAVE version used the '\' operator to solve the set of linear equations, the C version uses LU decomposition.

If the system is linear, stability is ensured unconditionaly. If the system is non-linear, then the method still works well.

This method is nice in that choice of DT trades of accuracy against computation time without compromising stability. In addition, the correct stready-state values are achieved.

This approach can also be used for constrained state equations of the form:

```
E(x) dx/dt = f(x,u)
```

where E(x) is a state-dependent matrix. The approximate solution is then given by:

```
(E(x)-A*DT)x := (E(x)-A*DT)x + f(x,u)DT
```

which reduces to the ordinary differential equation case when E(x)=I.

The _smx representation includes the E matrix.

4.2.3 Runge Kutta IV integration

Runge Kutta IV approximates the solution of the Ordinary Differential Equation

```
dx/dt = f(x,t)
```

by

```
x := x + (DT/6)*(k1 + 2*k2 + 2*k3 + k4)
```

where

```
k1 := f(x,t)
k2 := f(x+(1/2)*k1,t+(1/2)*DT)
k3 := f(x+(1/2)*k2,t+(1/2)*DT)
k4 := f(x+k3,t+DT)
```

The **MTT** implementation of Runge-Kutta integration is a fourth order, fixed-step, explicit integration method.

For some systems of equations, the increased accuracy of using a fourth order method can allow larger step-lengths to be used than would allowed by the lower order Euler integration method.

It should be noted that during the interemediate calculations (k1...k4), the input vector `u` is not advanced w.r.t. time; the system inputs are assumed to be constant over the period of the integration step-length.

4.2.4 Hybrd algebraic solver

The hybrd algebraic solver of MINPACK, which is used by Octave in the `fsolve` routine, may be used in conjunction with one of the other integration methods to solve semi-explicit, index 1, differential algebraic equations; these may be generated in **MTT** models by use of `unknown` SS Components see Section 6.7.1 [SS component labels], page 38.

This method requires that compiled simulation code is used; either -cc or -oct. To perform a simulation based on a model `sys`,

```
mtt -cc -ae hybrd -i euler sys odeso view
```

MTT will attempt to minimise the residual error at each integration time-step using the hybrd routine.

This method of simulation is particularly well suited to stiff systems where very fast dynamics are of little interest. Care must be taken to ensure that an acceptable level of convergence is achieved by the solver for the system under investigation.

4.3 Simulation input

This is defined in the system_input.txt file. A default file is created automatically by **MTT**. This is done explicitly by

```
mtt system input txt
```

If the file already exists, the same command checks that all inputs are defined and that all defined inputs exist in the system and promts the user to correct discrepancies.

Inputs are defined by the full system name appearing in the structure file (see Section 6.8 [Structure (struc)], page 47). They can depend on states (again defined by name), time (defined by t) and parameters

For example:

```
system_pump_l_1_u       = 4e5*atm;
system_pump_r_1_u       = 4e5*(t<10)*atm;
system_ss_i             = 0*kg;
system_ss_o             = 3e-3*kg;
system_v_1_u            = (t>10);
system_v_ll_1_u         = 1;
system_v_lr_1_u         = (t<10);
system_v_ul_1_u         = 0;
system_v_ur_1_u         = (t>10);
```

4.4 Simulation logic

This is defined in the system_logic.txt file. A default file is created automatically by **MTT**. This is done explicitly by

```
mtt system logic txt
```

If the file already exists, the same command checks that the logic corresponding to all switch states (see Section 1.8 [Switched systems], page 5) are defined and that all defined logic exists in the system and promts the user to correct discrepancies.

Logical inputs are defined by the full system name corresponding to MTT_switch components appearing in the structure file (see Section 6.8 [Structure (struc)], page 47) *with '_logic' appended*. They can depend on states (again defined by name), time (defined by t) and parameters

For example:

```
bounce_ground_1_mtt_switch_logic = bounce_intf_1_mtt3<0;
```

4.5 Simulation initial state

This is defined in the system_state.txt file. A default file is created automatically by **MTT**. This is done explicitly by

```
mtt system state txt
```

If the file already exists, the same command checks that all states are defined and that all defined states exist in the system and prompts the user to correct discrepancies.

States are defined by the full system name appearing in the structure file (see Section 6.8 [Structure (struc)], page 47). They can depend on parameters. For example

```
system_c_l = (1e4/k_l)/kg;
system_c_ll = (1e4/k_s)/kg;
system_c_lr = (1e4/k_s)/kg;
system_c_u = (1e4/k_l)/kg;
```

4.6 Simulation code

simulation code can be generated by **MTT** in the form of the `ode2odes` transformation. This can be produced in a number of languages, including .m, .oct, C and C++ see Chapter 9 [Languages], page 78.

To generate simulation code in C (deprecated):

```
mtt -c [options] sys ode2odes c
```

Similarly, to generate C++ code:

```
mtt -cc [options] sys ode2odes cc
```

To generate an executable based on the C++ representation:

```
mtt -cc [options] sys ode2odes exe
```

4.6.1 Dynamically linked functions

Some model representations can be compiled into dynamically loaded code (shared objects) which are compiled prior to use in other modelling and simulation environments; in particular, .oct files can be generated for use in GNU Octave (see Section 10.4.2 [Creating GNU Octave .oct files], page 81) and .mex files can be generated for use in Matlab (see Section 10.4.3 [Creating Matlab .mex files], page 82) or Simulink (see Section 10.4.4 [Embedding MTT models in Simulink], page 82). The use of compiled (and possibly compiler-optimised) code can offer significant processing speed advantages over equivalent interpreted functions (e.g. .m files) for computationally intensive procedures.

The C++ code generated by **MTT** allows the same code to be generated as standalone code, Octave .oct files or Matlab .mexglx files. Although **MTT** usually takes care of the compilation options, if it is necessary to compile the code on a machine on which **MTT** is not installed, the appropriate flag should be passed to the compiler pre-processor:

- `-DCODEGENTARGET=STANDALONE`
- `-DCODEGENTARGET=OCTAVEDLD`
- `-DCODEGENTARGET=MATLABMEX`

4.7 Simulation output

The view (see Section 10.1 [Views], page 79) representation provides a graphical representation of the results of a simulation; the postscript language provides the same thing in a form that can be included in a document.

These are two simulation output representations

`odes` ordinary differential equation solution (states)

`odeso` ordinary differential equation solution (output)

Particular output variables can be selected by adding a fourth argument in one of 2 forms

`'name1;name2;..;namen'`
: plot the variables with names na1 .. namen against time

`'name1:name2'`
: plot the variable with name2 against that with name 1

An example of plotting a single variable against time is:

```
mtt -o -cc -ss OttoCycle odeso ps 'OttoCycle_cycle_V'
```

An example of plotting one variable against another is:

```
mtt -o -cc -ss OttoCycle odeso ps 'OttoCycle_cycle_V:OttoCycle_cycle_P'
```

4.7.1 Viewing results with gnuplot

Simulation plots may be conveniently selected, viewed with gnuplot and saved to file (in PostScript format) using the command

```
mtt [options] rc odeso gnuplot
```

This will cause a menu to be displayed, from which states and outputs may be selected for viewing. Clicking on a *parameter name* will, by default, cause the time history of the selected parameter to be displayed.

As with **xMTT** (see Section 2.1 [Menu-driven interface], page 6), the Wish Tcl/Tk interpreter must be installed to make use of this feature.

4.7.2 Exporting results to SciGraphica

Simulation results can be converted into an XML-format SciGraphica (version 0.61) *.sg* file with the command

```
mtt [options] sys odes sg
```

The SciGraphica file will contain two worksheets, X_sys and Y_sys, containing the state and output time-histories from the simulation.

5 Sensitivity models

The sensitivity model of a system is a set of equations giving the sensitivity of the system outputs with respect to system parameters. **MTT** has built in methods for assisting with the development of such models.

This feature is experimental at the moment, but the following example gives an idea of what can be achieved.

```
mtt copy rc
cd rc
mtt -s src ode view
mtt -s src odeso view
```

The sensitivity system src is automatically created from the system rc using the predefined sR and sC components together with vector junctions (see Section 6.5.1.14 [Vector components], page 32). The four outputs are the two system outputs plus the two sensitivity functions.

An alternative route is to create the sensitivity functions by symbolic differentiation. The following sensitivity representations are available:

`scse` sensitivity constrained-state equations

`sm` sensitivity state matrices

`scsm` sensitivity constrained-state matrices

6 Representations

As discussed in Section 1.1 [What is a Representation?], page 1, a system has many representations. The purpose of **MTT** is to provide an easy way to generate such representation by applying the appropriate sequence of transformations. The representations supported by **MTT** are summarised in Section 6.1 [Representation summary], page 25.

There is a two-fold division of representations into those with which the user defines the system and its various attributes, and those which are derived from these. The *defining representations* are listed in Section 6.2 [Defining representations], page 27.

Each representation is implemented in one or more languages depending on its use. These languages are discussed in Chapter 9 [Languages], page 78 and are associated with appropriate tools for modifying or viewing the representations.

6.1 Representation summary

Some of the the representations available in **MTT** are (in alphabetical order):

`abg`	acausal bond graph
`cbg`	causal bond graph
`cr`	constitutive relationship for each subsystem
`cse`	constrained-state equations
`csm`	constrained-state matrices
`dae`	differential-algebraic equations
`daes`	dae solution - state
`daeso`	dae solution - output
`def`	definitions - system orders etc.
`desc`	Verbal description of system
`dm`	descriptor matrices
`ese`	elementary system equations
`fr`	frequency response
`input`	numerical input declaration
`ir`	impulse response - state
`iro`	impulse response - output
`lbl`	label file
`lmfr`	loglog modulus frequency response
`lpfr`	semilog phase frequency response
`nifr`	Nichols style frequency response
`numpar`	numerical parameter declaration

`nyfr`	Nyquist style frequency response
`obs`	observer equations for CGPC
`ode`	ordinary differential equations
`odes`	ode solution - state
`odes`	ODE simulation header file
`odeso`	ode solution - output
`odess`	ode numerical steady-states - states
`odesso`	ode numerical steady-states - outputs
`rbg`	raw bond graph
`rep`	report
`rfe`	robot-form equations
`sabg`	stripped acausal bond graph
`simp`	simplification information
`sm`	state matrices
`smx`	state matrices containing explicit states and inputs
`sms`	ode
`smss`	SM simulation header file
`sr`	step response - state
`sro`	step response - output
`ss`	steady-state equations
`sspar`	steady-state definition
`struc`	structure - list of inputs, outputs and states
`sub`	Executable subsystem list
`sub`	LaTeX subsystem list
`sympar`	symbolic parameters
`tf`	transfer function

A complete list can be found via the `help representations` command (see Section 2.4.1.1 [help representations], page 10).

Many of these representations have more than one language (see Chapter 6 [Representations], page 25) associated with them.

Some of these representations define the system (see Section 6.2 [Defining representations], page 27).

6.2 Defining representations

The following representations define the system and therefore must, ultimately, be defined by the user. However, all of these are assigned default values by **MTT** and may then be subsequently edited (see Section 10.3 [Text editors], page 79) viewed or operated on by the appropriate tools (see Chapter 10 [Language tools], page 79).

`system_abg.fig`
: the acausal bond graph (see Section 6.4 [Acausal bond graph (abg)], page 27)

`system_lbl.txt`
: the label file (see Section 6.7 [Labels (lbl)], page 37)

`system_desc.tex`
: the description file (see Section 8.2.2 [Detailed], page 77)

`system_simp.r`
: algebraic simplifications to make output more readable (see Section 6.10.2 [Symbolic parameters for simplification (simp.r)], page 50)

`system_subs.r`
: algebraic substitutions to resolve, eq trig. identities (see Section 6.10.1 [Symbolic parameters (subs.r)], page 50)

`system_simpar.txt`
: simulation parameters (see Section 4.2 [Simulation parameters], page 18)

`system_numpar.txt`
: numerical parameters (see Section 6.10.3 [Numeric parameters (numpar)], page 50)

`system_input.txt`
: the system input for simulations (see Section 4.3 [Simulation input], page 21)

`system_logic.txt`
: the switching logic for simulations (see Section 4.4 [Simulation logic], page 21)

`system_sspar.r`
: defines the system steady-state (see Section 4.1.2 [Steady-state solutions - symbolic (ss)], page 18)

6.3 Verbal description (desc)

Systems can be documented in LaTeX using the _desc.tex file. This file is included in the report (see Section 6.17 [Report], page 63) if the abg tex option is included in the rep.txt file. As usual, **MTT** provides a default text file to be edited by the user (see Section 10.3 [Text editors], page 79).

6.4 Acausal bond graph (abg)

The acausal bond graph is the main input to **MTT**. It is up to you, as a system modeler, to distill the essential aspects of the system that you wish to model and capture this information in the form of a bond graph.

The inexperienced modeler may wish to look in one of the standard textbooks and copy some bond graphs of systems to get going.

To create the acausal bond graph of system 'sys' in language fig type:

```
mtt sys abg fig
```

To create the acausal bond graph of system 'sys' in language m type:

```
mtt sys abg m
```

To view the acausal bond graph of system 'sys' type:

```
mtt sys abg view
```

6.5 Acausal bond graph - LaTeX ready (labg)

This representation exists in the fig (see Section 6.5.1 [Language fig (abg.fig)], page 28) only. It is used for high-quality figures generated using fig2dev with the -Lpstex option.

6.5.1 Language fig (abg.fig)

A bond graph is made up of:

`bonds` To connect components together.

`strokes` To indicate causality.

`components`
Either simple or compound.

`artwork` Irrelevant to the system but useful to the user.

An icon library of bonds, components and other symbols is available within xfig (see Section 6.5.1.1 [icon library], page 28).

6.5.1.1 Icon library

A number of predefined iconic symbols are available within xfig.

```
Click onto the library icon
Click onto the library pull-down menu and select BondGraph
Select iconic symbols from the presented list
```

6.5.1.2 Bonds

Bonds are represented by polylines with two segments. They must be the default style (i.e. plain not dashed or dotted). The shortest segment is taken to be the half-arrow. its positioning is significant because:

- It points in the direction of power flow; thus a bond normally points towards C, I and R components.
- the corresponding side of the bond indicates flow causality; the other side represents effort causality. This is significant when using causal half-strokes (see Section 6.5.1.3 [strokes], page 29). Please adopt the convention of having the half-arrows below horizontal bonds and to the right of vertical bonds.

6.5.1.3 Strokes

Causal strokes are represented by single-segment polylines. There are two sorts of strokes:

- *Full* strokes: these are the usual bond-graph strokes and determine both the effort and flow causality in the usual way. The *centre* of the stroke should be at about one end of the bond and be at right angles to it.
- *Half* strokes: these are an innovation in **MTT** and allow you to specify the effort and flow causality independently. The *end* of the stroke should be at about one end of the bond and be at right angles to it. If the causal half-stroke is on the *same* side as the half-arrow (see Section 6.5.1.2 [bonds], page 28) then it determines *flow* causality; if, on the other hand, it is on the *opposite* side to the half-arrow (see Section 6.5.1.2 [bonds], page 28) then it determines *effort* causality. Two half strokes on the *same*, but on *opposite* sides of the bond are equivalent to a a full stroke at the same end of the bond.

MTT is reasonably forgiving; but a neat diagram will be less ambiguous to you as well as to **MTT**.

Causality is indicated as follows:

- *Effort* is imposed at the *same* end as the stroke.
- *Flow* is imposed at the *opposite* end as the stroke.

6.5.1.4 Components

Components are represented by a text string in fig. The recommended style is: 20pt, Times-Roman and centre justified.

The component text string can be of the following forms:

`type` Just the type of the component is indicated. Components may be either Simple components (see Section 6.5.1.5 [Simple components], page 30) or Compound components (see Section 6.5.1.8 [Compound components], page 31). For example:

```
R
```

`type:label`
Both the type and the label of the component are given. The type must be a valid name (see Section 6.5.1.16 [Valid names], page 33.The name provides a link to more information to be found in See Section 6.7 [Labels (lbl)], page 37. For example:

```
R:r
```

`type:label:cr`
Not only are the type and the label of the component given, but also the component cr argument. The type must be a valid name (see Section 6.5.1.16 [Valid names], page 33.The name provides a link to more information to be found in See Section 6.7 [Labels (lbl)], page 37. For example:

```
R:r:flow,r
```

`type:label:expression`
Expression is a mathematical expression relating the effort (called mtt_e) to the flow (called mtt_f). For example the following three forms are equivalent

```
R:r:mtt_e=r*mtt_f
R:r:mtt_e-r*mtt_f=0
R:r:mtt_f=mtt_e/r
```

A non-linear example is:

```
R:r:mtt_e = sin(mtt_f)
```

`type*n` The name, together with the number '`n`' of repetitions of the component, are given. This repetition only makes sense if the component has an even number of ports (see Section 6.5.1.11 [Port labels], page 31); n copies of the component are concatenated with odd Named ports (see Section 6.5.1.11 [Port labels], page 31) of the component being connected to the even Named ports of the previous component in the chain in numerical order. This feature is particularly useful if the component is compound and can be used for, example to give a lumped approximation of a distributed system. For example:

```
MySystem*25
```

`type:label*n`
This complete form and is a combination of the simpler forms. For example:

```
MySystem:MyLabel*25
```

6.5.1.5 Simple components

The following simple components are defined in MTT.

`R`	Standard one-port R
`C`	Standard one-port C
`I`	Standard one-port I
`SS`	Source-sensor
`TF`	Transformer
`GY`	Gyrator
`AE`	Effort amplifier
`AF`	Flow amplifier
`CSW`	Switched one-port C
`ISW`	Switched one-port I

6.5.1.6 SS components

$$

`SS` components provide input and output variables for a system; Named SS components (see Section 6.5.1.9 [Named SS components], page 31) provide this for subsystems.

6.5.1.7 Simple components - implementation

Each simple component, with name NAME, is defined by two m files:

`NAME_cause.m`
defines the possible causal patterns for the component

`NAME_eqn.m`
: defines the equations generated

Only the experienced user would normally define simple components - Compound components (see Section 6.5.1.8 [Compound components], page 31) are recommended for DIY components.

6.5.1.8 Compound components

Compound components are systems described by bond graphs and implemented by MTT. They have special SS components, Named SS components (see Section 6.5.1.9 [Named SS components], page 31), to indicate connections to the encapsulating system.

Like any other system, they are described by a graphical Bond Graph description (see Section 6.5.1 [Language fig (abg.fig)], page 28), and a label file (see Section 6.7 [Labels (lbl)], page 37).

By convention, all of the files describing a component live in a directory with the same name as the component.

6.5.1.9 Named SS components

Named SS components provide the link from the system which *defines* compound component to the system which *uses* a compound component see Section 6.5.1.8 [Compound components], page 31. A named SS components is of the form `SS:[name]`;

Where 'name' is a name consisting of alphanumeric characters and underscore; for example:

```
SS:[Mechanical_1]
```

Each such named SS provides one of the ports (see Section 1.6.1 [Ports], page 4). The direction of the named SS components. (see Section 6.5.1.9 [Named SS components], page 31) is coerced (see Section 6.5.1.10 [Coerced bond direction], page 31) to have the same direction as the bond connected to the corresponding port. Thus the direction of the direction of the named SS components has no significance unless the component is at the top level of a system.

If a named SS component exists at the top level (see Section 3.3.1 [Top level], page 16) and is treated as an ordinary SS component with the given direction and with the attributes specified in the label file (see Section 6.7 [Labels (lbl)], page 37).

6.5.1.10 Coerced bond direction

Named SS components (see Section 6.5.1.9 [Named SS components], page 31) provide the mechanism for declaring the ports (see Section 1.6.1 [Ports], page 4) of a component. The corresponding bond has a direction. However, under some circumstances, it may be useful to reverse this direction. **MTT** provides a coercion mechanism for this: the the direction of the bond attached to the named SS component (see Section 6.5.1.9 [Named SS components], page 31) is replaced by the direction of the bond attached to the component port.

6.5.1.11 Port labels

Most multi-port components have ports see Section 1.6.1 [Ports], page 4)which display different behaviors; the exception to this is the junction (`0` and `1`) components. For this

reason, **MTT** provides a method for unambiguously identifying the ports of a multi-port component by port labels.

A port label is indicated by a name within parentheses of the form `[name]`, where 'name' is a name consisting of alphanumeric characters and underscore; for example:

```
[Mechanical_1]
```

This provides a label for corresponding to the component to which the nearest bond-end is attached.

The following rules must be be obeyed:

- If a component has any port labels at all, there must be one for each port of the component.

Port labels may be grouped into vector port labels (see Section 6.5.1.12 [Vector port labels], page 32). Components with compatible (ie containing the same number of ports) vector ports may be connected by a *single* bond (see Section 1.5 [Bonds], page 4); such a bond implies the corresponding number of bonds (one for each element of the vector port label). All such bonds inherit the same direction and any *explicit* causal strokes (see Section 6.5.1.3 [strokes], page 29)

6.5.1.12 Vector port labels

Port labels (see Section 6.5.1.11 [Port labels], page 31) may be grouped into vector port labels of the form `[name1,name2,name3]`.

```
[Mechanical_1,Electrical,Hydraulic_5]
```

6.5.1.13 Port label defaults

Whether impicitly or explicity, all ports of components (with the exception of 0 and 1 junctions) must have labels (see Section 6.5.1.11 [Port labels], page 31). However, these can be omitted from the bond graph in the following circumstances and default labels are supplied by **MTT**.

1. A single unlabled inport defaults to [in]
2. A single unlabled outport defaults to [out]

These defaults may, in turn be aliases (see Section 6.7.9 [Aliases], page 42) for port labels (see Section 6.5.1.11 [Port labels], page 31) or vector port labels (see Section 6.5.1.12 [Vector port labels], page 32). Combining the default and alias mechanism is a powerful tool for creating uncluttered, yet complex, bond graph models.

6.5.1.14 Vector Components

Vectors of components can be created in four cases: `0` junctions, `1` junctions, `SS` components and `SS` port components.

In each case, the presence of a vector component is indicated by a single port label (see Section 6.5.1.11 [Port labels], page 31) of one of two forms:

1. containing numerals from 1 to the order of the vector. Thus a vector of 3 components is indicated by a port label of the form [1,2,3].
2. 1: followed by the order of the vector. Thus a vector of 3 components is indicated by a port label of the form [1:3].

Within the corresponding label file (see Section 6.7 [Labels (lbl)], page 37), the components of a vector port can be accessed using _i where i is the corresponding index. Thus a port SS:[Electrical] appearing near the port label [1,2,3] could contain the port alias (see Section 6.7.9.1 [Port aliases], page 42)

```
%ALIAS  in Electrical_1,Electrical_2,Electrical_3
```

6.5.1.15 Artwork

You are encouraged to annotate your bond graphs extensively - this makes them an immediately readable document whilst retaining the precise and unambiguous expressive power of the bond graph.

You may add any Fig (see Section 9.1 [Fig], page 78) object to the bond graph as long as it will not be interpreted as part of the bond graph. The reccommended way to acheive this is to put the Bond Graph at depth 0,10,20 etc (ie depth modulo 10 is zero) and artwork at any other depth.

For compatibility with earlier versions of **MTT**, the following objects are ignored even at level 0. However, their use is strongly discouraged.

- Adding text is OK as long as it cannot be confused with components (see Section 6.5.1.4 [components], page 29). In particular, you can include invalid component characters such as white space, ", ', ! etc.
- Adding boxes, arcs etc is always OK.
- Adding dotted or dashes lines is always OK.

The stripped abg file (sabg) (see Section 6.6 [Stripped acausal bond graph (sabg)], page 36) shows only those parts of the diagram recognised by **MTT** and is therefore useful for distinguishing artwork.

6.5.1.16 Valid Names

A valid name is a text string containing alphanumeric characters. It must **NOT** contain underscore '_', hyphen '-', ':' or '*'.

The following names should be avoided

```
if endif
```

The following reserved words in reduce should also be avoided (with any case)

```
Commands ALGEBRAIC ANTISYMMETRIC ARRAY BYE CLEAR CLEARRULES COMMENT
CONT DECOMPOSE DEFINE DEPEND DISPLAY ED EDITDEF END EVEN FACTOR FOR
FORALL FOREACH GO GOTO IF IN INDEX INFIX INPUT INTEGER KORDER LET
LINEAR LISP LISTARGP LOAD LOAD PACKAGE MASS MATCH MATRIX MSHELL
NODEPEND NONCOM NONZERO NOSPUR ODD OFF ON OPERATOR ORDER OUT PAUSE
PRECEDENCE PRINT PRECISION PROCEDURE QUIT REAL REMFAC REMIND RETRY
RETURN SAVEAS SCALAR SETMOD SHARE SHOWTIME SHUT SPUR SYMBOLIC
SYMMETRIC VECDIM VECTOR WEIGHT WRITE WTLEVEL

Boolean Operators EVENP FIXP FREEOF NUMBERP ORDP PRIMEP

Infix Operators := = >= > <= < => + * / ^ ** . WHERE SETQ OR AND
MEMBER MEMQ EQUAL NEQ EQ GEQ GREATERP LEQ LESSP PLUS DIFFERENCE MINUS
```

```
TIMES QUOTIENT EXPT CONS Numerical Operators ABS ACOS ACOSH ACOT ACOTH
ACSC ACSCH ASEC ASECH ASIN ASINH ATAN ATANH ATAN2 COS COSH COT COTH
CSC CSCH EXP FACTORIAL FIX FLOOR HYPOT LN LOG LOGB LOG10 NEXTPRIME
ROUND SEC SECH SIN SINH SQRT TAN TANH

Prefix Operators APPEND ARGLENGTH CEILING COEFF COEFFN COFACTOR CONJ
DEG DEN DET DF DILOG EI EPS ERF FACTORIZE FIRST GCD G IMPART INT
INTERPOL LCM LCOF LENGTH LHS LINELENGTH LTERM MAINVAR MAT MATEIGEN MAX
MIN MKID NULLSPACE NUM PART PF PRECISION RANDOM RANDOM NEW SEED RANK
REDERR REDUCT REMAINDER REPART REST RESULTANT REVERSE RHS SECOND SET
SHOWRULES SIGN SOLVE STRUCTR SUB SUM THIRD TP TRACE VARNAME

Reserved Variables CARD NO E EVAL MODE FORT WIDTH HIGH POW I INFINITY
K!* LOW POW NIL PI ROOT MULTIPLICITY T

Switches ADJPREC ALGINT ALLBRANCH ALLFAC BFSPACE COMBINEEXPT
COMBINELOGS COMP COMPLEX CRAMER CREF DEFN DEMO DIV ECHO ERRCONT
EVALLHSEQP EXP EXPANDLOGS EZGCD FACTOR FORT FULLROOTS GCD IFACTOR INT
INTSTR LCM LIST LISTARGS MCD MODULAR MSG MULTIPLICITIES NAT NERO
NOSPLIT OUTPUT PERIOD PRECISE PRET PRI RAT RATARG RATIONAL RATIONALIZE
RATPRI REVPRI RLISP88 ROUNDALL ROUNDBF ROUNDED SAVESTRUCTR
SOLVESINGULAR TIME TRA TRFAC TRIGFORM TRINT

Other Reserved Ids BEGIN DO EXPR FEXPR INPUT LAMBDA LISP MACRO PRODUCT
REPEAT SMACRO SUM UNTIL WHEN WHILE WS
```

6.5.2 Language m (rbg.m)

The raw bond graph of system 'sys' is represented as an m file with heading:

```
function [rbonds, rstrokes,rcomponents,rports,n_ports] = sys_rbg
```

This representation is a half-way house between the fig (see Section 6.5.1 [Language fig (abg.fig)], page 28) and m (see Section 6.5.3 [Language m (abg.m)], page 35) representations. It contains the geometric information from the fig file in a form digestible by Octave (see Section 10.4 [Octave], page 79).

The five outputs of this function are:

- rbonds
- rstrokes
- rcomponents
- rports
- n_ports

rbonds is a matrix with

- one row for each bond (see Section 6.5.1.2 [bonds], page 28)
- columns 1 and 2 containing the x,y coordinates for one end of the bond

- columns 3 and 4 containing the x,y coordinates for the corner of the bond
- columns 5 and 6 containing the x,y coordinates for the other end of the bond

rstrokes is a matrix with (see Section 6.5.1.3 [strokes], page 29)

- one row for each stroke or half-stroke
- columns 1 and 2 containing the x,y coordinates for one end of the stroke
- columns 3 and 4 containing the x,y coordinates for the other end of the stroke

rcomponents is a matrix with (see Section 6.5.1.4 [components], page 29)

- one row for each component
- columns 1 and 2 containing the x,y coordinates of the component
- the remaining columns containing fig file information

rports is a matrix with (see Section 6.5.1.11 [Port labels], page 31)

- one row for each component port that is explicitly labeled
- columns 1 and 2 containing the x,y coordinates of the port label
- column 3 contains the port number.

n_ports is the number of ports associated with the system – i.e. the number of Named SS components (see Section 6.5.1.9 [Named SS components], page 31).

6.5.2.1 Transformation abg2rbg_fig2m

This transformation takes the acausal bond graph as a fig file (see Section 6.5.1 [Language fig (abg.fig)], page 28) and transforms it into a raw bond graph in m-file format (see Section 6.5.2 [Language m (rbg.m)], page 34).

This transformation is implemented in GNU awk (gawk). It scans both the fig file (see Section 6.5.1 [Language fig (abg.fig)], page 28) and the label file (see Section 6.7 [Labels (lbl)], page 37) and generates the rbg (see Section 6.5.2 [Language m (rbg.m)], page 34) with components sorted according to the label file. It also generates a file sys_fig.fig containing details of the bond graph with the components removed.

6.5.3 Language m (abg.m)

The acausal bond graph of system 'sys' is represented as an m file with heading:

```
function [bonds,components,n_ports] = sys_abg
```

The three outputs of this function are:

- bonds
- components
- n_ports

bonds is a matrix with

- one row for each bond
- the first column contains the arrow-orientated (see Section 6.5.3.1 [Arrow-orientated causality], page 36) causality of the *effort* variable.
- the second column contains the arrow-orientated (see Section 6.5.3.1 [Arrow-orientated causality], page 36) causality of the *flow* variable.

components is a matrix with

- one row for each component
- one column for each bond impinging on the component. The *magnitude* of each entry corresponds to the bond number (the appropriate row index of‘ bonds’); the sign is positive if the bond arrow points into the component and negative otherwise.

n_ports is the number of ports associated with the system – i.e. the number of Named SS components (see Section 6.5.1.9 [Named SS components], page 31).

6.5.3.1 Arrow-orientated causality

The arrow-orientated causality convention assigns -1, 0 or 1 to both the effort and flow (see Section 1.4 [Variables], page 3) sides of a bond to represent the causal stroke (see Section 6.5.1.3 [strokes], page 29) as follows:

`0` if there is no causality set.

`1` if the causal stroke is at the arrow end of the bond.

`-1` if the causal stroke is at the other end of the bond.

see Section 6.5.3.2 [Component-orientated causality], page 36.

6.5.3.2 Component-orientated causality

The component-orientated causality convention assigns -1, 0 or 1 to both the effort and flow (see Section 1.4 [Variables], page 3) sides of a bond to represent the causal stroke (see Section 6.5.1.3 [strokes], page 29) as follows:

`0` if there is no causality set.

`1` if the causal stroke is at the component end of the bond.

`-1` if the causal stroke is at the other end of the bond.

see Section 6.5.3.1 [Arrow-orientated causality], page 36.

6.5.3.3 Transformation rbg2abg_m

This transformation takes the raw bond graph and, by doing some geometrical computation, determines the topology of the bond graph – ie what is close to what.

6.5.4 Language tex (abg.tex)

For the purpose of producing a report (see Section 6.17 [Report], page 63), **MTT** generates a LaTeX (see Section 10.5 [LaTeX], page 83) file describing the bond graph and its subsystems. Additional information may be supplied using the description representation (see Section 8.2.2 [Detailed], page 77).

6.6 Stripped acausal bond graph (sabg)

The stripped acausal bond graph is the acausal bond graph representation (see Section 6.4 [Acausal bond graph (abg)], page 27) without the artwork (see Section 6.5.1.15 [artwork], page 33). It is useful to check for mistakes by showing precisely what is recognised by **MTT**.

6.6.1 Language fig (sabg.fig)

The stripped acausal bond graph can be generated as a fig (see Section 9.1 [Fig], page 78) file using

```
mtt syst sabg fig
```

6.6.2 Stripped acausal bond graph (view)

This representation has the standard text view (see Section 10.1 [Views], page 79).

6.7 Labels (lbl)

Bond graph components have optional labels. These provide pointers to further information relating to the component; this avoids clutter on the bond graph.

The label file contains the following non-blank lines (blank lines are ignored)

- Summary - lines beginning with #SUMMARY
- Description - lines beginning with #DESCRIPTION
- Alias - lines beginning with #ALIAS
- Comments - lines beginning with #
- Labels - other non-blank lines

Note, for compatability with old versions, % may be used in place of #; but the use of % is deprecated. Each label contains three fields (in the following order) separated by white space and on one line:

1. The component name see Section 6.7.3 [Component names], page 39. This must be a valid name (see Section 6.5.1.16 [Valid names], page 33.
2. The component constitutive relationship see Section 6.7.4 [Component constitutive relationship], page 39
3. The component arguments see Section 6.7.5 [Component arguments], page 39

Not each component see Section 6.5.1.4 [components], page 29 needs a label, only those which are explicitly labeled on the Bond Graph see Section 6.4 [Acausal bond graph (abg)], page 27. **MTT** checks whether all components labelled on the bond graph have labels and vice versa.

If no lbl file exists, **MTT** will create a valid one for you; including a default set of arguments and crs for both simplae and compound components.

If wish to create one to edit yourself, type

```
mtt system_name lbl txt
```

An example lbl file (for the RC system is):

```
%% Label file for system RC (RC_lbl.txt)
%SUMMARY RC
%DESCRIPTION <Detailed description here>
% Port aliases
%ALIAS  in      in
%ALIAS  out     out

% Argument aliases
```

```
%ALIAS  $1      c
%ALIAS  $2      r

%% Each line should be of one of the following forms:
%            a comment (ie starting with %)
%            component-name     cr_name arg1,arg2,..argn
%            blank

% ---- Component labels ----

% Component type C
        c               lin     effort,c

% Component type R
        r               lin     flow,r

% Component type SS
        [in]    SS              external,external
        [out]   SS              external,external
```

The old-style lbl files (see Section 6.7.11 [Old-style labels (lbl)], page 45) are NO LONGER supported – you are encouraged to convert them ASAP.

6.7.1 SS component labels

In addition to the label there are two information fields, see Section 6.7 [Labels (lbl)], page 37. The first must be 'SS', the second contains two information fields of the form info_field_1,info_field_2.

These two information fields correspond to the effort and flow variables of the of the SS components as follows

`info_field_1`
: effort

`info_field_2`
: flow

Each of these two fields contains one of the following *attributes*:

`external` indicates that the corresponding variable is a system input or output

`internal` indicates that the variable does not appear as a system output; it is an error to label an input in this way.

`a number` the value of the input; or the value of the (imposed) output

`a symbol` the symbolic value of the input; or the value of the (imposed) output

`unknown` used for the SS method of solving algebraic loops. This indicates that the corresponding system input (SS output) is to be chosen to set the corresponding system output (SS input) to zero.

`zero` used for the SS method of solving algebraic loops. This indicates that the corresponding system output (SS input) is to be set to zero using the variable indicted by the corresponding 'unknown' label.

Some examples are:

```
%% ss1 is both a source and sensor
ss1     SS              external,external
%% ss1 acts as a flow sensor - it imposes zero effort.
ss2     SS              0,external
```

6.7.2 Other component labels

In addition to the label there are two information fields, see Section 6.7 [Labels (lbl)], page 37. They correspond to the constitutive relationship (see see Section 1.6.2 [Constitutive relationship], page 5 and arguments of the component as follows

`info_field_1`
: constitutive relationship

`info_field_2`
: parameters

Some examples are:

```
%Armature resistance
r_a     lin     effort,r_a

%Gearbox ratio
n       lin     effort,n
```

MTT supports parameter-passing to (see Section 6.7.10 [Parameter passing], page 44) subsystems.

6.7.3 Component names

The component name field must contain a valid name (see Section 6.5.1.16 [Valid names], page 33 corresponding to the name (the bit after the :) of each named component (see Section 6.5.1.4 [components], page 29) on the bond graph (see Section 6.4 [Acausal bond graph (abg)], page 27).

6.7.4 Component constitutive relationship

The constitutive relationship field contains the name of a constitutive relationship for the component. There are three sorts of constitutive relationship recognised by **MTT**:

1. A generic constitutive relationship such as *lin* (the generic linear constitutive relationship.
2. A local constitutive relationship with the same name as the component type
3. The *SS* constitutive relationship reserved for *SS* components. All labels for *SS* components must contain SS in this field.

6.7.5 Component arguments

6.7.6 Parameter declarations

It is sometimes useful to use parameters (in addition to those implied by the Component arguments see Section 6.7.5 [Component arguments], page 39) to compute values in, for example the numpar file. These can be declared in the label file; for examples , the two parameters par1 and par 2 can be declared as:

```
#PAR par1
#PAR par2
```

On the other hand, some CR arguments (eg foo and bar) may not correspond to parameters. These can be excluded from the sympar list using the NOTPAR declaration

```
#NOTPAR foo
#NOTPAR bar
```

For comapability with old code, VAR may be used in place of PAR, but this usage is deprecated.

6.7.7 Units declarations

The units and domains of ports (see Section 1.6.1 [Ports], page 4) are declared as:

```
#UNITS Port_name domain effort_units flow_units
```

where "Port_name" is the name of the port, domain is one of:

`electrical`
: the electrical domain

`translational`
: the translational mechanical domain

`rotational`
: the rotational mechanical domain

`fluid` the fluid domain

`thermal` the thermal domain

and effort_units and flow_units are corresponding units for the effort and the flow.

Allowed units are those defined in the **units** package.

MTT checks that units are

- defined consistently with the domain
- the same for connected ports when both ports have defined units.

No checks are done if one or both ends of a bond are not connected to a port with defined units.

The word “none” can be specified in place of a unit to prevent **MTT** from checking the corresponding effort or flow while still checking the units of the other variable, if it is specified. This can be used to force checking of either the effort or flow on signal bonds or in pseudo-bond graphs where the domain of the two variables may not be identical.

6.7.8 Interface Control Definition

It is sometimes useful to be able to automatically generate a set of assignments mapping **MTT** inputs and outputs to an external interface definition. This can be achieved with use of the *#ICD* directive.

```
#ICD    PressureSensor PUMP1_PRESSURE_SENSOR,Pa;null,none
#ICD    Electrical PUMP1_VOLTAGE,volt;PUMP1_CURRENT,amp

% Component type De
PressureSensor SS      external

% Component type SS
Electrical SS external,external
```

The ICD directive consists of 3 whitespace delimited fields:

1. [%|#]ICD
2. component name
3. Four comma (,) or semi-colon (;) delimited fields:
   1. name of effort parameter
   2. unit of effort parameter
   3. name of flow parameter
   4. unit of flow parameter

If no parameter name is required, a value of "null" should be used. If the parameter does not have any units, a value of "none" should be used.

ICD parameters may be aliased see Section 6.7.9 [Aliases], page 42 in the same way as normal parameters, thus it is possible to define some or all of the ICD in higher level components.

The command

```
mtt sys ICD txt
```

will generate a text file containing a list of mappings:

```
## Interface Control Definition for System sys
## sys_ICD.txt: Generated by MTT Thu Jul 12 21:21:21 CDT 2001

Input:  PUMP1_VOLTAGE           sys_P1_1_Electrical      Causality: Effort   Units: vo
Output: PUMP1_CURRENT           sys_P1_1_Electrical      Causality: Flow     Units: am
Output: PUMP1_PRESSURE_SENSOR   sys_P1_1_PressureSensor  Causality: Effort   Units: Pa
```

A set of assignments can be generated with the command

```
mtt sys ICD m
```

resulting in:

```
# Interface Control Definition mappings for system sys
# sys_ICD.m: Generated by MTT Thu Jul 12 21:26:56 CDT 2001

# Inputs
```

```
        mttu(1) = PUMP1_VOLTAGE;

# Outputs

        PUMP1_CURRENT                   = mtty(1);
        PUMP1_PRESSURE_SENSOR           = mtty(2);
```

A similar file will be generated by the command

```
mtt sys ICD cc
```

6.7.9 Aliases

Aliases provide a convenient mechanism for relabelling words appearing in the label file (see Section 6.7 [Labels (lbl)], page 37). There are three contexts in which the alias mechanism is used:

1. renaming ports (see Section 6.7.9.1 [Port aliases], page 42),
2. renaming parameters (see Section 6.7.9.2 [Parameter aliases], page 43) and
3. renaming components (see Section 6.7.9.4 [Component aliases], page 43).

All three mechanisms use the same form of statement within the label file

```
%ALIAS short_label       real_label
```

MTT distinguishes between the three forms as follows:

- Parameter aliases: 'short_label' starts with a '$'
- Component aliases: 'real_label' contains the directory separator '/'
- Port aliases: neither of the above

6.7.9.1 Port aliases

Aliases provide a way of refering to (see Section 6.5.1.11 [Port labels], page 31) or vector port labels (see Section 6.5.1.12 [Vector port labels], page 32) on the bond graph using a short-hand notation. With in a component label file (see Section 6.7 [Labels (lbl)], page 37) statements of the following forms can occur

```
%ALIAS short_label       real_label
```

When the component is used within another component, the short_lable may be used in place of the real_label. More than one alias per label can be used, for example

```
%ALIAS short_label_1       real_label
%ALIAS short_label_2       real_label
%ALIAS short_label_3       real_label
```

The port can then be refered to in four ways: as real_label, short_label_1, short_label_2 or short_label_3. An alternative notation for the ALIAS statement in this case is

```
%ALIAS short_label_1|short_label_2|short_label_3       real_label
```

The alias feature is particularly powerful in conjunction with vector port labels (see Section 6.5.1.12 [Vector port labels], page 32) and the port label default (see Section 6.5.1.13 [Port label defaults], page 32) mechanisms. For example, a component with 5 ports appearing in the lbl file as:

```
[Hydraulic_in]  external        external
[Hydraulic_out] external        external
[Power_Shaft]           external        external
[Thermal_in]    external        external
[Thermal_out]   external        external
```

together with the following statements in the label file:

```
%ALIAS  in              Thermal_in,Hyydraulic_in
%ALIAS  out             Thermal_out,Hydraulic_out
%ALIAS  shaft|power     Power_Shaft
```

can appear in the bond graph containing that component with one bond labeled either [shaft] or [power] or [Power_Shaft], one unlabeled vector bond pointing in and one unlabeled vector bond pointing out.

6.7.9.2 Parameter aliases

Parameter aliases are of the form

```
%ALIAS $n       actual parameter
```

where n is an integer (unique within the label file). For example

```
%ALIAS  $1              c_v
%ALIAS  $2              density,ideal_gas,r
%ALIAS  $3              alpha
%ALIAS  $4              flow,k_p
```

Assigns four symbolic parameters to the corresponding strings These four parameters (`$1`–`$4`) can then be used for parameter passing(see Section 6.7.10 [Parameter passing], page 44).

6.7.9.3 CR aliases

CR aliases are of the form

```
%ALIAS $an       actual parameter
```

where n is an integer (unique within the label file). For example

```
%ALIAS  $a1  lin
```

assigns the symbolic parameter to be lin. This parameter `$1` can then be used for passing a diofferent cr to the component (see Section 6.7.10 [Parameter passing], page 44).

6.7.9.4 Component aliases

Component aliases are of the form

```
%ALIAS Component_name   Component_location
```

An example appears in the following label file fragment

```
...
%ALIAS  wPipe   CompressibleFlow/wPipe
%ALIAS  Poly    CompressibleFlow/Poly
....
```

The two components 'wPipe' and 'Poly' are both to be found within the library 'Compressible flow' and the respective subdirectories. This follows the **MTT** convention that compound components (see Section 6.5.1.8 [Compound components], page 31) live within a directory of the same name.

6.7.10 Parameter passing

MTT supports parameter-passing to subsystems within label files (see Section 6.7 [Labels (lbl)], page 37). Within a subsystem, explicit constitutive relationships and parameters (or groups thereof) can be replaced by postitional parameters such as `$1`, `$2` etc. Although this can be done directly, it is recommended that this is done via the alias mechanism (see Section 6.7.9.2 [Parameter aliases], page 43).

In a subsystem `$i`, is replaced by the ith field of a colon `;` separated field in the calling label file. This field may include commas `,` and the four arithmetic operators `+`, `-`, `*` and `/`.

For example, consider the following example label file fragment (associated with a component called Pump:

```
...

%ALIAS  $1              c_v
%ALIAS  $2              density,ideal_gas,r
%ALIAS  $3              alpha
%ALIAS  $4              flow,k_p

%ALIAS  wPipe   CompressibleFlow/wPipe
%ALIAS  Poly    CompressibleFlow/Poly

% Component type wPipe
        pipe    none                    c_v;density,ideal_gas,r

% Component type Poly
        poly            Poly            alpha
```

The 4 parameters `$1`, `$2`, `$3`, and `$4` can be passed from a higher level component as in the following label file fragment:

```
% Component type Pump
        comp            none            c_v;rho,ideal_gas,r;alpha;effort,k_c
        turb            none            c_v;rho,ideal_gas,r;alpha;effort,k_t
```

Thus in component 'comp':

- `$1` is replaced by c_v
- `$2` is replaced by rho,ideal_gas
- `$3` is replaced by alpha
- `$4` is replaced by effort,k_c

whereas in component 'turb' the first three parameters are the same but

- `$4` is replaced by effort,k_t

6.7.11 Old-style labels (lbl)

Old syle labels (mtt version 2.x) are supported by mtt version 3.x. However, you are advised to use the new form (see Section 6.7 [Labels (lbl)], page 37).

Each line of the `_label.txt` file is of one of three forms:

1. Contains three fields (separated by white space) of the form

```
label   field_1   field_2
```

2. Blank
3. Preceded by %

Only the first is noticed by **MTT**; the second and third are for providing helpful commenting.

The role of the two information fields depends on the component with the corresponding label. In particular the classes of components are:

- SS components, see Section 6.5.1.6 [SS components], page 30.
- Other components, see Section 6.5.1.4 [components], page 29.

Named SS component, see Section 6.5.1.9 [Named SS components], page 31 never have labels.

6.7.11.1 SS component labels (old-style)

In addition to the label there are two information fields, see Section 6.7 [Labels (lbl)], page 37. They correspond to the effort and flow of the components as follows

`info_field_1`
: effort

`info_field_2`
: flow

Each of these two fields contains one of the following *attributes*:

: external indicates that the corresponding variable is a system input or output

`internal`
: indicates that the variable does not appear as a system output; it is an error to label an input in this way.

`a number`
: the value of the input; or the value of the (imposed) output

`a symbol`
: the symbolic value of the input; or the value of the (imposed) output

`unknown`
: used for the SS method of solving algebraic loops. This indicates that the corresponding system input (SS output) is to be chosen to set the corresponding system output (SS input) to zero.

`zero`
: used for the SS method of solving algebraic loops. This indicates that the corresponding system output (SS input) is to be set to zero using the variable indicted by the corresponding 'unknown' label.

Some examples are:

```
%Label  field1          field2
ss1     external        external
ss2     0               external
```

6.7.11.2 Other component labels (old-style)

In addition to the label there are two information fields, see Section 6.7 [Labels (lbl)], page 37. They correspond to the constitutive relationship (see see Section 1.6.2 [Constitutive relationship], page 5 and arguments of the component as follows

`info_field_1`
: constitutive relationship

`info_field_2`
: parameters

Some examples are:

```
%Armature resistance
r_a     lin     effort,r_a

%Gearbox ratio
n       lin     effort,n
```

MTT supports parameter-passing to (see Section 6.7.11.3 [Parameter passing (old-style)], page 46) subsystems.

6.7.11.3 Parameter passing (old-style)

MTT supports parameter-passing to (see Section 6.7.11.3 [Parameter passing (old-style)], page 46) subsystems within label files (see Section 6.7 [Labels (lbl)], page 37). Within a subsystem, explicit constitutive relationships and parameters (or groups thereof) can be replaced by `$1`, `$2`, etc.

In a subsystem `$i`, is replaced by the ith field of a colon `;` separated field in the calling label file. This field may include commas `,`.

For example subsystem ROD contains the following lines in the label file:

```
%DESCRIPTION    Parameter 1:    length from end 1 to mass centre
%DESCRIPTION    Parameter 2:    length from end 2 to mass centre
%DESCRIPTION    Parameter 3:    inertia about mass centre
%DESCRIPTION    Parameter 4:    mass
%DESCRIPTION    See Section 10.2 of "Metamodelling"


%Inertias
J       lin     flow,$3
m_x     lin     flow,$4
m_y     lin     flow,$4

%Integrate angular velocity to get angle
th

%Modulated transformers
s1      lsin    flow,$1
s2      lsin    flow,$2
```

```
c1      lcos    flow,$1
c2      lcos    flow,$2
```

This can be used in a higher-level lbl (see Section 6.7 [Labels (lbl)], page 37) file as:

```
%SUMMARY Pendulum example from Section 10.3 of "Metamodelling"

%Rod parameters
rod     none    l;l;j;m
```

6.7.12 Language tex (desc.tex)

This file may contain any LaTeX compatible commands. Any mathematics should conform to the AMSmath package.

6.8 Structure (struc)

The causal bond graph implies a set of equations describing the system. The Structure (struc) representation describes the structure of these equations in terms of the input, outputs, states and non-states of the system.

6.8.1 Language txt (struc.txt)

This text tile contains a description of the system structure (see Section 6.8 [Structure (struc)], page 47 with 5 tab-separated columns containing the following information:

`type` input, output state or nonstate

index an integer corresponding to the array index

component name the name of the component corresponding to the variable

`system name`
the name of the system containing the component

`repetition`
an integer corresponding to the repetition of a repeated subsystem.

An example of such a file (corresponding to rc) (see Section 3.1 [Quick start], page 13) is:

```
input           1       e1      rc      1
output          1       e2      rc      1
state           1       c       rc      1
```

6.8.2 Language tex (struc.tex)

This LaTeX (see Section 10.5 [LaTeX], page 83) file contains a description of the system structure (see Section 6.8 [Structure (struc)], page 47 in `longtable` format. It is a useful item to include in a report(see Section 6.17 [Report], page 63).

6.8.3 Language tex (view)

This representation has the standard text view (see Section 10.1 [Views], page 79).

6.9 Constitutive relationship (cr)

The constitutive relationship (see Section 1.6.2 [Constitutive relationship], page 5) of a simple component (see Section 6.5.1.5 [Simple components], page 30 is defined in the symbolic algebra language Reduce (see Section 9.3 [Reduce], page 78). The constitutive relationship of a compound components (see Section 6.5.1.8 [Compound components], page 31) is implied by the constitutive relationships of its constituent components.

6.9.1 Predefined constitutive relationships

Some common cr's are predefined by MTT; these are:

`lin` a linear constitutive relationship

`exotherm` an exothermic reaction

6.9.1.1 lin

The constitutive relationship `lin` is predefined for the following components.

`R` (one-port) R component

`TF` transformer

`GY` gyrator

`MTF` modulated transformer

`MGY` modulated gyrator

`FMR` flow-modulated resistor

Lin takes two arguments in the form causality,gain

`causality`
the causality (effort or flow) of the *input* to the constitutive relationship

`gain` the gain of the component when the input causality is as specified in the first argument.

For example the arguments

```
flow,r
```

given to an R component corresponds to

```
e = rf
```

if if the input causality is flow or

```
f = e/r
```

if if the input causality is effort.

6.9.1.2 exotherm

6.9.2 DIY constitutive relationships

You can write your own constitutive relationships using Reduce (see Section 9.3 [Reduce], page 78). This requires some understanding as to how **MTT** represent the elementary system equations (see Section 6.12 [Elementary system equations], page 57). Looking at the predefined constitutive relationships is a good way to get started (see Section 11.5 [File structure], page 86).

6.9.3 Unresolved constitutive relationships

Consider the following CR file.

```
FOR ALL rho,g,vol,h,topt,bott,flowin,press
LET tktf2(rho,g,vol,h,topt,bott,effort,2,press,effort,1)
        = tank(rho,g,vol,h,topt,bott,press);
```

Assuming that 'tank' is not defined in a reduce file, MTT will leave it unresolved when generating m or c code.

The resulting function can then be expressed as octave (see Section 6.9.4 [Unresolved constitutive relationships - Octave], page 49) or c++ code as (see Section 6.9.5 [Unresolved constitutive relationships - c++], page 49) appropriate.

6.9.4 Unresolved constitutive relationships - Octave

Following the example of the previous section, the unresolved CR 'tank' can be expressed as an Octave m-file. For example:

```
function p = tank (rho,g,vol,h,topt,bott,press)

  ## usage:  p = tank (vol,h,topt,bott,press)
  ##
  ##

   val = press; zt = topt; zb = bott;
   zval = 0.5*(abs(zb+(zt-zb)*val-h)+(zb+(zt-zb)*val-h));

   p = rho*g*zval + 0.5*(1+tanh((press-0.98)*500))*100000;

endfunction
```

This will be automatically loaded into octave.

6.9.5 Unresolved constitutive relationships - c++

Following the example of the previous section, the unresolved CR 'tank' can be expressed in c++ code. For example:

```
inline double tank(const double rho,
   const double g,
   const double vol,
   const double h,
   const double topt,
   const double bott,
   const double press)


  /*  ## usage:  p = tank (vol,h,topt,bott,press)
    ##
    ##
  */
  double p, val, zval, zt, zb;
```

```
val = press;
zt = topt;
zb = bott;
zval = 0.5 * (abs(zb + (zt - zb) * val - h) + zb + (zt - zb) * val - h);

p = rho * g * zval + 0.5 * (1 + tanh((press - 0.98) * 500)) * 100000L;

return p;
```

To make sure that this is used in system 'model', the model_cr.h file must be as follows:

```
// CR headers for system model
#include "tank.c"
```

6.10 Parameters

In general, lbl (see Section 6.7 [Labels (lbl)], page 37) files contain symbolic parameters. **MTT** provides three ways of substituting for these parameters:

- symbolic substitution
- symbolic substitution for simplification of displayed equations
- numeric

6.10.1 Symbolic parameters (subs.r)

This file contains reduce statements to symbolically change the expressions describing the system. For example, a useful set of trig substitutions is:

```
LET cos(~x)*cos(~y) = (cos(x+y)+cos(x-y))/2;
LET cos(~x)*sin(~y) = (sin(x+y)-sin(x-y))/2;
LET sin(~x)*sin(~y) = (cos(x-y)-cos(x+y))/2;
LET cos(~x)^2       = (1+cos(2*x))/2;
LET sin(~x)^2       = (1-cos(2*x));
```

6.10.2 Symbolic parameters for simplification (simp.r)

This file contains reduce statements to symbolically change the expressions describing the system. Unlike the subs.r file (see Section 6.10.1 [Symbolic parameters (subs.r)], page 50) it does not affect all system transformations; only those converting to LaTeX form.

6.10.3 Numeric parameters (numpar)

When computing time and frequency responses; or when evaluating functions in Octave (see Section 10.4 [Octave], page 79); symbolic parameters need numerical instantiations.

The numpar representation provides the relevant *numerical* information. It comes in a number of languages:

`txt` a textual description of the parameter values – this is the defining representation (see Section 6.2 [Defining representations], page 27).

`m` readable by `octave` a high-level interactive language for numerical computation – translated by **mtt** from the txt version.

`c` readable by `gcc` a c compiler – translated by **mtt** from the txt version.

6.10.3.1 Text form (numpar.txt)

This is the textual form of the numerical parameters representation (see Section 6.10.3 [Numeric parameters (numpar)], page 50). Lines are either

`assignment statements`
: variable = value

`comments` lines beginning with #

`commented assignment statements`
: variable = value # comments

An example file is:

```
# Numerical parameter file (rc_numpar.txt)
# Generated by MTT at Mon Jun 16 15:10:17 BST 1997

# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# %% Version control history
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
# %% $Id: mtt.texi,v 1.30 2005/03/15 12:04:15 gawthrop Exp $
# %% $Log: mtt.texi,v $
# %% Revision 1.30  2005/03/15 12:04:15  gawthrop
# %% New labg.fig rep - pretty LaTeX figures.
# %%
# %% Revision 1.29  2005/01/19 09:45:15  geraint
# %% Fixed whitespace.
# %%
# %% Revision 1.28  2005/01/06 16:03:55  geraint
# %% Updated options list.
# %%
# %% Revision 1.27  2005/01/06 14:42:33  geraint
# %% Fixed explicit/implicit descriptions of forward/backward Euler.
# %% Replaced -c examples with -cc.
# %%
# %% Revision 1.26  2005/01/06 12:28:36  geraint
# %% Minor typos.
# %%
# %% Revision 1.25  2004/08/27 20:12:34  geraint
# %% Added note about "none" as an option for units.
# %%
# %% Revision 1.24  2004/08/13 01:39:24  geraint
# %% Fixed variable names in diy makefile example (again)
# %%
# %% Revision 1.23  2004/08/11 08:24:28  geraint
# %% Replaced "gnuplot view" with "odeso gnuplot" in examples.
# %%
# %% Revision 1.21  2004/07/23 11:05:26  geraint
```

```
# %% Updated Reduce URL: http://www.reduce-algebra.com
# %%
# %% Revision 1.20  2003/10/23 18:14:43  geraint
# %% Added subsection with script for model-specific options.
# %% Fixed typo in url to mtt.sf.net.
# %%
# %% Revision 1.19  2003/10/10 22:22:18  geraint
# %% typo.
# %%
# %% Revision 1.18  2003/09/07 20:41:19  geraint
# %% *** empty log message ***
# %%
# %% Revision 1.17  2003/08/19 14:20:38  gawthrop
# %% Version 5.0 of MTT
# %% Remove xref errors (spurious spaces)
# %%
# %% Revision 1.16  2003/08/19 14:11:23  gawthrop
# %% Links to legal stuff
# %%
# %% Revision 1.15  2003/08/19 14:01:45  gawthrop
# %% Added legal appendices
# %%
# %% Revision 1.14  2003/08/06 14:50:56  gawthrop
# %% Describe the alias mechanism for invoking mtt options
# %%
# %% Revision 1.13  2002/12/13 10:07:07  gawthrop
# %% Added example in sh section of DIY reps
# %%
# %% Revision 1.12  2002/09/19 08:09:31  gawthrop
# %% Updated documentation documentation
# %%
# %% Revision 1.11  2002/08/20 15:51:17  gawthrop
# %% Update to work with ident DIY rep
# %%
# %% Revision 1.10  2002/07/22 10:45:22  geraint
# %% Fixed gnuplot rep so that it correctly re-runs the simulation if input files have
# %%
# %% Revision 1.9  2002/07/05 13:29:34  geraint
# %% Added notes about generating dynamically linked functions for Octave and Matlab.
# %%
# %% Revision 1.8  2002/07/04 21:34:12  geraint
# %% Updated gnuplot view description to describe Tcl/Tk interface instead of obsolete
# %%
# %% Revision 1.7  2002/04/23 09:51:54  gawthrop
# %% Changed incorrect statement about searching for components.
# %%
# %% Revision 1.6  2001/10/15 14:29:50  gawthrop
```

```
# %% Added documentaton on  [1:N] style port labels
# %%
# %% Revision 1.5  2001/07/23 03:35:29  geraint
# %% Updated file structure (mtt/bin).
# %%
# %% Revision 1.4  2001/07/23 03:25:02  geraint
# %% Added notes on -ae hybrd, rk4, ode2odes.cc, .oct dependencies.
# %%
# %% Revision 1.3  2001/07/13 03:02:38  geraint
# %% Added notes on #ICD, gnuplot.txt and odes.sg rep.
# %%
# %% Revision 1.2  2001/07/03 22:59:10  gawthrop
# %% Fixed problems with argument passing for CRs
# %%
# %% Revision 1.1  2001/06/04 08:18:52  gawthrop
# %% Putting documentation under CVS
# %%
# %% Revision 1.66  2000/12/05 14:20:55  peterg
# %% Added the c++  anf m CR info.
# %%
# %% Revision 1.65  2000/11/27 15:36:15  peterg
# %% NOPAR --> NOTPAR
# %%
# %% Revision 1.64  2000/11/16 14:22:48  peterg
# %% added UNITS declaration
# %%
# %% Revision 1.63  2000/11/03 14:41:08  peterg
# %% Added PAR and NOTPAR stuff
# %%
# %% Revision 1.62  2000/10/17 17:53:34  peterg
# %% Added some simulation details
# %%
# %% Revision 1.61  2000/09/14 17:13:06  peterg
# %% New options table
# %%
# %% Revision 1.60  2000/09/14 17:09:20  peterg
# %% Tidied up valid name sections
# %% Tidied up defining represnetations table
# %% Verion 4.6
# %%
# %% Revision 1.59  2000/08/30 13:09:00  peterg
# %% Updated option table
# %%
# %% Revision 1.58  2000/08/01 13:30:19  peterg
# %% Version 4.4
# %% updated STEPFACTOR info
# %% describes octave and OCST interfaces
```

```
# %%
# %% Revision 1.57  2000/07/20 07:55:44  peterg
# %% Version 4.3
# %%
# %% Revision 1.56  2000/05/19 17:49:17  peterg
# %% Extended the user defined representation section -- new nppp rep.
# %%
# %% Revision 1.55  2000/03/16 13:53:31  peterg
# %% Correct date
# %%
# %% Revision 1.54  2000/03/15 21:22:57  peterg
# %% Updated to 4.1 -- old style SS no longer supported
# %%
# %% Revision 1.53  1999/12/22 05:33:10  peterg
# %% Updated for 4.0
# %%
# %% Revision 1.52  1999/11/23 00:25:11  peterg
# %% Added the sensitivity reps
# %%
# %% Revision 1.51  1999/11/16 04:43:47  peterg
# %% Added start of sensitivity section
# %%
# %% Revision 1.50  1999/11/16 00:30:35  peterg
# %% Updated simulation section
# %% Added vector components
# %%
# %% Revision 1.49  1999/07/20 23:44:58  peterg
# %% V 3.8
# %%
# %% Revision 1.48  1999/07/19 03:08:33  peterg
# %% Added documentation for (new) SS lbl fields
# %%
# %% Revision 1.47  1999/03/09 01:42:22  peterg
# %% Rearranged the User interface section
# %%
# %% Revision 1.46  1999/03/09 01:18:01  peterg
# %% Updated for 3.5 including xmtt
# %%
# %% Revision 1.45  1999/03/03 02:39:26  peterg
# %% Minor updates
# %%
# %% Revision 1.44  1999/02/17 06:52:14  peterg
# %% New level formula dor artwork
# %%
# %% Revision 1.43  1998/11/25 16:49:24  peterg
# %% Put in subs.r documentation (was called params.r)
# %%
```

```
# %% Revision 1.42  1998/11/24 12:24:59  peterg
# %% Added section on simulation output
# %% Version 3.4
# %%
# %% Revision 1.41  1998/09/02 12:04:15  peterg
# %% Version 3.2
# %%
# %% Revision 1.40  1998/08/27 08:36:39  peterg
# %% Removed in. methods except Euler anf implicit
# %%
# %% Revision 1.39  1998/08/18 10:44:28  peterg
# %% Typo
# %%
# %% Revision 1.38  1998/08/18 09:16:38  peterg
# %% Version 3.1
# %%
# %% Revision 1.37  1998/08/17 16:14:30  peterg
# %% Version 3.1 - includes documentation on METHOD=IMPLICIT
# %%
# %% Revision 1.36  1998/07/30 17:33:15  peterg
# %% VERSION 3.0
# %%
# %% Revision 1.35  1998/07/22 11:00:53  peterg
# %% Correct date!
# %%
# %% Revision 1.34  1998/07/22 11:00:13  peterg
# %% Version to BAe
# %%
# %% Revision 1.33  1998/07/17 19:32:19  peterg
# %% Added more about aliases
# %%
# %% Revision 1.32  1998/07/05 14:21:56  peterg
# %% Further additions (Carlisle-Glasgow)
# %%
# %% Revision 1.31  1998/07/04 11:35:57  peterg
# %% Strarted new lbl description
# %%
# %% Revision 1.30  1998/07/02 18:39:20  peterg
# %% Started 3.0
# %% Added alias and default sections.
# %%
# %% Revision 1.29  1998/05/19 19:46:58  peterg
# %% Added the odess description
# %%
# %% Revision 1.28  1998/05/14 09:17:22  peterg
# %% Added METHOD variable to the simpar file
# %%
```

```
# %% Revision 1.27  1998/05/13 10:03:09  peterg
# %% Added unknown/zero SS label documentation.
# %%
# %% Revision 1.26  1998/04/29 15:12:46  peterg
# %% Version 2.9.
# %%
# %% Revision 1.25  1998/04/12 17:00:26  peterg
# %% Added new port features: coerced direction and top-level behaviour.
# %%
# %% Revision 1.24  1998/04/05 18:27:20  peterg
# %% This was the 2.6 version
# %%
# Revision 1.23  1997/08/24  11:17:51  peterg
# This is the released  version 2.5
#
# %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

# Parameters
c =     1.0; # Default value
r =     1.0; # Default value
# Initial states
x(1) =  0.0; # Initial state for rc (c)
```

As usual, **MTT** provides a default text file to be edited by the user (see Section 10.3 [Text editors], page 79).

6.11 Causal bond graph (cbg)

The causal bond graph is the causally complete version of the Acausal bond graph (see Section 6.4 [Acausal bond graph (abg)], page 27).

To create the causal bond graph of system 'sys' in language fig type:

```
mtt sys cbg fig
```

To create the causal bond graph of system 'sys' in language m type:

```
mtt sys cbg m
```

To view the causal bond graph of system 'sys' type:

```
mtt sys cbg view
```

6.11.1 Language fig (cbg.fig)

The fig file is created by **MTT**. It is identical to the corresponding acausal representation (see Section 6.5.1 [Language fig (abg.fig)], page 28) except that

- the new causal strokes are added (using a double thickness line in blue)
- components that are undercausal are bold and green
- components that are overcausal are bold and red

6.11.2 Language m (cbg.m)

The causal bond graph of system 'sys' is represented as an m file with heading:

```
function [cbonds,status] = sys_cbg
```

The two outputs of this function are:

- cbonds
- status

cbonds is a matrix with

- one row for each bond
- the first column contains the arrow-orientated (see Section 6.5.3.1 [Arrow-orientated causality], page 36) causality of the *effort* variable.
- the second column contains the arrow-orientated (see Section 6.5.3.1 [Arrow-orientated causality], page 36) causality of the *flow* variable.

status is a matrix with

- one row for each component
- the first column contains 1 if the component is overcausal; 0 if the component is causally complete and -1 if the component is undercausal.

A successful model would therefore have all zeros in the status matrix.

6.11.2.1 Transformation abg2cbg_m

This transformation takes the acausal bond graph as an m file (see Section 6.5.3 [Language m (abg.m)], page 35) and transforms it into a causal bond graph in m-file format (see Section 6.11.2 [Language m (cbg.m)], page 57).

It is based on the m-function abg2cbg.m which iteratively tries to complete causality whilst recursively searching the bond graph structure. If causality is incomplete, it picks the first acausal dynamic (C or I) component, asserts integral causality, and tries again.

This is essentially the sequential causality assignment procedure of Karnopp and Rosenberg.

The transformation informs the user of the final status in terms of the percentage of causally complete components; a successful model will yield 100% here.

6.12 Elementary system equations (ese)

The elementary system equations are a complete set of assignment statements describing the dynamic system corresponding to the bond graph. They are in the Reduce (see Section 9.3 [Reduce], page 78) language.

Because these are based on a causally complete system, these assignment statements are directly soluble by substitution.

Unlike early versions of **MTT**, **MTT** does *not* sort the equations in order of solution, but rather leaves them sorted by component and subsystem.

These are not supposed to be read by the user, so there is no view facility as such. However, you may read these with your favourite text editor and, to this end, helpful comment lines have been added.

Wherever components have an explicit constitutive relationship, the corresponding RHS of the equation has a standard form.

```
cr(arguments,out_causality,outport,
        input_1, causality_1, port_1,
        ....
        input_i, causality_i, port_i,
        ....
        input_n, causality_n, port_n
        );
```

where the symbols have the following meaning

`arguments`
: the constitutive relationship arguments

`out_causality`
: the causality (effort or flow) of the output variable (see Section 1.4 [Variables], page 3)

`outport`
: the number (integer) of the output port of the system

`input_i`
: the ith input to the component

`causality_i`
: the causality (effort or flow) of the ith input variable (see Section 1.4 [Variables], page 3)

`port_i`
: the number (integer) of the ith input port of the system

An example for a resistor with linear constitutive relationship is:

```
rc_1_bond4_flow := lin(flow,r,flow,1,
        rc_1_bond4_effort,effort,1
        );
```

6.12.0.1 Transformation cbg2ese_m2r

This transformation takes the causal bond graph as an m file (see Section 6.11.2 [Language m (cbg.m)], page 57) and transforms it into elementary system equations in Reduce (see Section 9.3 [Reduce], page 78) form.

It is based on the m-function cbg2ese.m which iteratively traverses the causal bond graph writing equations as it goes.

It also writes out the system structure as the file `sys_def.r`.

6.13 Differential-Algebraic Equations (dae)

The system differential algebraic equations describe the system dynamics together together with any algebraic constraints.

They are generated in language `lang` for system `sys` by:

```
mtt sys dae lang
```

Valid languages are:

`r`
: reduce (see Section 9.3 [Reduce], page 78).

`m` m (see Section 9.2 [m], page 78).

`view` reduce (see Section 10.1 [Views], page 79).

There are five sets of variables describing the system:

`x` the system states (corresponding to C and I components with integral causality.

`z` the system nonstates (corresponding to C and I components with derivative causality.

`u` the system inputs (corresponding to SS components with external attribute).

`ui` the *internal* system inputs (corresponding to SS components with internal attribute) used to solve algebraic loops (see Section 1.7 [Algebraic loops], page 5).

`y` the system outputs (corresponding to SS components with external attribute).

In general there are four sets of equations. The right-hand side of each is a function of x, dz/dt, u and ui and the left hand sides are:

1. the derivative of x (dx/dt)
2. z
3. w=0 (the algebraic equations)
4. y

6.13.1 Language reduce (dae.r)

The system DAEs (see Section 6.13 [Differential-Algebraic Equations], page 58) are represented in the reduce (see Section 9.3 [Reduce], page 78) language as arrays containing the algebraic expressions for the right hand sides of each set of equations. The arrays are:

`MTTx` x – the system states (corresponding to C and I components with integral causality.

`MTTz` z – the system nonstates (corresponding to C and I components with derivative causality.

`MTTu` u – the system inputs (corresponding to SS components with external attribute).

`mttv` ui – the *internal* system inputs (corresponding to SS components with internal attribute) used to solve algebraic loops (see Section 1.7 [Algebraic loops], page 5).

`MTTy` y – the system outputs (corresponding to SS components with external attribute).

6.13.1.1 Transformation ese2dae_r

This transformation (see Section 1.2 [What is a Transformation?], page 2) uses Reduce (see Section 9.3 [Reduce], page 78) to combine the elementary system equations (see Section 6.12 [Elementary system equations], page 57) with the constitutive relationships (see Section 1.6.2 [Constitutive relationship], page 5) and simplify the result.

6.13.2 Language m (dae.m)

The system DAEs (see Section 6.13 [Differential-Algebraic Equations], page 58) are represented in the m (see Section 9.2 [m], page 78) language as two m-functions of the form:

```
function resid = sys_dae(dx,x,t)
function y  = sys_dae(dx,x,t)
```

Where x is the dae *descriptor* vector and dx its time derivative; t is the time. The first function is of a form suitable for solution by DASSL; the second function can then be used to find the coresponding system output.

6.13.2.1 Transformation dae_r2m

This transformation (see Section 1.2 [What is a Transformation?], page 2) uses Reduce (see Section 9.3 [Reduce], page 78) to rewrite the elementary system equations (see Section 6.12 [Elementary system equations], page 57) in m-file format (see Section 9.2 [m], page 78) . Numerical parameters are declared as global.

6.14 Constrained-state Equations (cse)

The system constrained-state equations describe the system dynamics for a special class of systems (see the book for details). The resuting equations are of the form:

```
E(x) dx/dt = f(x,u)
y = g(x,u)
```

They typically occure where two or more states are constrained to be equal, or proportional, to each other. For example, two capacitors in parallel or two inertias connected by a stiff shaft.

They are generated in language `lang` for system `sys` by:

```
mtt sys cse lang
```

Valid languages are:

`r` reduce (see Section 9.3 [Reduce], page 78).

`m` m (see Section 9.2 [m], page 78).

`view` reduce (see Section 10.1 [Views], page 79).

There are three sets of variables describing the system:

`x` the system states (corresponding to C and I components with integral causality.

`u` the system inputs (corresponding to SS components with external attribute).

`y` the system outputs (corresponding to SS components with external attribute).

In general there are two sets of equations. The right-hand side of each is a function of x and u and the left hand sides are:

1. the derivative of x (dx/dt) y

6.14.1 Language reduce (cse.r)

The system CSEs (see Section 6.14 [Constrained-state Equations], page 60) are represented in the reduce (see Section 9.3 [Reduce], page 78) language as arrays containing the algebraic expressions for the right hand sides of each set of equations. The arrays are:

`MTTx` x – the system states (corresponding to C and I components with integral causality.

`MTTu` u – the system inputs (corresponding to SS components with external attribute).

`MTTy` y – the system outputs (corresponding to SS components with external attribute).

together with the array containing the elements of the E matrix.

6.14.1.1 Transformation dae2cse_r

This transformation (see Section 1.2 [What is a Transformation?], page 2) Reduce (see Section 9.3 [Reduce], page 78) to find various Jacobians which are combined to find the E matrix and the constrained-state equations (see Section 6.14 [Constrained-state Equations], page 60).

6.14.2 Language m (view)

This representation has the standard text view (see Section 10.1 [Views], page 79).

6.15 Ordinary Differential Equations

The system ordinary differential equations describe the system dynamics.

They are generated in language `lang` for system `sys` by:

```
mtt sys ode lang
```

Valid languages are:

`r` reduce (see Section 9.3 [Reduce], page 78).

`m` m (see Section 9.2 [m], page 78).

`view` reduce (see Section 10.1 [Views], page 79).

There are three sets of variables describing the system:

`x` the system states (corresponding to C and I components with integral causality.

`u` the system inputs (corresponding to SS components with external attribute).

`y` the system outputs (corresponding to SS components with external attribute).

In general there are two sets of equations. The right-hand side of each is a function of x and u and the left hand sides are:

1. the derivative of x (dx/dt) y

6.15.1 Language reduce (ode.r)

The system ODEs (see Section 6.15 [Ordinary Differential Equations], page 61) are represented in the reduce (see Section 9.3 [Reduce], page 78) language as arrays containing the algebraic expressions for the right hand sides of each set of equations. The arrays are:

`MTTx` x – the system states (corresponding to C and I components with integral causality.

`MTTu` u – the system inputs (corresponding to SS components with external attribute).

`MTTy` y – the system outputs (corresponding to SS components with external attribute).

6.15.1.1 Transformation cse2ode_r

This transformation (see Section 1.2 [What is a Transformation?], page 2) uses Reduce (see Section 9.3 [Reduce], page 78) to invert the E matrix of the constrained-state equations (see Section 6.14 [Constrained-state Equations], page 60) and simplify the result.

6.15.2 Language m (ode.m)

The system ODEs (see Section 6.15 [Ordinary Differential Equations], page 61) are represented in the m (see Section 9.2 [m], page 78) language as two m-functions of the form:

```
function dx = sys_ODE(x,t)
function y  = sys_ODE(dx,x,t)
```

Where x is the ODE *state* vector and dx its time derivative; t is the time. The first function is of a form suitable for solution by odesol; the second function can then be used to find the corresponding system output.

6.15.2.1 Transformation ode_r2m

This transformation (see Section 1.2 [What is a Transformation?], page 2) uses Reduce (see Section 9.3 [Reduce], page 78) to rewrite the ordinary differential equations (see Section 6.15 [Ordinary Differential Equations], page 61) in m-file format (see Section 9.2 [m], page 78) . Numerical parameters are declared as global.

6.15.3 Language m (view)

This representation has the standard text view (see Section 10.1 [Views], page 79).

6.16 Descriptor matrices (dm)

The system descriptor matrices A, B, C, D and E describe the *linearised* system dynamics in the form

```
E dx/dt = Ax + Bu
y = Cx + Du
```

They are generated in language `lang` for system `sys` by:

```
mtt sys dm lang
```

Valid languages are:

`r` reduce (see Section 9.3 [Reduce], page 78).

`m` m (see Section 9.2 [m], page 78).

`view` reduce (see Section 10.1 [Views], page 79).

6.16.1 Language reduce (dm.r)

The system descriptor matrices (see Section 6.16 [Descriptor matrices], page 62) are represented in the reduce (see Section 9.3 [Reduce], page 78) language as arrays containing the four matrices. The arrays are:

`MTTA` A

`MTTB` B

`MTTA` C

`MTTD` D

`MTTE` E

6.16.2 Language m (dm.m)

The system descriptor matrices (see Section 6.16 [Descriptor matrices], page 62) are represented in the m (see Section 9.2 [m], page 78) language as an m-function of the form:

```
function [A,B,C,D,E] = sys_dm
```

System numeric parameters (see Section 1.6.4 [Numeric parameters], page 5) are passed via global variables defined in the _numpar.m file. Thus the system descriptor matrices are typically generated in Octave (see Section 10.4 [Octave], page 79) as follows:

```
sys_numpar
[A,B,C,D,E] = sys_dm
```

Parameters can be changed from their default values by entering their values directly into Octave (see Section 10.4 [Octave], page 79) and then invoking `sys_dm`; for example

```
sys_numpar
par_1 = 25
par_2 = par_1 + 3
[A,B,C,D,E] = sys_dm
```

6.17 Report (rep)

MTT has a report-generator feature. The user specifies the report contents in a text file (see Section 6.17.1 [Report (text)], page 63) using an appropriate text editor (see Section 10.3 [Text editors], page 79).

For example, the report can be viewed by typing

```
mtt system rep view
```

6.17.1 Language text (rep.txt)

The user specifies the report contents in a text file (see Section 6.17.1 [Report (text)], page 63) using an appropriate text editor (see Section 10.3 [Text editors], page 79). The text file contains lines which are either comments (indicated by %) or valid **MTT** commands. The report will then contain appropriate sections. The following languages are supported by the report generator:

`m` `octave` a high-level interactive language for numerical computation.

`r` `reduce` a high-level interactive language for symbolic computation.

`tex` `latex` a text processor.

`ps` `ghostview` another document viewer.

`c` `gcc` a c compiler.

For example:

```
mtt rc abg tex
mtt rc cbg ps
mtt rc struc tex
mtt rc ode tex
mtt rc sro ps
mtt rc tf tex
mtt rc lmfr ps
```

The acausal bond graph (abg) (see Section 6.4 [Acausal bond graph (abg)], page 27) with the tex language is handled in a special way: the acausal Bond Graph in fig format (see Section 6.5.1 [Language fig (abg.fig)], page 28), the label file (see Section 6.7 [Labels (lbl)], page 37) the description file (see Section 8.2.2 [Detailed], page 77), together with corresponding subsystems are included in the report. It is recommended that the first (non-comment line) in the file should be:

```
mtt <system> abg tex
```

where `<system>` is the name of the (top-level) system.

As usual, **MTT** provides a default text file to be edited by the user (see Section 10.3 [Text editors], page 79).

In the special case that the first argument to mtt (normally the system) is a directory, a default text file is provided which generates a report for all systems to be found in that directory tree.

6.17.2 Language view

This representation has the standard text view (see Section 10.1 [Views], page 79).

7 Extending MTT

MTT has a number of built-in mechanisms for the user to extend its capabilities. As **MTT** is based on 'Make' it is unsurprising that some of these involve the creation of 'make files'.

7.1 Makefiles

If a file called 'Makefile' exists in the current directory, **MTT** executes it using make before doing anything else. This is useful if one of the .txt files contains a reference to, for example, an octave function of which **MTT** unaware. Such a function can be created using the makefile. An example 'Makefile' is

```
# Makefile for the Two link GMV example

all: msdP_tf.m TwoLinkP_obs.m TwoLinkP_sm.m twolinkp_sm.m TwoLinkGMV_numpar.m

msdP_tf.m: msdP_abg.fig
        mtt -q msdP tf m

TwoLinkP_obs.m: TwoLinkP_abg.fig TwoLinkP_lbl.txt
        mtt -q TwoLinkP obs m

TwoLinkP_sm.m: TwoLinkP_abg.fig TwoLinkP_lbl.txt
        mtt -q TwoLinkP sm m

twolinkp_sm.m: TwoLinkP_sm.m
        cp -v TwoLinkP_sm.m twolinkp_sm.m

TwoLinkGMV_numpar.m: TwoLinkGMV_numpar.txt
        mtt -q TwoLinkGMV numpar m
```

All of the files in the line stating 'all:' are created when **MTT** is executed (if they don't already exist).

7.2 New (DIY) representations

It may be convenient to create new representations for **MTT**; in particular, it is nice to be able to include the result of some numerical or symbolic computations within an **MTT** report (see Section 6.17 [Report], page 63). Therefore **MTT** provides a mechanism for doing this.

Future extensions of **MTT** will use such representations stored in $MTT_REP.

There are three parts to creating a DIY representation called myrep

1. Creating a make file in Make format called myrep_rep.make
2. Optionally creating a shell script called myrep_rep.sh
3. Optionally creating a documentation file in LaTeX format called myrep_rep.tex

7.2.1 Makefile

To create a new representation 'myrep' in a language 'mylang', create a file with the name

```
myrep_rep.make
```

This file must contain text in 'make' syntax. It is executed by **MTT** and the two arguments 'SYS' (the system name) and 'LANG' (the language) are passed to it by **MTT**. Note that **MTT** cannot know of any prerequisites, but these can be explicitly included in the makefile (which may include execution of **MTT** itself.

The following example declares the new representation 'ident' which is created in conjunction with the shell-script ident_rep.sh (see Section 7.2.2 [Shell-script (DIY representations)], page 69).

```
# -*-makefile-*-

#SUMMARY        Identification
#DESCRIPTION    Partially know system identification using
#DESCRIPTION    using bond graphs

# Makefile for representation ident
# File ident_rep.make

#Copyright (C) 2000,2001,2002 by Peter J. Gawthrop

## Model targets
model_reps =  ${MTT_SYS}_sympar.m ${MTT_SYS}_simpar.m ${MTT_SYS}_state.m
model_reps += ${MTT_SYS}_numpar.m ${MTT_SYS}_input.m ${MTT_SYS}_ode2odes.m
model_reps += ${MTT_SYS}_def.m

## Prepend s to get the sensitivity targets
sensitivity_reps = ${model_reps:%=s%}

## Model prerequisites
model_pre =  ${MTT_SYS}_abg.fig ${MTT_SYS}_lbl.txt
model_pre += ${MTT_SYS}_rdae.r ${MTT_SYS}_numpar.txt

## Prepend s to get the sensitivity targets
sensitivity_pre = ${model_pre:%=s%}


## Simulation targets
sims = ${MTT_SYS}_sim.m s${MTT_SYS}_ssim.m

## m-files needed for ident
ident_m = ${MTT_SYS}_ident.m ${MTT_SYS}_ident_numpar.m

## Targets for the ident simulation
ident_reps = ${ident_m} ${sims} ${model_reps} ${sensitivity_reps}
```

```
## ps output files etc
psfiles = ${MTT_SYS}_ident.ps ${MTT_SYS}_ident.comparison.ps
figfiles = ${psfiles:%.ps=%.fig}
gdatfiles = ${psfiles:%.ps=%.gdat}
datfiles = ${psfiles:%.ps=%.dat2}

## LaTeX files etc
latexfiles = ${MTT_SYS}_ident_par.tex

all: ${MTT_SYS}_ident.${MTT_LANG}

echo:
        echo "sims: ${sims}"
        echo "model_reps: ${model_reps}"
        echo "sensitivity_reps: ${sensitivity_reps}"
        echo "ident_reps: ${ident_reps}"

${MTT_SYS}_ident.view: ${psfiles}
        ident_rep.sh ${MTT_SYS} view

${psfiles}: ${figfiles}
        ident_rep.sh ${MTT_SYS} ps

${figfiles}: ${gdatfiles}
        ident_rep.sh ${MTT_SYS} fig

${gdatfiles}: ${datfiles}
        ident_rep.sh ${MTT_SYS} gdat

${datfiles} ${latexfiles}: ${ident_reps}
        ident_rep.sh ${MTT_SYS} dat2

${MTT_SYS}_ident.m:
        ident_rep.sh ${MTT_SYS} m

${MTT_SYS}_ident_numpar.m:
        ident_rep.sh ${MTT_SYS} numpar.m

## System model reps
## Generic txt files
${MTT_SYS}_%.txt:
        mtt ${MTT_OPTS} -q -stdin ${MTT_SYS} $* txt

## Specific m files
${MTT_SYS}_ode2odes.m: ${model_pre}
        mtt -q -stdin ${MTT_OPTS} ${MTT_SYS} ode2odes m
```

```
${MTT_SYS}_sim.m: ${MTT_SYS}_ode2odes.m
        mtt ${MTT_OPTS} -q -stdin ${MTT_SYS} sim m

## Numpar files
${MTT_SYS}_numpar.m:
        mtt ${MTT_SYS} numpar m

## Sympar files
${MTT_SYS}_sympar.m:
        mtt ${MTT_SYS} sympar m

## Generic txt to m
${MTT_SYS}_%.m: ${MTT_SYS}_%.txt
        mtt ${MTT_OPTS} -q -stdin ${MTT_SYS} $* m

## r files
${MTT_SYS}_def.r: ${MTT_SYS}_abg.fig
        mtt ${MTT_OPTS} -q -stdin ${MTT_SYS} def r

${MTT_SYS}_rdae.r:
        mtt ${MTT_OPTS} -q -stdin ${MTT_SYS} rdae r

## Sensitivity model reps
## Generic txt files
s${MTT_SYS}_%.txt:
        mtt ${MTT_OPTS} -q -stdin -s s${MTT_SYS} $* txt

## Specific m files
## Numpar files
s${MTT_SYS}_numpar.m:
        mtt -s s${MTT_SYS} numpar m

## Sympar files
s${MTT_SYS}_sympar.m:
        mtt -s s${MTT_SYS} sympar m

s${MTT_SYS}_ode2odes.m: ${sensitivity_pre}
        mtt -q -stdin ${MTT_OPTS} -s s${MTT_SYS} ode2odes m

s${MTT_SYS}_ssim.m:
        mtt -q -stdin ${MTT_OPTS} -s s${MTT_SYS} ssim m

s${MTT_SYS}_def.m:
        mtt -q -stdin ${MTT_OPTS} -s s${MTT_SYS} def m


## Generic txt to m
```

```
s${MTT_SYS}_%.m: s${MTT_SYS}_%.txt
        mtt ${MTT_OPTS} -q -stdin s${MTT_SYS} $* m

## r files
s${MTT_SYS}_rdae.r:
        mtt ${MTT_OPTS} -q -stdin -s s${MTT_SYS} rdae r
```

7.2.2 Shell-script

For more complex DIY representations, it is convenient to define new commands to be used by the Makefile (see Section 7.2.1 [Makefile (DIY representations)], page 66).

The following example shows this in the context of the DIY representation 'ident' used as an example in the previous section (see Section 7.2.1 [Makefile (DIY representations)], page 66).

```
#! /bin/sh

## ident_rep.sh
## DIY representation "ident" for mtt
# Copyright (C) 2002 by Peter J. Gawthrop

ps=ps

sys=$1
rep=ident
lang=$2
mtt_parameters=$3
rep_parameters=$4

## Some names
target=${sys}_${rep}.${lang}
def_file=${sys}_def.r
dat2_file=${sys}_ident.dat2
dat2s_file=${sys}_idents.dat2
ident_numpar_file=${sys}_ident_numpar.m
option_file=${sys}_ident_mtt_options.txt

## Get system information
if [ -f "${def_file}" ]; then
 echo Using ${def_file}
else
  mtt -q ${sys} def r
fi

ny=‘mtt_getsize $1 y‘
nu=‘mtt_getsize $1 u‘
```

```
check_new_options() {
    if [ -f "${option_file}" ]; then
        old_options=‘cat ${option_file}‘
        if [ "${mtt_options}" != "${old_options}" ]; then
           echo ${mtt_options} > ${option_file}
        fi
    else
        echo ${mtt_options} > ${option_file}
    fi
}

## Make the _ident.m file
make_ident() {
filename=${sys}_${rep}.m
date=‘date‘
echo Creating ${filename}

cat > ${filename} <<EOF
function [epar,Y] = ${sys}_ident (y,u,t,par_names,Q,extras)

  ## usage:  [epar,Y] = ${sys}_ident (y,u,t,par_names,Q,extras)
  ##
  ## last      last time in run
  ## ppp_names Column vector of names of ppp params
  ## par_names Column vector of names of estimated params
  ## extras    Structure containing additional info
  ##
  ## Created by MTT on ${date}

  ## Sensitivity system name
  system_name = "s${sys}"

  ##Sanity check
  if nargin<3
    printf("Usage: [y,u,t] = ${sys}_ident(y,u,t,par_names,Q,extras);");
    return
  endif

  if nargin<6
    ## Set up optional parameters
    extras.criterion = 1e-3;
    extras.emulate_timing = 0;
    extras.max_iterations = 10;
    extras.simulate = 2;
    extras.v =  1e-2;
    extras.verbose = 1;
    extras.visual = 1;
```

```
endif

## System info
[n_x,n_y,n_u,n_z,n_yz] = ${sys}_def;
sympar  = ${sys}_sympar;
simpar  = ${sys}_simpar;
sympars  = s${sys}_sympar;
simpars  = s${sys}_simpar;

## Parameter indices
i_par = ppp_indices (par_names,sympar,sympars);

## Initial model state
x_0 = zeros(2*n_x,1);

## Initial model parameters
par_0 = s${sys}_numpar;

## Reset simulation parameters
[n_data,m_data] = size(y);
dt = t(2)-t(1);
simpars.last = (n_data-1)*dt;
simpars.dt = dt;

## Identification
[epar,Par,Error,Y,iterations,x] = ppp_optimise(system_name,x_0,par_0,simpars,u,y,i_par,Q,

## Do some plots
figure(1);
title("Comparison of data");
xlabel("t");
ylabel("y");
[N,M] = size(Y);
plot(t,Y(:,M-n_y+1:M),"1;Estimated;", t,y,"3;Actual;");
figfig("${sys}_ident_comparison");

## Create a table of the parameters
[n_par,m_par] = size(i_par);
fid = fopen("${sys}_ident_par.tex", "w");
fprintf(fid,"\\\\begin{table}[htbp]\\n");
fprintf(fid," \\\\centering\\n");
fprintf(fid," \\\\begin{tabular}{|l|l|}\\n");
fprintf(fid,"  \\\\hline\\n");
fprintf(fid,"  Name & Value \\\\\\\\ \\n");
fprintf(fid,"  \\\\hline\\n");
for i = 1:n_par
  fprintf(fid,"$%s$ & %4.2f \\\\\\\\ \\n", par_names(i,:), epar(i_par(i,1)));
```

```
  endfor
  fprintf(fid,"  \\\\hline\\n");
  fprintf(fid,"\\\\end{tabular}\\n");
  fprintf(fid,"\\\\caption{Estimated Parameters}\\n");
  fprintf(fid,"\\\\end{table}\\n");
  fclose(fid);

endfunction
EOF
}

make_ident_numpar() {
echo Creating ${ident_numpar_file}
cat > ${sys}_ident_numpar.m <<EOF
function [y,u,t,par_names,Q,extras] = ${sys}_ident_numpar;

  ## usage: [y,u,t,par_names,Q,extras] = ${sys}_ident_numpar;
  ## Edit for your own requirements
  ## Created by MTT on ${date}


  ## This section sets up the data source
  ## simulate = 0  Real data (you supply ${sys}_ident_data.dat)
  ## simulate = 1  Real data input, simulated output
  ## simulate = 2  Unit step input, simulated output
  simulate = 2;


  ## System info
  [n_x,n_y,n_u,n_z,n_yz] = ${sys}_def;
  simpars = s${sys}_simpar;

  ## Access or create data
  if (simulate<2)               # Get the real data
    if (exist("${sys}_ident_data.dat")==2)
      printf("Loading ${sys}_ident_data.dat\n");
      load ${sys}_ident_data.dat
    else
      printf("Please create a loadable file ${sys}_ident_data.dat containing y,u and t\n");
      return
    endif
  else
    switch simulate
      case 2                    # Step simulation
        t = [0:simpars.dt:simpars.last]';
        u = ones(size(t));
      otherwise
```

```
        error(sprintf("simulate = %i not implemented", simulate));
    endswitch
  endif

  if (simulate>0)
    par = ${sys}_numpar();
    x_0 = ${sys}_state(par);
    dt = t(2)-t(1);
    simpars.dt = dt;
    simpars.last = t(length(t));
    y =  ${sys}_sim(zeros(n_x,1), par, simpars, u);
  endif

  ## Default parameter names - Put in your own here
  sympar = ${sys}_sympar;       # Symbolic params as structure
  par_names = struct_elements (sympar); # Symbolic params as strings
  [n,m] = size(par_names);      # Size the string list

  ## Sort by index
  for [i,name] = sympar
    par_names(i,:) = sprintf("%s%s",name, blanks(m-length(name)));
  endfor

  ## Output weighting vector
  Q = ones(n_y,1);

  ## Extra parameters
  extras.criterion = 1e-5;
  extras.emulate_timing = 0;
  extras.max_iterations = 10;
  extras.simulate = simulate;
  extras.v =  1e-2;
  extras.verbose = 1;
  extras.visual = 1;

endfunction
EOF
}

make_dat2() {

## Inform user
echo Creating ${dat2_file}

## Use octave to generate the data
octave -q <<EOF
  [y,u,t,par_names,Q,extras] = ${sys}_ident_numpar;
```

```
  [epar,Y] = ${sys}_ident (y,u,t,par_names,Q,extras);
  [N,M] = size(Y);
  y_est = Y(:,M);
  data = [t,y_est,u];
  save -ascii ${dat2_file} data
EOF

## Tidy up the latex stuff - convert foo_123 to foo_{123}
cat ${sys}_ident_par.tex > mtt_junk
sed  -e "s/_\([a-z0-9,]*\)/_{\1}/g" < mtt_junk >${sys}_ident_par.tex
rm mtt_junk
}

case ${lang} in
    numpar.m)
        ## Make the numpar stuff
        make_ident_numpar;
        ;;
    m)
        ## Make the code
        make_ident;
        ;;
    dat2)
        ## The dat2 language (output data) & fig file
        make_dat2;
        ;;
    gdat)
        cp ${dat2_file} ${dat2s_file}
        dat22dat ${sys} ${rep}
        dat2gdat ${sys} ${rep}
        ;;
    fig)
        gdat2fig ${sys}_${rep}
        ;;
    ps)
        figs=`ls ${sys}_ident*.fig | sed -e 's/\.fig//'`
        for fig in ${figs}; do
            fig2dev -Leps ${fig}.fig > ${fig}.ps
        done
        texs=`ls ${sys}_ident*.tex | sed -e 's/\.tex//'`
        for tex in ${texs}; do
          makedoc "" "${sys}" "ident_par" "tex" "" "" "$ps"
          doc2$ps ${sys}_ident_par "$documenttype"
        done
        ;;
    view)
        pss=`ls ${sys}_ident*.ps`
```

```
        echo Viewing ${pss}
        for ps in ${pss}; do
          gv ${ps}&
        done
        ;;
    *)
        echo Language ${lang} not supported by ${rep} representation
        exit 3
esac
```

7.2.3 Documentation

7.3 Component library

If **MTT** does not recognise a component (eg named MyComponent) as a simple component (see Section 6.5.1.5 [Simple components], page 30) or as already existing, it searches the library search path $MTT_COMPONENTS (see Section 11.4.2 [$MTT_COMPONENTS], page 85) for a directory called MyComponent containing MyComponent_lbl.txt. It then copies the *entire* directory into the current working directory. Thus, for example, the directory could contain MyComponent_desc.tex MyComponent_abg.fig MyComponent_lbl.txt and MyComponent_cr.r in addition to MyComponent_lbl.txt.

8 Documentation

8.1 Manual

MTT is documented in this manual. The manual can be invoked in various ways:

`mtt manual`
: Brings up a pdf version of the manual

`mtt info`
: Brings up an xterm containing an info version of the manual

`mtt hinfo`
: Brings up an html browser containing the manual

`emacs`
: type ^h^i followed by mmtt in the command window

`browser`
: point browser to http://mtt.sf.net

8.2 On-line documentation

MTT components, constitutive relations, examples and representations in libraries (see Section 7.3 [Component library], page 75) are documented in two ways:

1. brief
2. verbose

8.2.1 Brief on-line documentation

Documentation of DIY components, examples, constitutive relationships and representations is provides by the programmer by inserting code of the form

```
#SUMMARY     One line summary
#DESCRIPTION Multi-line
#DESCRIPTION More detailed description
```

within the appropriate file (usually at or near the top):

`components`
: _lbl.txt (see Section 6.7 [Labels (lbl)], page 37)

`examples`
: _lbl.txt (see Section 6.7 [Labels (lbl)], page 37)

`constitutive relations`
: _cr.r (see Section 6.9.2 [DIY constitutive relationships], page 48)

`representations`
: _rep.make (see Section 7.2.1 [Makefile (DIY representations)], page 66)

This documentation is accessed by the user in various ways

`mtt help name`
: prints basic information on the screen

`mtt system lbl view`
: gives formatted information about the component or example

`Including mtt system abg tex in the _rep.txt file`
: gives formatted information about the component or example within the report

8.2.2 Detailed on-line documentation

DIY components, examples, constitutive relationships can be described textually in LaTeX (.tex) description file; this is the only language for this representation. This representation is used by the LaTeX language version (see Section 6.5.4 [Language tex (abg.tex)], page 36) of the acausal bond graph representation (see Section 6.4 [Acausal bond graph (abg)], page 27).

The file may contain any LaTeX commands valis for the "article" document type but must **not** contain:

- documentclass commands
- document environments

9 Languages

These are a number of languages used by **MTT** to implement the various representations. Each has associated Language tools (see Chapter 10 [Language tools], page 79) to manipulate and/or view the representation.

`fig`	`Fig` a graphical description language.
`m`	`octave` a high-level interactive language for numerical computation.
`r`	`reduce` a high-level interactive language for symbolic computation.
`tex`	`latex` a text processor.
`dvi`	`xdvi` a document viewer.
`ps`	`ghostview` another document viewer.
`gdat`	`gnuplot` a data viewer.
`c`	`gcc` a c compiler.
`sg`	`scigraphica` a plotting package.

These tools are automatically invoked as appropriate by **MTT**; but for more advanced use, these tools can be used directly on files (with the appropriate suffix) generated by **MTT**.

9.1 Fig

Please see xfig documentation.

9.2 m

Please see Octave documentation

9.3 Reduce

Please see the reduce documentation.

9.4 c

Please see the gcc documentation.

10 Language tools

10.1 Views

A number of representations (see Chapter 6 [Representations], page 25) have a language representation which is particularly useful for viewing by the user. These views are invoked, where appropriate by the command:

```
mtt sys rep view
```

where `sys` is the system name and `rep` a corresponding representation.

10.2 Xfig

10.3 Text editors

All representations live in text files and thus may be edited using your favourite text editor; however, the Fig (see Section 9.1 [Fig], page 78) representation is pretty meaningless in this form and so you should use Xfig (see Section 10.2 [Xfig], page 79) for representation in this language.

Its up to you which text editor to use. I recommend emacs, but simpler (and less powerful) editors such as xedit, textedit and vi are also ok.

I usually run **MTT** out of an emacs shell window and keep the rest of the files in emacs buffers.

10.4 Octave

Octave is a numerical matrix-based language See Section "Octave" in *Octave*. It is similar to Matlab in many ways. In most cases, m-files generated by **MTT** can be understood by both Matlab and Octave (and no doubt other Matlab lookalikes).

MTT provides the octave function `mtt`. The octave command

```
help mtt
```

gives the following information:

```
usage:  mtt (system[,representation,language])

Invokes mtt from octave to generate system_representation.language
Ie equivalent to "mtt system representation language" at the shell
Representation and language defualt to "sm" and "m" respectively
```

Thus for example, if octave is in the directory containing the system rc the following session generates the state matrices of the system "rc" with the defaut capacitance but resitance r=0.1.

```
octave> mtt("rc");
Creating rc_rbg.m
Creating rc_cmp.m
Creating rc_fig.fig
Creating rc_sabg.fig
```

```
Creating rc_alias.txt
Creating rc_alias.m
Creating rc_sub.sh
Creating rc_abg.m
Creating rc_cbg.m (maximise integral causality)
Creating rc_type.sh
Creating rc_ese.r
Creating rc_def.r
Creating rc_struc.txt
Creating rc_rdae.r
Creating rc_subs.r
Creating rc_cr.txt
Creating rc_cr.r
Copying CR SS to here from
Copying CR lin to here from
Creating rc_dae.r
Creating rc_sympar.txt
Creating rc_sympar.r
Creating rc_cse.r
Creating rc_sspar.r
Creating rc_csm.r
Creating rc_ode.r
Creating rc_ss.r
Creating rc_sm.r
Creating rc_switch.txt
0 switches found
Creating rc_sympars.txt
Creating rc_sm.m
Copying rc_sm.m
octave> mtt("rc","numpar");
Creating rc_numpar.txt
Creating rc_numpar.m
Copying rc_numpar.m
octave> mtt("rc","sympar");
Creating rc_sympar.m
Copying rc_sympar.m
octave> par = rc_numpar
par =

  1
  1

octave> sym = rc_sympar;

octave> par(sym.r) = 0.1;
octave> [A,B,C,D] = rc_sm(par)
A = -10
```

```
B = 10

C = 1

D = 0

octave>
```

generates the data structure rc corresponding the the bond graph of the system called 'rc'. The following octave commands then generate the step reponse and bode diagram respectively:

```
step(rc);
bode(rc);
```

10.4.1 Octave control system toolbox (OCST)

MTT provides an interface to the Octave control system toolbox (OCST) using the mfile `mtt2sys`. the octave command

```
help mtt2sys
```

gives the following information.

```
usage:  sys = mtt2sys (Name[,par])

Creates a sys structure for the Octave Control Systems Toolbox
from an MTT system with name "Name"
Optional second argument is system parameter list
Assumes that Name_sm.m, Name_struc.m and Name_numpar.m exist
```

Thus for example, if octave is in the directory containing the system rc:

```
rc = mtt2sys("rc");
```

generates the data structure rc corresponding the the bond graph of the system called 'rc'. The following octave commands then generate the step reponse and bode diagram respectively:

```
step(rc);
bode(rc);
```

10.4.2 Creating GNU Octave .oct files

GNU Octave dynamically loaded functions (.oct files) can be created by instructing **MTT** to create the "oct" representation:

```
mtt [options] sys ode oct
```

This will cause **MTT** to create the C++ representation of the system (sys_ode.cc) and to then compile it as a shared object suitable for use within Octave. The resultant file may be used in an identical manner to the equivalent, but generally slower, interpreted .m file.

Usage information for the function may be obtained within Octave in the usual manner:

```
octave:1> help rc_ode

rc_ode is the dynamically-linked function from the file
```

```
/home/mttuser/rc/rc_ode.oct

Usage: [mttdx] = rc_ode(mttx,mttu,mttt,mttpar)
Octave ode representation of system rc
Generated by MTT on Fri Jul  5 11:23:08 BST 2002
```

Note that the first line of output from Octave identifies whether the compiled or interpreted function is being used.

Alternatively, standard representations may be generated using the Octave DLDs by use of the "-oct" switch:

```
mtt -oct rc odeso view
```

In order to successfully generate .oct files, Octave must be correctly configured prior to compilation and certain headers and libraries must be correctly installed on the system (see Section 11.3.2 [.oct file dependencies], page 85).

10.4.3 Creating Matlab .mex files

On GNU/Linux systems, Matlab dynamically linked executables (.mexglx files) can created by instructing **MTT** to create the "mexglx" representation:

```
mtt [options] sys ode mexglx
```

This will cause **MTT** to create the C++ representation of the system (sys_ode.cc) and to then compile it as a shared object suitable for use within Matlab.

If it is necessary to compile mex files for another platform, then the usual C++ representation (generated with the -cc flag) can be created and the resultant file compiled with the -DCODEGENTARGET=MATLABMEX flag on the target platform.

```
mtt_machine:
mtt -cc rc ode cc

matlab_machine:
matlab> mex -DCODEGENTARGET=MATLABMEX rc_ode.cc
```

10.4.4 Embedding MTT models in Simulink

It is possible to embed **MTT** functions or entire **MTT** models within Simulink simulations as Sfun blocks. If the zip package is installed on the system, the command

```
mtt sys sfun zip
```

will create a compressed archive containing sys.mdl, which may be embedded into a larger Simulink model. Also contained within the archive will be four sys_sfun*.c files,

- sys_sfun.c model state and output equations
- sys_sfun_ae.c model algebraic equations
- sys_sfun_input.c model inputs
- sys_sfun_interface.c interface between MTT model and Simulink

The last of these files must be edited to correctly map the inputs and outputs between the **MTT** and Simulink models. The two sections to edit are clearly marked with

```
/* Start EDIT */
....
```

```
/* End EDIT */
```

These four files should then be compiled with the Matlab "mex" compiler as described in the *README* file in the archive.

If it is desired to compile the .mex files directly from within **MTT** on a machine which has the Matlab header files installed, this may be done with the command

```
mtt sys sfun mexglx
```

which will generated the four .mex files and the .mdl file. In this case, the user must ensure that *sys_sfun_interface.c* has been correctly edited prior to compilation.

Note that solution of algebraic equations within Simulink is not possible unless the *Matlab Optimisation Toolbox* is installed.

10.5 LaTeX

LaTeX is a powerful text processor which **MTT** uses to provide visual output.

11 Administration

11.1 Software components

MTT is built from a set of readily-available software tools. These are:

- General purpose software tools.
- Octave (see Section 11.3 [Octave setup], page 85)
- REDUCE (see Section 11.2 [REDUCE setup], page 84)

The General purpose tools are (these will all be available with a standard Linux distribution):

`sh` Bourne shell

`gmake` Gnu make

`gawk` Gnu awk

`sed` Gnu sed

`grep` Gnu grep

`comm` Gnu Compare sorted files by line

`xfig` Figure editor, version 3 or greater.

`fig2dev` Fig file conversion, version 3 or greater.

`ghostview`
postscript viewer

`xdvi` dvi viewer

`dvips` dvi to postscript conversion

`latex` the text processor (LaTeX2e needed)

`latex2html`
converts latex to html

`perl` needed for latex2html

`gnuplot` a graph plotting program

`gnuscape` or other web/html browser such as netscape, Red Baron etc.

`gcc` GNU c compiler

11.2 REDUCE setup

Symbolic algebra is performed by REDUCE, which although not free software is the the result of international collaboration. The version I use is obtained from:

ZIB (http://www.zib.de)

11.3 Octave setup

Octave is available at various web sites including: http://www.octave.org

11.3.1 .octaverc

The `.octaverc` file should contain the following lines:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Startup file for Octave for use with MTT
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

implicit_str_to_num_ok = 1;
empty_list_elements_ok = 1;
```

11.3.2 .oct file dependencies

Successful compilation of .oct code requires that Octave has been configured to use dynamically linked libraries and that the Octave libraries `liboctave`, `libcruft` and `liboctinterp` are available on the system.

This can be acheived by compiling Octave from the source code, configured with the options `--enable-shared` and `--enable-dl`.

A number of additional libraries and headers are also required to be installed on a system. These include,

- *ncurses* and *readline* terminal control routines
- *blas* or *altas* basic linear algebra subprograms, usually optimised for the specific processor
- *fftw* fast Fourier transform routines
- *g2c* GNU Fortran to C conversion routines
- *kpathsea* TeX path search routines

Note that on many GNU/Linux distributions, the necessary headers are contained in development packages which must be installed in addition to the standard library package.

Further information on configuring and installing Octave to handle dynamic libraries (DLDs) can be found in the Octave documentation.

11.4 Paths

There are a number of paths that must be set correctely for **MTT** to work. These are normally set up by sourcing the file `mttrc` that lives in the **MTT** home directory.

11.4.1 $MTTPATH

The environment variable $MTTPATH points to the mtt home directory. This is usually `/usr/local/lib/mtt`.

11.4.2 $MTT_COMPONENTS

The environment variable $MTT_COMPONENTS is a colon-separated path pointing to directories containing components and subsystems. By default

```
MTT_COMPONENTS=.:$MTT_LIB/lib/comp/
```

but you may wish to add your own component libraries:

```
MTT_COMPONENTS=my_library_path:$MTT_COMPONENTS
```

11.4.3 $MTT_CRS

The environment variable $MTT_CRS is a colon-separated path pointing to directories containing constitutive relationships. By default

```
MTT_CRS=$MTTPATH/lib/cr
```

but you may wish to add your own component libraries:

```
MTT_CRS=my_cr_path:$MTT_CRS
```

11.4.4 $MTT_EXAMPLES

The environment variable $MTT_EXAMPLES is a colon-separated path pointing to directories containing EXAMPLES and subsystems. By default

```
MTT_EXAMPLES=$MTTPATH/lib/examples
```

but you may wish to add your own component libraries:

```
MTT_EXAMPLES=my_examples_path:$MTT_EXAMPLES
```

11.4.5 $OCTAVE_PATH

The `$OCTAVE_PATH` path must include the relevant paths for mtt to work properly. In particular, it must include:

```
$MTTPATH/trans/m
$MTTPATH/lib/comp/simple
$MTTPATH/lib/comp/compound
```

11.5 File structure

The recommended installation of **MTT** uses the following directory structure with corresponding contents. Normally, each of the listed directories is a subdirectory of `/usr/local`. The directory `mtt` is pointed to by $MTTPATH (see Section 11.4.1 [$MTTPATH], page 85).

`mtt/bin` This is the home directory for **MTT**. **MTT** itself lives here along with `mttrc`.

`mtt/bin/trans`
: The transformations executed by **MTT**.

`mtt/bin/trans/m`
: The `m-files` associated with the transformations.

`mtt/bin/trans/awk`
: The `awk` scripts associated with the transformations.

`mtt/lib` The place for components, examples and CRs which will be updated.

`mtt/lib/comp/simple`
: The `m-files` defining the simple components.

`mtt/lib/comp/compound`
: The `m-files` defining the compound components.

`mtt/lib/cr/r`
: constitutive relationship definitions

`mtt/lib/examples`
: Some examples.

`mtt/examples/metamodelling`
: Examples from the book.

`mtt/doc` The documentation files for **MTT**.

`mtt/doc/Examples`
: Examples used in the documentation.

Appendix A Legal stuff

A.1 GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.1.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

A.2 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

A.2.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

A.2.2 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted,

and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as