

MTT: Model Transformation Tools

September 2000
For version 4.6.

Peter Gawthrop

Copyright © 1996,1997,1998,1999,2000 Peter J. Gawthrop

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the section entitled “GNU General Public License” is included exactly in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the section entitled “GNU General Public License” may be included in a translation approved by the author instead of in the original English.

General information about MTT is available at URL

<http://www.mech.gla.ac.uk/~peterg/software/MTT>

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.
- Transformation₁ 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.
- Representation₁ 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 3) 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
- Transformation₁ → Representation₁
- Transformation₂ → Representation₂
- ...
- 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-stroke 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 6.4.1.2 \[bonds\], page 27](#)). Components have the following attributes:

ports provide the connections to other components (see [Section 1.6.1 \[Ports\], page 5](#))

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 6.4.1.9 \[Named SS components\]](#), page 29).

The direction of the named SS components. (see [Section 6.4.1.9 \[Named SS components\]](#), page 29) is coerced (see [Section 6.4.1.10 \[Coerced bond direction\]](#), page 30) to have the same direction as the bonds 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

Numeric 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 9.4 \[Octave\]](#), page 64).

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.

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 and 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

2 User interface

There are two user interfaces to **MTT**: a command line interface (see [Section 2.2 \[Command line interface\]](#), page 7) and a menu-driven interface (see [Section 2.1 \[Menu-driven interface\]](#), page 7).

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 8)

<system_name>

the name of the system being transformed

<representation>

the mnemonic for the system representation (see [Section 6.1 \[Representation summary\]](#), page 23)

<language>

the mnemonic for language for the representation (see [Chapter 8 \[Languages\]](#), page 63)

for example

```
mtt rc rep view
```

creates a view of the report describing system rc and

```
mtt rc sm m
```

creates an m file (suitable 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 -s -c syst cbg view
```

invokes the `-o`, `-s`, and `-c` options.

The available options are:

- `-q` quiet mode – suppress MTT banner
- `-A` solve algebraic equations symbolically
- `-D` debug – leave log files etc
- `-I` prints more information
- `-abg` start at `abg.m` representation
- `-c` c-code generation
- `-d <dir>` use directory `<dir>`
- `-dc` Maximise derivative (not integral) causality
- `-dc` Maximise derivative (not integral) causality
- `-i <implicit|euler>`
 Use implicit or euler integration
- `-o` ode is same as dae
- `-oct` use oct files in place of m files where appropriate
- `-opt` optimise code generation
- `-p` print environment variables
- `-partition`
 partition hierachical system
- `-r` reset time stamp on representation
- `-s` try to generate sensitivity BG (experimental)
- `-ss` use steady-state info to initialise simulations
- `-stdin` read input data from standard input for simulations
- `-sub <subsystem>`
 operate on this subsystem
- `-t` tidy mode (default)

```

-u          untidy mode (leaves files in current dir)
-v          verbose mode
-viewlevel <N>
            View N levels of hierachy
--version
            print version and exit
--versions
            print version of mtt and components and exit

```

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 di-
            rectory.

mtt rename <old_name> <new_name>
            Renames all of the defining representations (see Section 6.2 \[Defining represen-
            tations\], page 25) 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 cur-
            rent 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 'sys-
            tem'.

```

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 23](#)) 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 25](#)).

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 25](#)).

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 through **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 -c 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 58) 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 58), 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 -c 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 9.2 \[Xfig\], page 64](#)). 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 the 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 17](#)) 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 6.4.1 \[Language fig \(abg.fig\)\], page 26](#)), including the bonds (see [Section 1.5 \[Bonds\], page 4](#)), the components (see [Section 1.6 \[Components\], page 4](#)) and any artwork (see [Section 6.4.1.15 \[artwork\], page 31](#)) to make the Bond Graph more readable. The graphical editor xfig is (see [Section 9.2 \[Xfig\], page 64](#)) is self-explanatory. The icon library is helpful here (see [Section 6.4.1.1 \[icon library\], page 26](#)).
4. Add causal strokes (see [Section 6.4.1.3 \[strokes\], page 27](#)) 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 6.4.1.6 \[SS components\], page 29](#)).

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 25](#)) used to further specify the system. These can now be edited using your favorite text editor (see [Section 9.3 \[Text editors\], page 64](#)).

8. **MTT** will now generate the representations (see [Section 6.1 \[Representation summary\]](#), [page 23](#)) 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 15](#)) – 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 5](#)); within each component, ports are represented by named SS components (see [Section 6.4.1.9 \[Named SS components\]](#), [page 29](#)); outwith each component, ports are unambiguously identified by labels (see [Section 6.4.1.11 \[Port labels\]](#), [page 30](#)) and vector labels (see [Section 6.4.1.12 \[Vector port labels\]](#), [page 30](#)).

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 15](#)) is then followed using the top level system (see [Section 3.3.1 \[Top level\]](#), [page 17](#)); **MTT** then recursively operates on the lower level systems.

The report representation (see [Section 6.17 \[Report\]](#), [page 58](#)) 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 components (see [Section 6.4.1.9 \[Named SS components\]](#), [page 29](#)) are treated as ordinary SS components (see [Section 6.4.1.6 \[SS components\]](#), [page 29](#)).

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 23), 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 23):

`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 8 \[Languages\]](#), page 63) 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.

There are a number solution algorithms available:

- explicit solution via the matrix exponential
- backward Euler integration (explicit)
- forward Euler integration (implicit)

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 -c system odeso view`

creates the response of a *nonlinear* system via the `system_ode.c` representation using implicit integration.

`mtt -c -i euler system odeso view`

creates the response of a *nonlinear* system via the `system_ode.c` representation using euler integration.

Simulation parameters are described in the `system_simpar.txt` file (see [Section 4.2 \[Simulation parameters\]](#), page 19).

The steady-state solution of a system can also be “simulated” (see [Section 4.1 \[Steady-state solutions \(odess\)\]](#), page 19).

4.1 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 19) is used to provide the time scales.

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)
- STEPFACOR 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
- 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 20) - uses the `smx` representation to give stability.

4.2.1 Euler integration

Euler integration approximates the solution of the Ordinary Differential Equation

$$\frac{dx}{dt} = f(x,u)$$

by

$$\mathbf{x} := \mathbf{x} + \mathbf{f}(\mathbf{x}, \mathbf{u}) * \text{DDT}$$

where

$$\text{DDT} = \text{DT} / \text{STEPFACTOR}$$

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

$$(\text{maximum eigenvalue of } -\mathbf{A}) * \text{DT} / 2$$

where \mathbf{A} is the $n \times n$ matrix appearing in

$$\mathbf{f}(\mathbf{x}, \mathbf{u}) = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$$

If the system is non linear, the linearised system matrix \mathbf{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

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, \mathbf{u})$$

by

$$(\mathbf{I} - \mathbf{A} * \text{DT})\mathbf{x} := (\mathbf{I} - \mathbf{A} * \text{DT})\mathbf{x} + \mathbf{f}(\mathbf{x}, \mathbf{u})\text{DT}$$

where \mathbf{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 unconditionally. 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 steady-state values are achieved.

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

$$\mathbf{E}(\mathbf{x}) \frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}, \mathbf{u})$$

where $\mathbf{E}(\mathbf{x})$ is a state-dependent matrix. The approximate solution is then given by:

$$(\mathbf{E}(\mathbf{x}) - \mathbf{A} * \text{DT})\mathbf{x} := (\mathbf{E}(\mathbf{x}) - \mathbf{A} * \text{DT})\mathbf{x} + \mathbf{f}(\mathbf{x}, \mathbf{u})\text{DT}$$

which reduces to the ordinary differential equation case when $\mathbf{E}(\mathbf{x}) = \mathbf{I}$.

The `_smx` representation includes the \mathbf{E} matrix.

4.3 Simulation input

4.4 Simulation initial state

The initial state of a simulation of is set in the `state` representation with the language `txt`.

As usual, `MTT` defaults this for you. There are two possibilities

- The `-ss` switch is not present: the states default to zero
- The `-ss` switch is present: the states default to those set in the `sspar.r` file.

4.5 Simulation output

The view (see [Section 9.1 \[Views\], page 64](#)) 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 -c -ss OttoCycle odeso ps 'OttoCycle_cycle_V'
```

An example of plotting one variable against another is:

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

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 6.4.1.14 \[Vector components\]](#), page 31). 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:

<code>scse</code>	sensitivity constrained-state equations
<code>sm</code>	sensitivity state matrices
<code>scsm</code>	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 23](#).

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 25](#).

Each representation is implemented in one or more languages depending on its use. These languages are discussed in [Chapter 8 \[Languages\]](#), [page 63](#) 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 23) associated with them.

Some of these representations define the system (see [Section 6.2 \[Defining representations\]](#), page 25).

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 9.3 \[Text editors\]](#), page 64) viewed or operated on by the appropriate tools (see [Chapter 9 \[Language tools\]](#), page 64).

`system_abg.fig`
the acausal bond graph (see [Section 6.4 \[Acausal bond graph \(abg\)\]](#), page 26)

`system_lbl.txt`
the label file (see [Section 6.6 \[Labels \(lbl\)\]](#), page 35)

`system_desc.tex`
the description file (see [Section 6.7 \[Description \(desc\)\]](#), page 43)

`system_simp.r`
algebraic simplifications to make output more readable

`system_subs.r`
algebraic substitutions to resolve, eq trig. identities

`system_simpar.txt`
simulation parameters

`system_numpar.txt`
numerical parameters

`system_input.txt`
the system input for simulations

`system_sspar.r`
defines the system steady-state

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 58) 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 9.3 \[Text editors\]](#), page 64).

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
```

6.4.1 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 6.4.1.1 \[icon library\]](#), page 26).

6.4.1.1 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
```

6.4.1.2 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 casual half-strokes (see [Section 6.4.1.3 \[strokes\], page 27](#)). Please adopt the convention of having the half-arrows below horizontal bonds and to the right of vertical bonds.

6.4.1.3 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 6.4.1.2 \[bonds\], page 27](#)) then it determines *flow* causality; if, on the other hand, it is on the *opposite* side to the half-arrow (see [Section 6.4.1.2 \[bonds\], page 27](#)) 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.

6.4.1.4 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 6.4.1.5 \[Simple components\], page 28](#)) or Compound components (see [Section 6.4.1.8 \[Compound components\], page 29](#)). 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 6.4.1.16 \[Valid names\]](#), page 32). The name provides a link to more information to be found in [Section 6.6 \[Labels \(lbl\)\]](#), page 35. For example:

`R:r`

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 6.4.1.11 \[Port labels\]](#), page 30); n copies of the component are concatenated with odd Named ports (see [Section 6.4.1.11 \[Port labels\]](#), page 30) 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`

6.4.1.5 Simple components

The following simple components are defined in MTT.

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

6.4.1.6 SS components

\$\$

SS components provide input and output variables for a system; Named SS components (see [Section 6.4.1.9 \[Named SS components\]](#), page 29) provide this for subsystems.

6.4.1.7 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 6.4.1.8 \[Compound components\]](#), page 29) are recommended for DIY components.

6.4.1.8 Compound components

Compound components are systems described by bond graphs and implemented by MTT. They have special SS components, Named SS components (see [Section 6.4.1.9 \[Named SS components\]](#), page 29), to indicate connections to the encapsulating system.

Like any other system, they are described by a graphical Bond Graph description (see [\[Language fig \(abg.fig\) \]](#), page [\[undefined\]](#)), and a label file (see [Section 6.6 \[Labels \(lbl\)\]](#), page 35).

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

6.4.1.9 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 6.4.1.8 \[Compound components\]](#), page 29. 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 5). The direction of the named SS components. (see [Section 6.4.1.9 \[Named SS components\]](#), page 29) is coerced (see [Section 6.4.1.10 \[Coerced bond direction\]](#), page 30) 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 17](#)) and is treated as an ordinary SS component with the given direction and with the attributes specified in the label file (see [Section 6.6 \[Labels \(lbl\)\], page 35](#)).

6.4.1.10 Coerced bond direction

Named SS components (see [Section 6.4.1.9 \[Named SS components\], page 29](#)) provide the mechanism for declaring the ports (see [Section 1.6.1 \[Ports\], page 5](#)) 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 6.4.1.9 \[Named SS components\], page 29](#)) is replaced by the direction of the bond attached to the component port.

6.4.1.11 Port labels

Most multi-port components have ports see [Section 1.6.1 \[Ports\], page 5](#)) 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 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 6.4.1.12 \[Vector port labels\], page 30](#)). 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 6.4.1.3 \[strokes\], page 27](#))

6.4.1.12 Vector port labels

Port labels (see [Section 6.4.1.11 \[Port labels\], page 30](#)) may be grouped into vector port labels of the form `[name1,name2,name3]`.

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

6.4.1.13 Port label defaults

Whether implicitly or explicitly, all ports of components (with the exception of 0 and 1 junctions) must have labels (see [Section 6.4.1.11 \[Port labels\], page 30](#)). However, these can be omitted from the bond graph in the following circumstances and default labels are supplied by **MTT**.

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

These defaults may, in turn be aliases (see [Section 6.6.7 \[Aliases\], page 38](#)) for port labels (see [Section 6.4.1.11 \[Port labels\], page 30](#)) or vector port labels (see [Section 6.4.1.12 \[Vector port labels\], page 30](#)). Combining the default and alias mechanism is a powerful tool for creating uncluttered, yet complex, bond graph models.

6.4.1.14 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 6.4.1.11 \[Port labels\], page 30](#)) containing numerals from 1 to the order of the vector. Thus a vector of 3 component is indicated by a port label of the form [1,2,3].

Within the corresponding label file (see [Section 6.6 \[Labels \(lbl\)\], page 35](#)), 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 6.6.7.1 \[Port aliases\], page 39](#))

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

6.4.1.15 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 8.1 \[Fig\], page 63](#)) object to the bond graph as long as it will not be interpreted as part of the bond graph. The recommended way to achieve 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 6.4.1.4 \[components\], page 27](#)). 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 6.5 \[Stripped acausal bond graph \(sabg\)\]](#), [page 34](#)) shows only those parts of the diagram recognised by **MTT** and is therefore useful for distinguishing artwork.

6.4.1.16 Valid Names

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

6.4.2 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 6.4.1 \[Language fig \(abg.fig\)\]](#), [page 26](#)) and m (see [Section 6.4.3 \[Language m \(abg.m\)\]](#), [page 33](#)) representations. It contains the geometric information from the fig file in a form digestible by Octave (see [Section 9.4 \[Octave\]](#), [page 64](#)).

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 6.4.1.2 \[bonds\]](#), [page 27](#))
- 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 6.4.1.3 \[strokes\]](#), [page 27](#))

- 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 6.4.1.4 \[components\]](#), [page 27](#))

- 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 6.4.1.11 \[Port labels\]](#), [page 30](#))

- 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 6.4.1.9 \[Named SS components\]](#), page 29).

6.4.2.1 Transformation `abg2rbg_fig2m`

This transformation takes the acausal bond graph as a fig file (see [Section 6.4.1 \[Language fig \(abg.fig\)\]](#), page 26) and transforms it into a raw bond graph in m-file format (see [Section 6.4.2 \[Language m \(rbg.m\)\]](#), page 32).

This transformation is implemented in GNU awk (gawk). It scans both the fig file (see [Section 6.4.1 \[Language fig \(abg.fig\)\]](#), page 26) and the label file (see [Section 6.6 \[Labels \(lbl\)\]](#), page 35) and generates the rbg (see [Section 6.4.2 \[Language m \(rbg.m\)\]](#), page 32) 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.

6.4.3 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 6.4.3.1 \[Arrow-orientated causality\]](#), page 34) causality of the *effort* variable.
- the second column contains the arrow-orientated (see [Section 6.4.3.1 \[Arrow-orientated causality\]](#), page 34) 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 6.4.1.9 \[Named SS components\]](#), page 29).

6.4.3.1 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 6.4.1.3 \[strokes\], page 27](#)) 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 6.4.3.2 \[Component-orientated causality\], page 34](#).

6.4.3.2 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 6.4.1.3 \[strokes\], page 27](#)) 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 6.4.3.1 \[Arrow-orientated causality\], page 34](#).

6.4.3.3 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.

6.4.4 Language `tex` (`abg.tex`)

For the purpose of producing a report (see [Section 6.17 \[Report\], page 58](#)), **MTT** generates a LaTeX (see [Section 9.5 \[LaTeX\], page 66](#)) file describing the bond graph and its subsystems. Additional information may be supplied using the description representation (see [Section 6.7 \[Description \(desc\)\], page 43](#)).

6.5 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 26](#)) without the artwork (see [Section 6.4.1.15 \[artwork\], page 31](#)). It is useful to check for mistakes by showing precisely what is recognised by **MTT**.

6.5.1 Language fig (sabg.fig)

The stripped acausal bond graph can be generated as a fig (see [Section 8.1 \[Fig\], page 63](#)) file using

```
mtt syst sabg fig
```

6.5.2 Stripped acausal bond graph (view)

This representation has the standard text view (see [Section 9.1 \[Views\], page 64](#)).

6.6 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

Each label contains three fields (in the following order) separated by white space and on one line:

1. The component name see [Section 6.6.3 \[Component names\], page 37](#). This must be a valid name (see [Section 6.4.1.16 \[Valid names\], page 32](#)).
2. The component constitutive relationship see [Section 6.6.4 \[Component constitutive relationship\], page 38](#)
3. The component arguments see [Section 6.6.5 \[Component arguments\], page 38](#)

Not each component see [Section 6.4.1.4 \[components\], page 27](#) needs a label, only those which are explicitly labeled on the Bond Graph see [Section 6.4 \[Acausal bond graph \(abg\)\], page 26](#). **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. 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 6.6.9 \[Old-style labels \(lbl\)\], page 41](#)) are NO LONGER supported – you are encouraged to convert them ASAP.

6.6.1 SS component labels

In addition to the label there are two information fields, see [Section 6.6 \[Labels \(lbl\)\], page 35](#). 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

```

6.6.2 Other component labels

In addition to the label there are two information fields, see [Section 6.6 \[Labels \(lbl\)\]](#), [page 35](#). They correspond to the constitutive relationship (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 [\(undefined\) \[Parameter passing \]](#), [page \(undefined\)](#)) subsystems.

6.6.3 Component names

The component name field must contain a valid name (see [Section 6.4.1.16 \[Valid names\]](#), [page 32](#) corresponding to the name (the bit after the :) of each named component (see [Section 6.4.1.4 \[components\]](#), [page 27](#)) on the bond graph (see [Section 6.4 \[Acausal bond graph \(abg\)\]](#), [page 26](#)).

6.6.4 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.

6.6.5 Component arguments

6.6.6 Variable declarations

It is sometimes useful to use variables (in addition to those implied by the Component arguments see [Section 6.6.5 \[Component arguments\], page 38](#)) to compute values in, for example the `numpar` file. These can be declared in the label file; for examples , the two variables `var1` and `var 2` can be declared as:

```
#VAR var1
#VAR var2
```

6.6.7 Aliases

Aliases provide a convenient mechanism for relabelling words appearing in the label file (see [Section 6.6 \[Labels \(lbl\)\], page 35](#)). There are three contexts in which the alias mechanism is used:

1. renaming ports (see [Section 6.6.7.1 \[Port aliases\], page 39](#)),
2. renaming parameters (see [Section 6.6.7.2 \[Parameter aliases\], page 39](#)) and
3. renaming components (see [Section 6.6.7.3 \[Component aliases\], page 40](#)).

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

6.6.7.1 Port aliases

Aliases provide a way of referring to (see [Section 6.4.1.11 \[Port labels\], page 30](#)) or vector port labels (see [Section 6.4.1.12 \[Vector port labels\], page 30](#)) on the bond graph using a short-hand notation. Within a component label file (see [Section 6.6 \[Labels \(lbl\)\], page 35](#)) statements of the following forms can occur

```
%ALIAS short_label      real_label
```

When the component is used within another component, the short_label 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 referred 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 6.4.1.12 \[Vector port labels\], page 30](#)) and the port label default (see [Section 6.4.1.13 \[Port label defaults\], page 31](#)) 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,Hydraulic_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.

6.6.7.2 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 6.6.8 \[Parameter passing\], page 40](#)).

6.6.7.3 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 6.4.1.8 \[Compound components\]](#), page 29) live within a directory of the same name.

6.6.8 Parameter passing

MTT supports parameter-passing to subsystems within label files (see [Section 6.6 \[Labels \(lbl\)\]](#), page 35). Within a subsystem, explicit constitutive relationships and parameters (or groups thereof) can be replaced by positional 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 6.6.7.2 \[Parameter aliases\]](#), page 39).

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`

6.6.9 Old-style labels (lbl)

Old style labels (mtt version 2.x) are supported by mtt version 3.x. However, you are advised to use the new form (see [Section 6.6 \[Labels \(lbl\)\], page 35](#)).

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 6.4.1.6 \[SS components\], page 29](#).
- Other components, see [Section 6.4.1.4 \[components\], page 27](#).

Named SS component, see [Section 6.4.1.9 \[Named SS components\], page 29](#) never have labels.

6.6.9.1 SS component labels (old-style)

In addition to the label there are two information fields, see [Section 6.6 \[Labels \(lbl\)\], page 35](#). 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
```

6.6.9.2 Other component labels (old-style)

In addition to the label there are two information fields, see [Section 6.6 \[Labels \(lbl\)\]](#), [page 35](#). They correspond to the constitutive relationship (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 6.6.9.3 \[Parameter passing \(old-style\)\]](#), [page 42](#)) subsystems.

6.6.9.3 Parameter passing (old-style)

MTT supports parameter-passing to (see [Section 6.6.9.3 \[Parameter passing \(old-style\)\]](#), [page 42](#)) subsystems within label files (see [Section 6.6 \[Labels \(lbl\)\]](#), [page 35](#)). 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 6.6 \[Labels \(lbl\)\], page 35](#)) file as:

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

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

6.7 Description (desc)

The bond graph 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 6.4.4 \[Language tex \(abg.tex\)\], page 34](#)) of the acausal bond graph representation (see [Section 6.4 \[Acausal bond graph \(abg\)\], page 26](#)).

6.7.1 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 file contains a description of the system structure (see [Section 6.8 \[Structure \(struc\)\]](#), [page 44](#) 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 14](#)) 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 9.5 \[LaTeX\]](#), [page 66](#)) file contains a description of the system structure (see [Section 6.8 \[Structure \(struc\)\]](#), [page 44](#) in `longtable` format. It is a useful item to include in a report(see [Section 6.17 \[Report\]](#), [page 58](#)).

6.8.3 Language tex (view)

This representation has the standard text view (see [Section 9.1 \[Views\]](#), [page 64](#)).

6.9 Constitutive relationship (cr)

The constitutive relationship (see [Section 1.6.2 \[Constitutive relationship\], page 5](#)) of a simple component (see [Section 6.4.1.5 \[Simple components\], page 28](#)) is defined in the symbolic algebra language Reduce (see [Section 8.3 \[Reduce\], page 63](#)). The constitutive relationship of a compound components (see [Section 6.4.1.8 \[Compound components\], page 29](#)) 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 8.3 \[Reduce\]](#), [page 63](#)). This requires some understanding as to how **MTT** represent the elementary system equations (see [Section 6.12 \[Elementary system equations\]](#), [page 51](#)). Looking at the predefined constitutive relationships is a good way to get started (see [Section 10.5 \[File structure\]](#), [page 69](#)).

6.10 Parameters

In general, `lbl` (see [Section 6.6 \[Labels \(lbl\)\]](#), [page 35](#)) 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 46](#)) 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 9.4 \[Octave\]](#), [page 64](#)); 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 25).
- 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 46). 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 1.61 2000/09/14 17:13:06 peterg Exp $
# %% $Log: mtt.texi,v $
# %% 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 STEPFACOR 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  
# %% Started 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 9.3 \[Text editors\]](#), page 64).

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 26).

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 6.4.1 \[Language fig \(abg.fig\)\]](#), page 26) 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 6.4.3.1 \[Arrow-orientated causality\], page 34](#)) causality of the *effort* variable.
- the second column contains the arrow-orientated (see [Section 6.4.3.1 \[Arrow-orientated causality\], page 34](#)) 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 6.4.3 \[Language m \(`abg.m`\)\], page 33](#)) and transforms it into a causal bond graph in m-file format (see [Section 6.11.2 \[Language m \(`cbg.m`\)\], page 50](#)).

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 8.3 \[Reduce\], page 63](#)) 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 50](#)) and transforms it into elementary system equations in Reduce (see [Section 8.3 \[Reduce\]](#), [page 63](#)) 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 8.3 \[Reduce\]](#), page 63).
- `m` m (see [Section 8.2 \[m\]](#), page 63).
- `view` reduce (see [Section 9.1 \[Views\]](#), page 64).

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 52) are represented in the reduce (see [Section 8.3 \[Reduce\]](#), page 63) 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 8.3 \[Reduce\]](#), page 63) to combine the elementary system equations (see [Section 6.12 \[Elementary system equations\]](#), page 51) 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 52) are represented in the `m` (see [Section 8.2 \[m\]](#), page 63) 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 corresponding 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 8.3 \[Reduce\]](#), page 63) to rewrite the elementary system equations (see [Section 6.12 \[Elementary system equations\]](#), page 51) in `m`-file format (see [Section 8.2 \[m\]](#), page 63). 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 resulting equations are of the form:

$$\begin{aligned} E(x) \, dx/dt &= f(x,u) \\ y &= g(x,u) \end{aligned}$$

They typically occur 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 8.3 \[Reduce\]](#), page 63).
- `m` `m` (see [Section 8.2 \[m\]](#), page 63).
- `view` `reduce` (see [Section 9.1 \[Views\]](#), page 64).

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 54) are represented in the reduce (see [Section 8.3 \[Reduce\]](#), page 63) 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 8.3 \[Reduce\]](#), page 63) 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 54).

6.14.2 Language m (view)

This representation has the standard text view (see [Section 9.1 \[Views\]](#), page 64).

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 8.3 \[Reduce\]](#), page 63).
m m (see [Section 8.2 \[m\]](#), page 63).
view reduce (see [Section 9.1 \[Views\]](#), page 64).

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 55) are represented in the reduce (see [Section 8.3 \[Reduce\]](#), page 63) 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 8.3 \[Reduce\]](#), page 63) to invert the E matrix of the constrained-state equations (see [Section 6.14 \[Constrained-state Equations\]](#), page 54) and simplify the result.

6.15.2 Language m (ode.m)

The system ODEs (see [Section 6.15 \[Ordinary Differential Equations\]](#), page 55) are represented in the m (see [Section 8.2 \[m\]](#), page 63) 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 8.3 \[Reduce\]](#), page 63) to rewrite the ordinary differential equations (see [Section 6.15 \[Ordinary Differential Equations\]](#), page 55) in m-file format (see [Section 8.2 \[m\]](#), page 63) . Numerical parameters are declared as global.

6.15.3 Language `m` (view)

This representation has the standard text view (see [Section 9.1 \[Views\]](#), page 64).

6.16 Descriptor matrices (`dm`)

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

$$\begin{aligned} E \, dx/dt &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

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

```
mtt sys dm lang
```

Valid languages are:

- `r` reduce (see [Section 8.3 \[Reduce\]](#), page 63).
- `m` m (see [Section 8.2 \[m\]](#), page 63).
- `view` reduce (see [Section 9.1 \[Views\]](#), page 64).

6.16.1 Language `reduce` (`dm.r`)

The system descriptor matrices (see [Section 6.16 \[Descriptor matrices\]](#), page 57) are represented in the `reduce` (see [Section 8.3 \[Reduce\]](#), page 63) language as arrays containing the four matrices. The arrays are:

```
MTTA        A
MTTB        B
MTTC        C
MTTD        D
MTTE        E
```

6.16.2 Language m (dm.m)

The system descriptor matrices (see [Section 6.16 \[Descriptor matrices\], page 57](#)) are represented in the m (see [Section 8.2 \[m\], page 63](#)) 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 9.4 \[Octave\], page 64](#)) 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 9.4 \[Octave\], page 64](#)) 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 58](#)) using an appropriate text editor (see [Section 9.3 \[Text editors\], page 64](#)).

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 58](#)) using an appropriate text editor (see [Section 9.3 \[Text editors\], page 64](#)). 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 26](#)) with the tex language is handled in a special way: the acausal Bond Graph in fig format (see [Section 6.4.1 \[Language fig \(abg.fig\)\], page 26](#)), the label file (see [Section 6.6 \[Labels \(lbl\)\], page 35](#)) the description file (see [Section 6.7 \[Description \(desc\)\], page 43](#)), 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 9.3 \[Text editors\], page 64](#)).

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 9.1 \[Views\], page 64](#)).

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 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** is 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 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 58).

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 ‘nppp’ which is created with the Octave script `sys_nppp.m` where ‘sys’ is the system name. This needs a number of files (for example ‘`sys_ode2odes.out`’) which are themselves created by **MTT**.

```
# --makefile--
# Makefile for representation nppp
# File nppp_rep.make

#Copyright (C) 2000 by Peter J. Gawthrop

all: $(SYS)_nppp.$(LANG)

$(SYS)_nppp.view: $(SYS)_nppp.ps
    echo Viewing $(SYS)_nppp.ps; ghostview $(SYS)_nppp.ps&

$(SYS)_nppp.ps: $(SYS)_ode2odes.out s$(SYS)_ode2odes.out \
    $(SYS)_sim.m s$(SYS)_sim.m \
    $(SYS)_state.m $(SYS)_sympar.m $(SYS)_numpar.m \
    s$(SYS)_state.m s$(SYS)_sympar.m s$(SYS)_numpar.m \
    $(SYS)_sm.m $(SYS)_def.m s$(SYS)_def.m
    octave $(SYS)_nppp.m

$(SYS)_ode2odes.out:
    mtt -q -c -stdin $(SYS) ode2odes out

s$(SYS)_ode2odes.out:
    mtt -q -c -stdin -s s$(SYS) ode2odes out

$(SYS)_sim.m:
    mtt -q -c $(SYS) sim m

s$(SYS)_sim.m:
    mtt -q -c -s s$(SYS) sim m

$(SYS)_state.m:
    mtt -q $(SYS) state m

$(SYS)_sympar.m :
    mtt -q $(SYS) sympar m

$(SYS)_numpar.m:
    mtt -q $(SYS) numpar m

s$(SYS)_state.m:
    mtt -q -s s$(SYS) state m

s$(SYS)_sympar.m :
    mtt -q -s s$(SYS) sympar m
```

```
s$(SYS)_numpar.m:  
    mtt -q -s s$(SYS) numpar m  
  
$(SYS)_sm.m:  
    mtt -q $(SYS) sm m  
  
$(SYS)_def.m:  
    mtt -q $(SYS) def m  
  
s$(SYS)_def.m:  
    mtt -q -s s$(SYS) def m
```

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

8 Languages

These are a number of languages used by **MTT** to implement the various representations. Each has associated Language tools (see [Chapter 9 \[Language tools\], page 64](#)) to manipulate and/or view the representation.

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

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**.

8.1 Fig

Please see xfig documentation.

8.2 m

Please see Octave documentation

8.3 Reduce

Please see the reduce documentation.

8.4 c

Please see the gcc documentation.

9 Language tools

9.1 Views

A number of representations (see [Chapter 6 \[Representations\], page 23](#)) 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.

9.2 Xfig

9.3 Text editors

All representations live in text files and thus may be edited using your favourite text editor; however, the Fig (see [Section 8.1 \[Fig\], page 63](#)) representation is pretty meaningless in this form and so you should use Xfig (see [Section 9.2 \[Xfig\], page 64](#)) 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.

9.4 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 default 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 default capacitance but resistance $r=0.1$.

```
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_struct.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 to the bond graph of the system called 'rc'. The following octave commands then generate the step response and bode diagram respectively:

```

step(rc);
bode(rc);

```

9.4.1 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 to the bond graph of the system called 'rc'. The following octave commands then generate the step response and bode diagram respectively:

```

step(rc);
bode(rc);

```

9.5 LaTeX

LaTeX is a powerful text processor which **MTT** uses to provide visual output.

10 Administration

10.1 Software components

MTT is built from a set of readily-available software tools. These are:

- General purpose software tools.
- Octave (see [Section 10.3 \[Octave setup\]](#), page 68)
- REDUCE (see [Section 10.2 \[REDUCE setup\]](#), page 68)

The General purpose tools are (these will all be available with a standard Linux distribution):

<code>sh</code>	Bourne shell
<code>gmake</code>	Gnu make
<code>gawk</code>	Gnu awk
<code>sed</code>	Gnu sed
<code>grep</code>	Gnu grep
<code>comm</code>	Gnu Compare sorted files by line
<code>xfig</code>	Figure editor, version 3 or greater.
<code>fig2dev</code>	Fig file conversion, version 3 or greater.
<code>ghostview</code>	postscript viewer
<code>xdvi</code>	dvi viewer
<code>dvips</code>	dvi to postscript conversion
<code>latex</code>	the text processor (LaTeX2e needed)
<code>latex2html</code>	converts latex to html
<code>perl</code>	needed for latex2html
<code>gnuplot</code>	a graph plotting program
<code>gnuscape</code>	or other web/html browser such as netscape, Red Baron etc.
<code>gcc</code>	GNU c compiler

10.2 REDUCE setup

Symbolic algebra is performed by REDUCE, which although not free software is the result of international collaboration. The version I use is obtained from:

ZIB (<http://www.zib.de>)

10.3 Octave setup

Octave is available at various web sites including:

10.3.1 .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;
```

10.4 Paths

There are a number of paths that must be set correctly for **MTT** to work. These are normally set up by sourcing the file `mttrc` that lives in the **MTT** home directory.

10.4.1 \$MTTPATH

The environment variable `$MTTPATH` points to the `mtt` home directory. This is usually `/usr/local/lib/mtt`.

10.4.2 \$MTT_COMPONENTS

The environment variable `$MTT_COMPONENTS` is a colon-separated path pointing to directories containing components and subsystems. By default

```
MTT_COMPONENTS=$MTTPATH/lib/comp
```

but you may wish to add your own component libraries:

```
MTT_COMPONENTS=my_library_path:$MTT_COMPONENTS
```

10.4.3 \$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
```

10.4.4 \$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
```

10.4.5 \$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
```

10.5 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 10.4.1 \[`\$MTTPATH`\], page 68](#)).

`'mtt'` This is the home directory for **MTT**. **MTT** itself lives here along with `'mttrc'`.

`'mtt/trans'`
The transformations executed by **MTT**.

`'mtt/trans/m'`
The `m`-files associated with the transformations.

`'mtt/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.

Glossary

,

'name1:name2' 21
 'name1;name2;...;namen' 21

A

abg 23
 AE 28
 AF 28
 artwork 26
 assignment statements 47

B

bonds 26

C

c 18, 47, 58, 63
 C 28
 cbg 23
 commented assignment statements 47
 comments 47
 components 26
 cr 23
 cse 18, 23
 csm 23
 CSW 28

D

dae 23
 daes 23
 daeso 23
 def 23
 desc 23
 dm 23
 dvi 63

E

ese 23
 exotherm 46

F

fig 63
 fr 23

G

gdat 63
 GY 28

I

I 28
 input 23
 ir 18, 23
 iro 18, 23
 ISW 28

L

lbl 24
 lin 45
 lmfr 24
 lpfr 24

M

m 18, 47, 58, 63
 mtt -c -i euler system odeso view 19
 mtt -c system odeso view 19
 mtt <system> clean 9
 mtt clean 9
 mtt copy <system> 9
 mtt help 9
 mtt rename <old_name> <new_name> 9
 mtt system iro view 18
 mtt system representation vc 9, 12
 mtt system sro view 18
 mtt system vc 9, 12

N

NAME_cause.m 29
 NAME_eqn.m 29
 nifr 24
 numpar 24
 nyfr 24

O

obs	24
ode	18, 24
odes	21, 24
odeso	21, 24
odess	24
odesso	24

P

ps	58, 63
----------	--------

R

r	58, 63
R	28
rbg	24
rep	24
rfe	24

S

sabg	24
scse	22
scsm	22

simp	24
sm	22, 24
sms	24
smss	24
smx	24
sr	18, 24
sro	18, 24
ss	24
SS	28
sspar	24
strokes	26
struc	24
sub	24
sympar	25

T

tex	58, 63
tf	25
TF	28
txt	47
type	27
type*n	28
type:label	28
type:label*n	28

Index

<	
<name>	11
A	
Acausal bond graph (abg)	26
Administration	67
Algebraic loops	5
aliases	38
Arrow-orientated causality	34
artwork	31
B	
Bond graphs, what are they?	3
bonds	27, 33
Bonds	4
browser	9
C	
c	63
Causal bond graph (cbg)	50
cbonds	50
Clean	12
Coerced bond direction	30
Command line interface	7
component aliases	40
Component arguments	38
Component constitutive relationship	38
Component names	37
Component-orientated causality	34
components	10, 27, 33
Components	4
compound components	70
Compound components	29
Constitutive relationship	45
Constitutive Relationship	5
Constrained-state Equations	54
Constrained-state Equations (reduce)	55
Constrained-state Equations (view)	55
control systems	66
Copy	11
Creating complex models	17
Creating Models	14
Creating simple models	15
crs	11
cse.r	55
D	
DAE	52
dae.m	54
dae.r	53
def.r	52
Defining representations	23, 25
desc	43
Description	43
Descriptor matrices	57
Descriptor matrices (m)	58
Descriptor matrices (reduce)	57
Differential-Algebraic Equations	52
Differential-Algebraic Equations (m)	54
Differential-Algebraic Equations (reduce)	53
DIY constitutive relationships	46
dm	57
dm.m	58
dm.r	57
E	
Elementary system equations	51
Euler integration	19
examples	10
Extending MTT	60
F	
Fig	63
File structure	69
H	
help	10, 11
Help	9
Hybrid systems	6
I	
Icon	26
Implicit integration	20

L

Labels	35
Language fig (abg.fig)	26
Language fig (cbg.fig)	50
Language fig (sabg.fig)	35
Language m (abg.m)	33
Language m (cbg.m)	50
Language m (view)	35
Language tex (abg.tex)	34
Language tex (desc.tex)	43
Language tex (struc.tex)	44
Language tools	64
Language txt (struc.txt)	44
Languages	63
LaTeX	66
lbl	35, 41
library	26

M

m	63
m-files	64
Make	60
Makefiles	60
Matlab	64
Menu-driven interface	7
MTT, purpose of	1
mtt.m	64
mtt2sys	66
mttrc	68

N

n_ports	33
Named SS	29
Named SS components	29
New representations	60
Numeric parameters	5, 46, 47

O

OCST	66
Octave	64, 66
Octave interface	64
Octave setup	68
ODE	54, 55

ode.m	56
ode.r	56
Old-style labels	41
Options	8
Ordinary Differential Equations	55
Ordinary Differential Equations (m)	56
Ordinary Differential Equations (reduce)	56
Ordinary Differential Equations (view)	57
Other component labels	37
Other component labels (old-style)	42

P

parameter aliases	39
Parameter passing	40
Parameter passing (old-style)	42
Parameters	46
paths	68
port aliases	39
Port label defaults	31
port labels	30
ports	5, 30
Predefined constitutive relationships	45

Q

Quick start	14
-------------	----

R

Reduce	63
REDUCE setup	68
rep	58
rep.txt	58
Report	58
Report (text)	58
Report (view)	59
Representation summary	23
representations	10
Representations	23
Representations, defining	23
Representations, what are they?	1

S

Sensitivity models	22
simple components	70
Simple components	28
Simple components - implementation	29
Simulation	18
Simulation initial state	20
Simulation input	20
Simulation output	21
Simulation parameters	19
Software components	67
SS component labels	36
SS component labels (old-style)	41
SS components	29
status	50
Steady-state solutions	19
Stripped acausal bond graph (sabg)	34
strokes	27
struc	44
Structure	44, 52
Structure (view)	44
Switched systems	6
Symbolic parameters	5, 46
Symbolic parameters for simplification	46

T

Text editors	64
toolbox	66
Top level	17
Transformation abg2cbg_m	51
Transformation abg2rbg_fig2m	33

Transformation cbg2ese_m2r	52
Transformation cse2ode_r	56
Transformation dae_r2m	54
Transformation dae2cse_r	55
Transformation ese2dae_r	54
Transformation ode_r2m	57
Transformation rbg2abg_m	34
Transformations	2

U

User interface	7
Utilities	9

V

valid name	32
variable declarations	38
Variables	3
Vector components	31
vector port labels	30
Verbal description (desc)	26
Version control	12
view Constrained-state Equations	35, 55
view Ordinary Differential Equations	57
view Report	59
view Structure	44
views	64

X

Xfig	64
------------	----

Table of Contents

1	Introduction	1
1.1	What is a representation?	1
1.2	What is a transformation?	2
1.3	What is a bond graph?	3
1.4	Variables	3
1.5	Bonds	4
1.6	Components	4
1.6.1	Ports	5
1.6.2	Constitutive relationship	5
1.6.3	Symbolic parameters	5
1.6.4	Numeric parameters	5
1.7	Algebraic loops	5
1.8	Switched systems	6
2	User interface	7
2.1	Menu-driven interface	7
2.2	Command line interface	7
2.3	Options	8
2.4	Utilities	9
2.4.1	Help	9
2.4.1.1	help representations	10
2.4.1.2	help components	10
2.4.1.3	help examples	10
2.4.1.4	help crs	11
2.4.1.5	help <name>	11
2.4.2	Copy	11
2.4.3	Clean	12
2.4.4	Version control	12
3	Creating Models	14
3.1	Quick start	14
3.2	Creating simple models	15
3.3	Creating complex models	17
3.3.1	Top level	17
4	Simulation	18
4.1	Steady-state solutions (odess)	19
4.2	Simulation parameters	19
4.2.1	Euler integration	19
4.2.2	Implicit integration	20
4.3	Simulation input	20
4.4	Simulation initial state	20
4.5	Simulation output	21

5	Sensitivity models	22
6	Representations	23
6.1	Representation summary	23
6.2	Defining representations	25
6.3	Verbal description (desc)	26
6.4	Acausal bond graph (abg)	26
6.4.1	Language fig (abg.fig)	26
6.4.1.1	Icon library	26
6.4.1.2	Bonds	27
6.4.1.3	Strokes	27
6.4.1.4	Components	27
6.4.1.5	Simple components	28
6.4.1.6	SS components	29
6.4.1.7	Simple components - implementation	29
6.4.1.8	Compound components	29
6.4.1.9	Named SS components	29
6.4.1.10	Coerced bond direction	30
6.4.1.11	Port labels	30
6.4.1.12	Vector port labels	30
6.4.1.13	Port label defaults	31
6.4.1.14	Vector Components	31
6.4.1.15	Artwork	31
6.4.1.16	Valid Names	32
6.4.2	Language m (rbg.m)	32
6.4.2.1	Transformation abg2rbg_fig2m	33
6.4.3	Language m (abg.m)	33
6.4.3.1	Arrow-orientated causality	34
6.4.3.2	Component-orientated causality	34
6.4.3.3	Transformation rbg2abg_m	34
6.4.4	Language tex (abg.tex)	34
6.5	Stripped acausal bond graph (sabg)	34
6.5.1	Language fig (sabg.fig)	35
6.5.2	Stripped acausal bond graph (view)	35
6.6	Labels (lbl)	35
6.6.1	SS component labels	36
6.6.2	Other component labels	37
6.6.3	Component names	37
6.6.4	Component constitutive relationship	38
6.6.5	Component arguments	38
6.6.6	Variable declarations	38
6.6.7	Aliases	38
6.6.7.1	Port aliases	39
6.6.7.2	Parameter aliases	39
6.6.7.3	Component aliases	40
6.6.8	Parameter passing	40
6.6.9	Old-style labels (lbl)	41
6.6.9.1	SS component labels (old-style)	41

	6.6.9.2	Other component labels (old-style)	42
	6.6.9.3	Parameter passing (old-style)	42
6.7		Description (desc)	43
	6.7.1	Language tex (desc.tex)	43
6.8		Structure (struc)	44
	6.8.1	Language txt (struc.txt)	44
	6.8.2	Language tex (struc.tex)	44
	6.8.3	Language tex (view)	44
6.9		Constitutive relationship (cr)	45
	6.9.1	Predefined constitutive relationships	45
		6.9.1.1 lin	45
		6.9.1.2 exotherm	46
	6.9.2	DIY constitutive relationships	46
6.10		Parameters	46
	6.10.1	Symbolic parameters (subs.r)	46
	6.10.2	Symbolic parameters for simplification (simp.r)	46
	6.10.3	Numeric parameters (numpar)	46
		6.10.3.1 Text form (numpar.txt)	47
6.11		Causal bond graph (cbg)	50
	6.11.1	Language fig (cbg.fig)	50
	6.11.2	Language m (cbg.m)	50
		6.11.2.1 Transformation abg2cbg_m	51
6.12		Elementary system equations (ese)	51
		6.12.0.1 Transformation cbg2ese_m2r	52
6.13		Differential-Algebraic Equations (dae)	52
	6.13.1	Language reduce (dae.r)	53
		6.13.1.1 Transformation ese2dae_r	54
	6.13.2	Language m (dae.m)	54
		6.13.2.1 Transformation dae_r2m	54
6.14		Constrained-state Equations (cse)	54
	6.14.1	Language reduce (cse.r)	55
		6.14.1.1 Transformation dae2cse_r	55
	6.14.2	Language m (view)	55
6.15		Ordinary Differential Equations	55
	6.15.1	Language reduce (ode.r)	56
		6.15.1.1 Transformation cse2ode_r	56
	6.15.2	Language m (ode.m)	56
		6.15.2.1 Transformation ode_r2m	57
	6.15.3	Language m (view)	57
6.16		Descriptor matrices (dm)	57
	6.16.1	Language reduce (dm.r)	57
	6.16.2	Language m (dm.m)	58
6.17		Report (rep)	58
	6.17.1	Language text (rep.txt)	58
	6.17.2	Language view	59

7	Extending MTT	60
7.1	Makefiles	60
7.2	New representations	60
8	Languages	63
8.1	Fig	63
8.2	m	63
8.3	Reduce	63
8.4	c	63
9	Language tools	64
9.1	Views	64
9.2	Xfig	64
9.3	Text editors	64
9.4	Octave	64
9.4.1	Octave control system toolbox (OCST)	66
9.5	LaTeX	66
10	Administration	67
10.1	Software components	67
10.2	REDUCE setup	68
10.3	Octave setup	68
10.3.1	.octaverc	68
10.4	Paths	68
10.4.1	\$MTTPATH	68
10.4.2	\$MTT_COMPONENTS	68
10.4.3	\$MTT_CRIS	69
10.4.4	\$MTT_EXAMPLES	69
10.4.5	\$OCTAVE_PATH	69
10.5	File structure	69
	Glossary	71
	Index	73