Macchiato

A Simple and Scriptable Petri Nets Implementation

Version 1-6-1

Advanced User Manual

Dr. Mark James Wootton

Macchiato [mak'kjaːto], *from the Italian, meaning* "spotted", "marked" *or* "stained", *in reference to a latte macchiato, which resembles a Petri Net place with a token.*

Contents

1 Introduction

Macchiato was created to fulfil a hole in the range of available software for Petri Net modelling, between those byzantine in their complexity and those too limited in scope for research purposes. To this end, Macchiato is designed to demand no more than is necessary for the task at hand. Users who wish simply to create a Petri Net and execute a batch of simulations will find that that they can do so easily via either textual or graphical means, according to their preference. For use cases requiring more elaborate solutions, Macchiato provides both versatile scriptablity and support for integration with physical simulations. The aim of this document is to give clear descriptions of all of these features and their usage, with examples given throughout.

The use of Macchiato for Petri Nets can been seen in the following publication:

- Mark James Wootton, John D. Andrews, Adam L. Lloyd, Roger Smith, A. John Arul, Gopika Vinod, M. Hari Prasad, Vipul Garg. Risk Modelling of Ageing Nuclear Reactor Systems, *Annals of Nuclear Energy*, volume 166, page 108701, 2022.

The FMU interface seen in §7, has been used in:

- Mark James Wootton, John D. Andrews, Ying Zhou, Roger Smith, A. John Arul, Gopika Vinod, M. Hari Prasad, and Vipul Garg. A Hybridised Petri Net-Pseudo Bond Graph Model for a Nuclear Reactor Coolant System. *Submission pending*.

Please consider referencing one or both of these papers as relevant in work in which Macchiato has been used.

2 Dependences

- Python 3 [1]
  - NumPy [2] – only required by analysis scripts
  - Matplotlib [3] – only required by analysis scripts
- Graphviz [4] – only required by visualisation features
- Microsoft Visio [5] – only required for graphical Petri Net construction tool

3 Installation

Download and extract the files provided to a convenient folder and make a note of its directory. One can configure their system to allow Macchiato to be accessed conveniently from any working directory within a terminal or Python session, such that simulations can be initiated with command `Macchiato` from any location, and that a Python script using the module need only include the line `import Macchiato`, regardless of the file path. The procedure to achieve this is dependant on operating system and shell environment.

3.1 Windows 10 & PowerShell

For users of PowerShell on Windows 10, the integration must be done in two steps.

3.1.1 PowerShell Fuction

To allow easy execution of Macchiato simulations from any directory, create a folder in `Documents` called `WindowsPowerShell`. In this new folder, add a plain text file with the name `Microsoft.PowerShell_profile.ps1` containing the lines:

```
# Execute Macchiato simulations from any directory
function Macchiato {
python C:\path\to\Macchiato\Macchiato.py $args
}
```

Be sure to use to change the path given to the real location of `Macchiato.py` on your computer.

3.1.2 Python Path

To make Macchiato available for import in any Python instance or script:

- Open *"Settings"* from the Start Menu and search for *"advanced system settings"*.
- You will see a result for *"View advanced system settings"*, see figure 1.
- Clicking on this option will open a window with the title *"System Properties"*.
- On the *"Advanced"* tab, there is a button labelled *"Environmental Variables"*, which brings up a window with the same name, see figure 2.
- Beneath the *"System variables"* section, click on *"New..."*, see figure 3.
- In the field *"Variable name"*, enter `PYTHONPATH` and in *"Variable value"* enter the directory of the folder in which Macchiato's files have been placed, see figure 4.

Click *"Ok"* to close the aforementioned windows and the changes will take effect in any new PowerShell instance.

Note: At the time of writing, Windows 11 has recently been annouced. This may necessitate some adaptation of the instructions in this section.

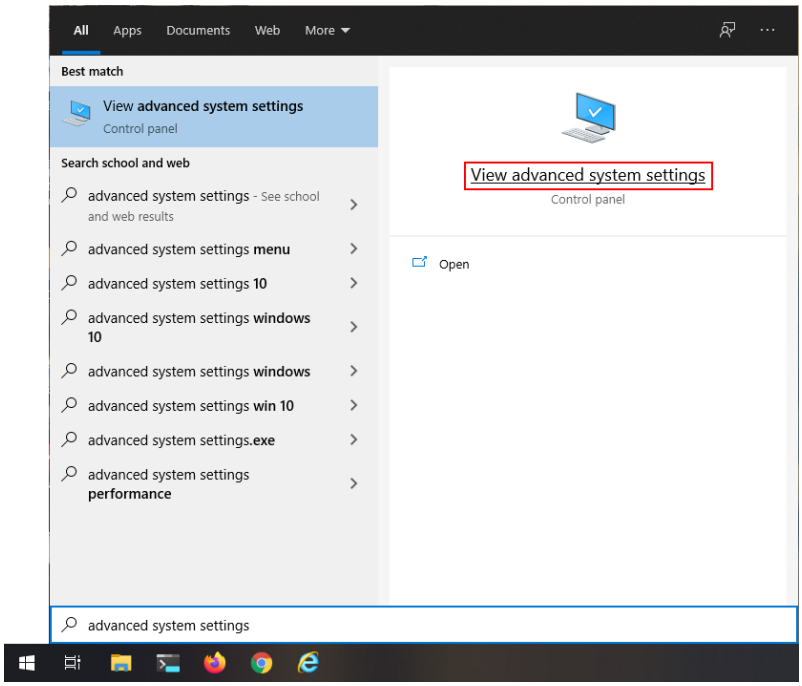


Figure 1: Open the start menu and search *"advanced system settings"*. Click on *"View advanced system settings"* when it appears.

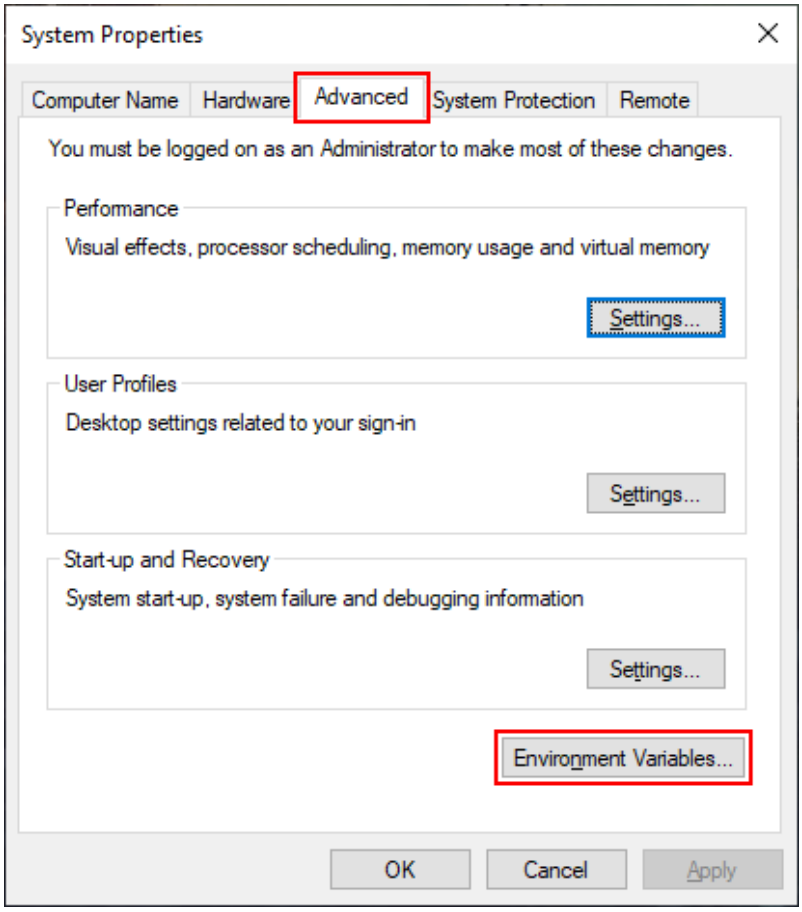


Figure 2: When the *"System Properties"* window appears, select *"Environment Variables"*, on the *"Advanced"* tab.

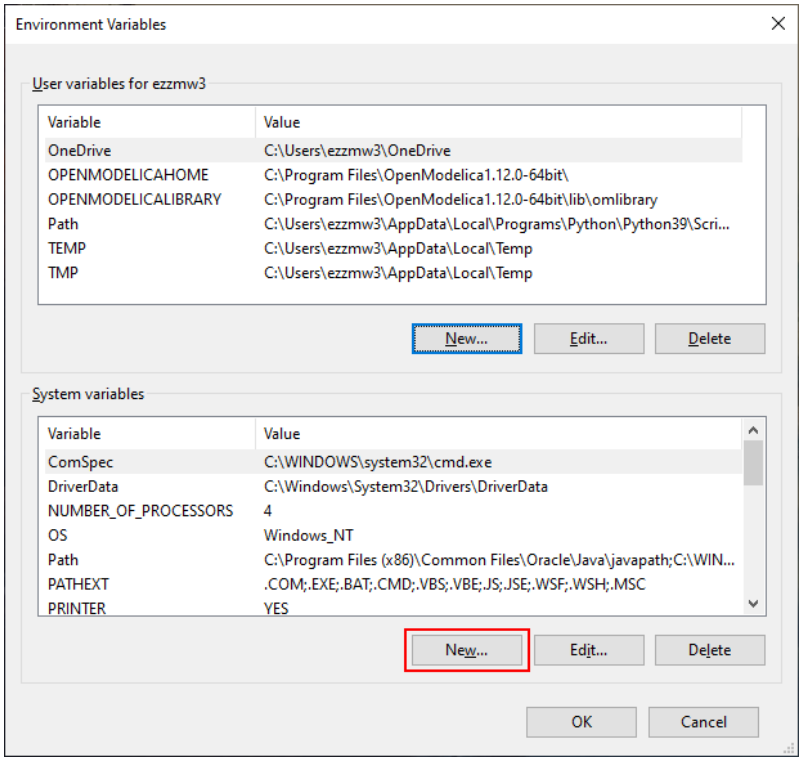


Figure 3: Add a new entry to the *"System variables"* list.

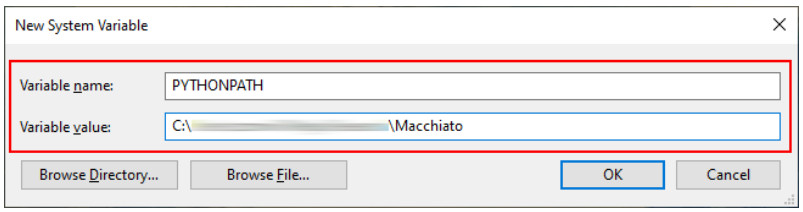


Figure 4: Give `PYTHONPATH` as the *"Variable name"* and the directory of the folder where Macchiato is located as the *"Variable value"*.

3.2 Bash & Z Shell (Linux, MacOS, Cygwin)

In a Bash or Z Shell environment, such as is default in many Linux distributions and in MacOS, add the following lines to the file found in the Home directory with the name `.bashrc` or `.zshrc` respectively:

```
# Execute Macchiato simulations from any directory
Macchiato(){
        python3 $HOME/path/to/Macchiato/Macchiato.py $*
}
# Import Macchiato in Python from any directory
export PYTHONPATH=$HOME/path/to/Macchiato:$PYTHONPATH
```

This assumes that Macchiato is located in a subdirectory of Home. If this is not the case, give the absolute path instead. Either way, make sure that the two paths above point to the actual location of `Macchiato.py` and its parent folder respectively.

The changes will take effect immediately in any new terminal instance. In an existing terminal, you can update via the command `source ~/.bashrc` or `source ~/.zshrc` as appropriate.

4 Graphviz

If the visualisation features are required, Graphviz must be available to call at the command line. It is recommended to install it via package manager. However, please note that Graphviz is entirely optional and the recommended method for producing figures is outlined in §6.2 *Graphical Petri Net Construction with Microsoft Visio*.

4.1 Windows

Windows users on PowerShell will need to install an appropriate command line installer first, for example Scoop [6], from which Graphviz is installed by:

```
> scoop install graphviz
```

To complete the process on many Windows installations it is necessary to then run the following command in a PowerShell instance with administrative privileges.

```
> dot -c
```

It is also possible to download an installer from Graphviz's website. This continues to be available as a legacy option, in which case the directory containing the executable dot.exe must be manually specified as the value of the docLoc parameters. This method is not recommended.

4.2 Linux

Graphviz is available through the package managers of most Linux distributions.

`apt` (Debian, Ubuntu, Linux Mint, Pop!_OS, elementary OS, etc.):

```
$ sudo apt install graphviz
```

`yum` (Fedora, CentOS, RHEL, Oracle Linux, etc.):

```
$ sudo yum install graphviz
```

`pacman` (Arch, Manjaro, etc.):

```
$ sudo pacman -Syy graphviz
```

4.3 MacOS

On MacOs install Graphviz via Homebrew [7], via:

```
% brew install graphviz
```

5 Petri Net Integration Algorithm

The flowchart in figure 5 depicts the process followed by Macchiato to execute an individual Petri Net simulation.

6 Usage

Macchiato Petri Net structures are stored in `*.mpn` files, the creation of which is discussed in §6.1 *Macchiato Petri Net Files (`*.mpn`)*.

Assuming the instructions in §3 *Installation*[†] have been followed, the following command will run a batch of simulations, where `{nSims}` should be replaced with the desired number of iterations:

```
$ Macchiato /path/to/PetriNet.mpn {nSims}
```

Note that regardless of the locations of Macchiato or the Petri Net file, the simulation output will be delivered within the current working directory. If `{nSims}` is omitted from the above command, the simulations will continue until the total time simulated across all iterations reaches or exceeds the product of **`maxClock`** and **`simsFactor`**, see §6.1.2 *Simulation Parameters*. Additional terminal output can be activated by placing `-v` or `--`**`verbose`** at the end of the above command, but be aware that this will negatively impact performance.

If it is necessary to restrict the output in the relevant `*.cvs` files produced to a limited selection of places or transitions, this can be done by adding the command line flags `-p` and `-t`, for example:

```
$ Macchiato /path/to/PetriNet.mpn -p P0 P1 P2 -t T0 T1 T2
```

The help text is displayed by:

```
$ Macchiato --help
```

6.1 Macchiato Petri Net Files (`*.mpn`)

A Petri Net description in `*.mpn` format may be created textually, as outlined in the following subsections, or via the tools seen in §6.2 *Graphical Petri Net Construction with Microsoft Visio*. One may also create and manipulate Petri Net structures in a Python script using the tools provided by the `Macchiato.py` module, as documented in §6.3 *Scripting Tools*.

It is assumed that the reader is already familiar with the basics of standard Petri Net modelling [8, 9].

6.1.1 Structure

An `*.mpn` should be comprised of three sections – the simulation parameters, the places list, and the transitions list. To mark the beginning of of the latter two, the lines **`Places`** and **`Transitions`** must respectively appear in the file. Any input on line after a `#` character is treated as a comment and ignored. An example file is seen in §6.1.7 *Example*.

[†]If not, replace `Macchiato` with `python /path/to/Macchiato.py`

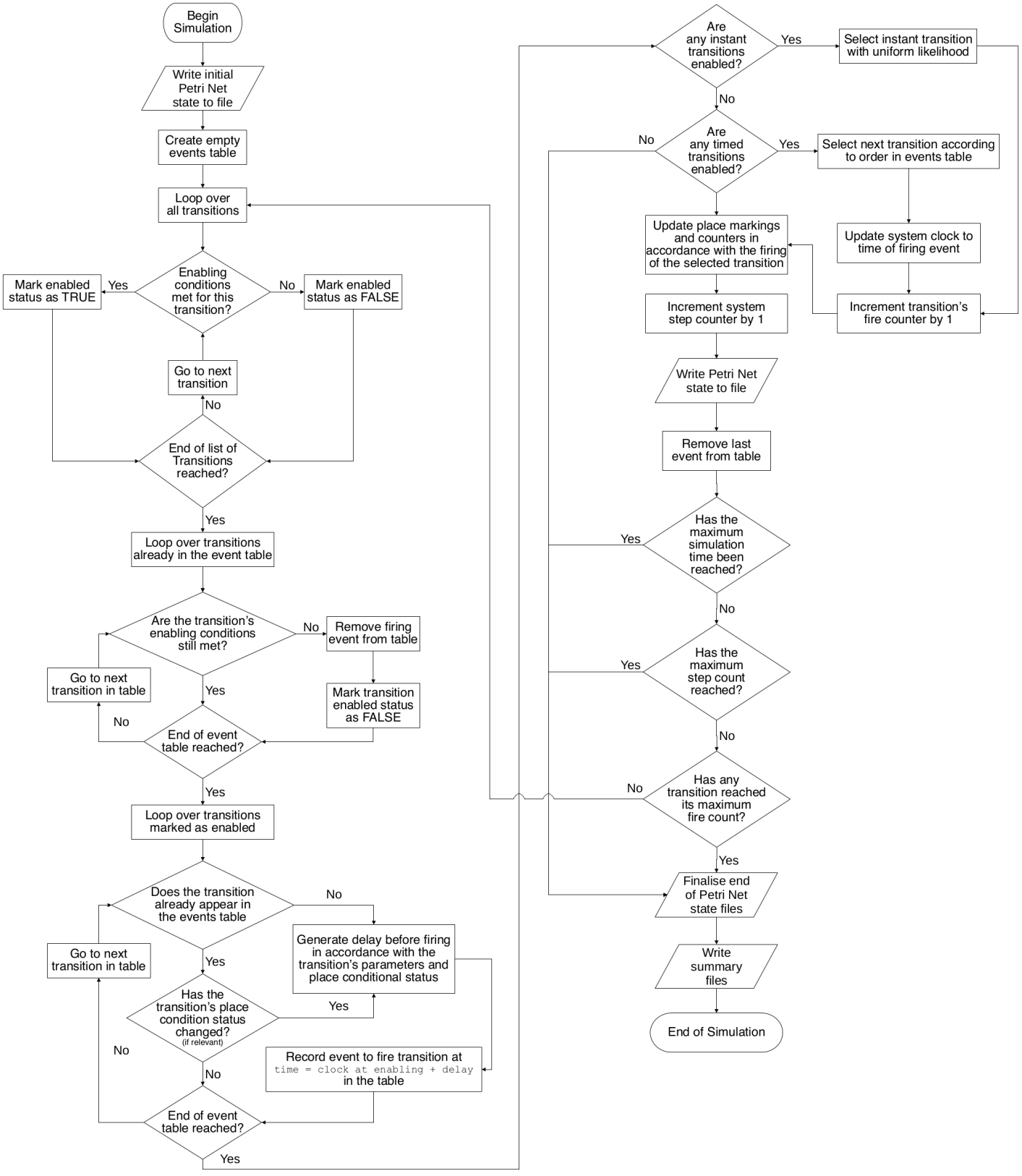


Figure 5: Algorithm for the integration of Petri Nets by Macchiato.

6.1.2 Simulation Parameters

`name` The label given to the Petri Net and used in output directories

`unit` The units of time to be used by the Petri Net (Default is `hrs`)

`runMode` The mode of integration to be used for simulation (Default is `schedule`). Changing this parameter is not recommended.

`dot` Toggle creation of snapshots of the Petri Net during simulation in `*.dot` format (Default is `False`).

`visualise` The file format for images produced from snapshots. Supported formats include, but are not limited to, `svg` (recommended), `pdf`, and `png` (Default is `None`, which produces no images. Note that dot must also be set to `True`, otherwise `visualise` will have no effect).

`details` Toggles label with Petri Net name, step, and clock in visualisations (Default is `True`)

`useGroup` Toggles use of place and transition groups in visualisation (Default is `True`)

`orientation` Orientation of Petri Net in visualisations. Options are `LR`, `RL`, `TB`, and `BT` (Default is `None` — produces default behaviour of Graphviz algorithm).

`debug` Default is `False`

`maxClock` A simulation is terminated if its clock exceeds this parameter (Default is 10^6 **`units`** of time). Be aware that this does not mean that a simulation cannot exceed this value. If a simulation with **`maxClock`** set to 10 has a clock value of 9 and its next transition due to fire after a delay of 5, it will reach a clock value of 14 on the next step, at which point the simulation will end, having crossed the **`maxClock`** threshold. If simulations of an exact maximum duration are required, use a fixed delay transition set to terminate a simulation when it has fired once.

`maxSteps` Greatest number of steps permitted in any one simulation (Default is 10^{12})

`simsFactor` Parameterises the total number of simulations performed (Default is 1.5×10^3). Repetition of simulations ends once the total simulated time surpasses the product of **`maxClock`** and **`simsFactor`**. If a set number of simulations is specified at the command line, **`simsFactor`** is overruled.

`dotLoc` Directory containing `dot.exe` for legacy mode visualisations – not recommended (Default is `None`).

Important Note: It is not recommended to use the visualise option beyond testing and development of Petri Nets and performance is significantly

affected. Instead, consider using the tools provided by `mpn_to_dot.py` and `dot_to_image.py` after the simulations are complete, see §6.6 *Visualisation*. If one is not intending to use `dot_to_image.py`, then it is also recommended to set **`dot`** to `False`.

6.1.3 Places

The places section of the file begins following the line **`Places`**. To add a place, simple add a new line with the desired name (spaces are not permitted). By default, a place's initial token count is zero, but a value may be specified after a space on the same line, e.g. `P1 2` will add a place of name *P1* with two tokens at the start of a simulation.

6.1.4 Transitions

The transitions section of the file begins following the line **`Transitions`**. A transition and its properties are specified with a line with the following format (do not include the brackets):

```
{Name}:{Timing}:{Parameters} IN {places} OUT {places} VOTE
   {threshold} RESET {places}
```

`{Timing}` specifies the duration from the enabling of a transition to its firing, which may be instantaneous, of a fixed length, or generated from a stochastic distribution. Most options require one or multiple parameters, subsequently delimited by `:` and expressed in terms of the parameter **`units`** where relevant. The following timing options are available:

`instant` A instant transition will fire on the next simulation step with zero advancement of the system clock. If multiple instant transitions are simultaneously enabled, one will be chosen at random with uniform weight.

`delay:a` A transition with a fixed delay fired after a set duration `a` once enabled.

`uniform:u` A transition with a uniform distribution will fire at some time, t, in the interval $0 < t <$ `u`.

`cyclic:c:ω` A cyclic transition fires at the next instance at with the simulation clock is a non-zero integer multiple of `c`. The second parameter `ω` allows one to apply an offset. For instance, if two transitions with the parameters **`cyclic`**`:1:0` and **`cyclic`**`:1:0.5` are persistently enabled, they will respectively fire at the system times, 1, 2, 3 units etc, and 1.5, 2.5, 3.5 **`units`** etc.

`weibull:<t>:β:σ` A transition with this option will be characterised by a Weibull distribution [10] with mean `<t>` and shape parameter `β`. Its firing time, t, is given by $t = \eta\left[\ln(X)\right]^{-\beta}$ where η is the scale parameter, such that $\eta = \langle t \rangle \left[\Gamma\left(\beta^{-1}+1\right)\right]^{-1}$, and X is a random variable uniformly

distributed in the range $0 < X < 1$, with Γ being the Gamma Function [11]. The parameter `σ` is optional and is used when the mean time has an associated uncertainty, such that the scale parameter used for each firing delay calculated is produced from a normal distribution [12] with mean equal to the default η and a standard deviation of `σ`.

`lognorm:μ:σ` A transition with log-normal distribution [13] timing fires after time, t, where $t = \exp(\mu + \sigma X)$, with X being a standard normal variable, and `μ` and `σ` respectively being the mean and standard deviation of the natural logarithm of the firing delay.

`rate:r` A transition of this type fires with a constant rate parameterised by `r`. Firing time, t, is exponentially distributed [12], such that $t = -r^{-1}\ln(X)$, where X is a random uniform variable in the range $0 < X < 1$.

`beta:p:q:k` A transition with the Beta Distribution [14] produces a firing delay, t, in the interval $0 < t < 1$, parameterised by `p` and `q`, which weight the probability density towards the extreme or central regions of the available outcome space. An optional parameter `k` can be added to scale the distribution, such that the range of possible values becomes $0 < t <$ `k`.

Note that a transition must be continuously enabled for the duration from firing time generation until it fires. If its enabled status is interrupted, its scheduled firing time will be discarded.

6.1.5 Arc Properties

Any places listed after **`IN`** and **`OUT`** will be connected to the transition by incoming and outgoing arcs respectively and places listed should be separated by a single space. The weight of an arc is 1 by default and can be given some other value by appending it, separated by `:`, to the name of the relevant place in the list. for example, **`IN`** `P1 P2:3 P3` **`OUT`** `P4 P5:2` specifies three incoming arcs, the second of which has a weight of 3, and two outgoing arcs, the latter of which has a weight of 2.

An incoming arc may be designated as a place conditional or inhibit arc with the code **`:pcn`** or **`:inh`** respectively, placed after the arc weight. The action of an inhibit arc is simple to disable its target transition when its weight is met, regardless of the status of the other arcs. A place conditional arc does not enable or disable its target transition but instead modifies its firing time parameters. The modification factor for a transition with C place conditional arcs is a function of the arc weights (which can be non-integer for place conditionals) and the number of tokens on the connected places, such that, $P = 1 + \sum_{i=1}^{C} W_i N_i$. The alterations to parameters are then, $a \to {}^{a}/_{P}$, $u \to {}^{u}/_{P}$, $c \to {}^{c}/_{P}$, $\langle t \rangle \to {}^{\langle t \rangle}/_{P}$, $\mu \to {}^{\mu}/_{P}$, $k \to {}^{k}/_{P}$, and $r \to rP$.

6.1.6 Additional Transitions Features

Transitions may also be given the properties **`VOTE`** and **`RESET`**. A voting transition does not require all of its incoming arc weights to be met to become enabled. Instead, only a given threshold need be met, placed after **`VOTE`**,

separated by a single space. For example, a transition `T1` with the place relationship given by `T1:instant IN P1 P2 P3 OUT P4 VOTE 2` would only require two of `P1`, `P2`, and `P3` to hold a token in order to fire. Note that all incoming arcs whose weight is satisfied are treated normally for the purposes of removing tokens when the transition fires. A reset transition has an associated list of places, delimited by `:` and following `RESET`, separated by a single space, e.g. `RESET P1:P2:P3`. When the transition fires, the places marked for reset are restored to the token count held at the beginning of the simulation.

6.1.7 Example

Below is an example `*.mpn` file to demonstrate the usage of the key features of Macchiato. Indentation or any initial white space will be ignored, meaning that the user can arbitrarily align the contents the input file, such as is most convenient in clearly outlining the structure of the Petri Net.

```
# Petri Net Parameters
name Test
units hrs
runMode schedule
visualise None
dot False

# Run Parameters
maxClock 1E3
maxSteps 100
simsFactor 1

# Build Petri Net
Places
        P0 2
        P1
        P2
        P3

Transitions
        T0:lognorm:1:1 IN P0 OUT P1 P3
        T1:weibull:1:0.5 IN P1 OUT P2:2
        T2:delay:2 IN P2:2 P3:inh OUT P1
        T3:rate:15 IN P3:5:pcn P1 OUT P2
        R:cyclic:7:1 IN P2 RESET P0:P1:P3
        V:beta:1:2:0.25 IN P0 P1 P3 OUT P2 VOTE 2
```

6.2 Graphical Petri Net Construction with Microsoft Visio

As an alternative to transcribing a Petri Net structure by hand, it is possible to graphically construct a model in Microsoft Visio and export as an `*.mpn` file. This is general considerably simpler and more expedient, and has the secondary benefit of concurrently producing high quality figures for reports and

publications.

In the directory PetriNetDrawingTools, one will find a Microsoft Visio drawing and a stencil file. To begin, make a copy of the the example file, which contains the macro that exports the Petri Net. A copy of the stencil file must be saved in the same directory or imported from the repository itself. Only objects from this stencil should be used.

When the example file is opened, a small banner will appear, asking the user whether macros should be enabled, as seen in figure 6. Click *"Enable Content"* as otherwise it will not be possible to export the Petri Net.

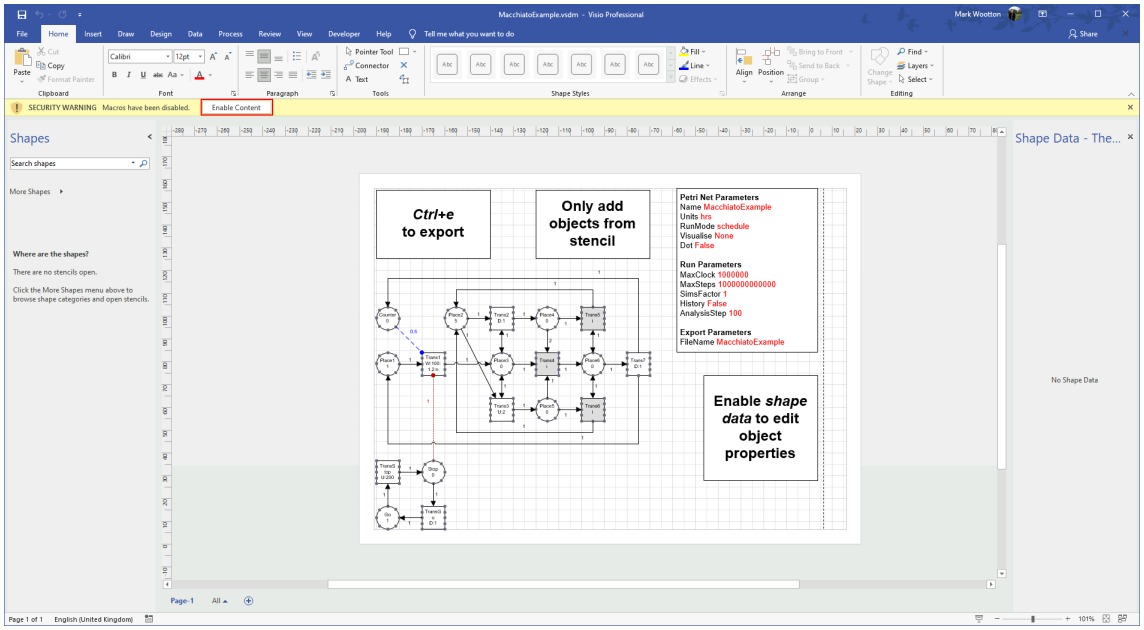


Figure 6: The example Petri Net as viewed in Microsoft Visio 2019. Remember to allow macros by clicking *"Enable Content"* on the yellow warning message bar.

If the Macchiato stencil is not already visible in the *"Shapes"* panel, import `MacchiatoStencil.vssx` via *"More Shapes"* → *"Open Stencil"*, see figure 7.

Make sure that "Shape Data Window" is enabled in the "Data" tab.

The *"Shape Data"* panel, shown in figure 9, is used to set the parameters for each object in the model, including the system parameters block.

New objects are added, either by dragging shapes from the stencil, or by copying existing ones. Particularly in the latter case, make sure that every place and transition has a unique name. The formatting of the objects can be changed freely. Likewise, the text of their labels can be edited as required or desired, but be aware that if the content pertaining to its properties is altered, the field in question will no longer auto-update with respect to later changes.

Connections must be made using the arc objects and not the default Visio connectors. The arcs are connected by dragging their end points onto the places and transitions, either linked to the centre or to an attachment point, as shown in

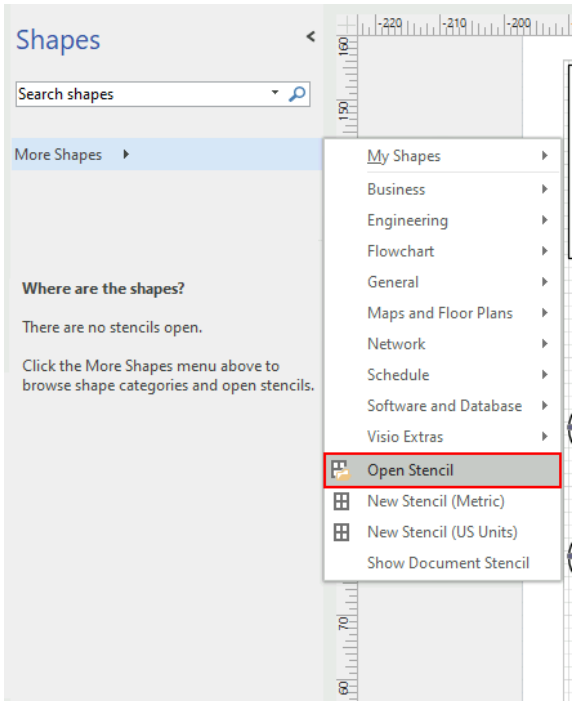


Figure 7: If not already visible, open the Macchiato stencil via the *"More Shapes"* menu, found on the left of the screen by default.

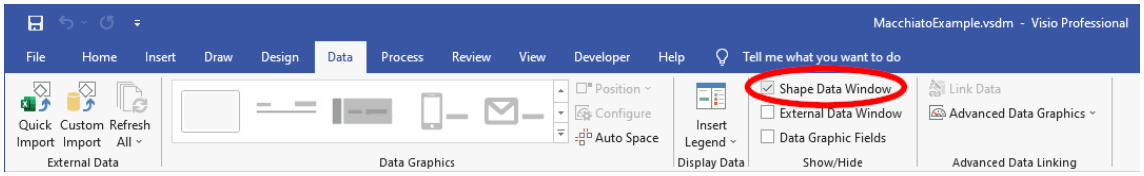


Figure 8: *"Shape Data Window"* must be enabled to edit Macchiato object parameters.

figure 10.

Additional points can be added from the *"Home"* tab, circled on figure 11. However, be certain that the correct object is selected as Visio will allow the user to attach a connection point associated with the currently selected object to any shape in the file, potentially causing failed or erroneous `*.mpn` export. The *"Shape Data"* panel, shown in figure 11, is used to set the parameters for each object in the model, including the system parameters block.

The model is exported to an `*.mpn` file when the key combination *"ctrl+e"* is pressed. The output is saved in the same directory as the source file, with the name specified in the system parameters object, see figure 12. Existing files will be overwritten, with no warning issued, so be careful not to unintentionally destroy work.

6.3 Scripting Tools

The Python module provided by `Macchiato.py` can be imported into a script to provide a range of tools suitable for more complex creation and manipulation of Petri Net models, as well as basic functions such as

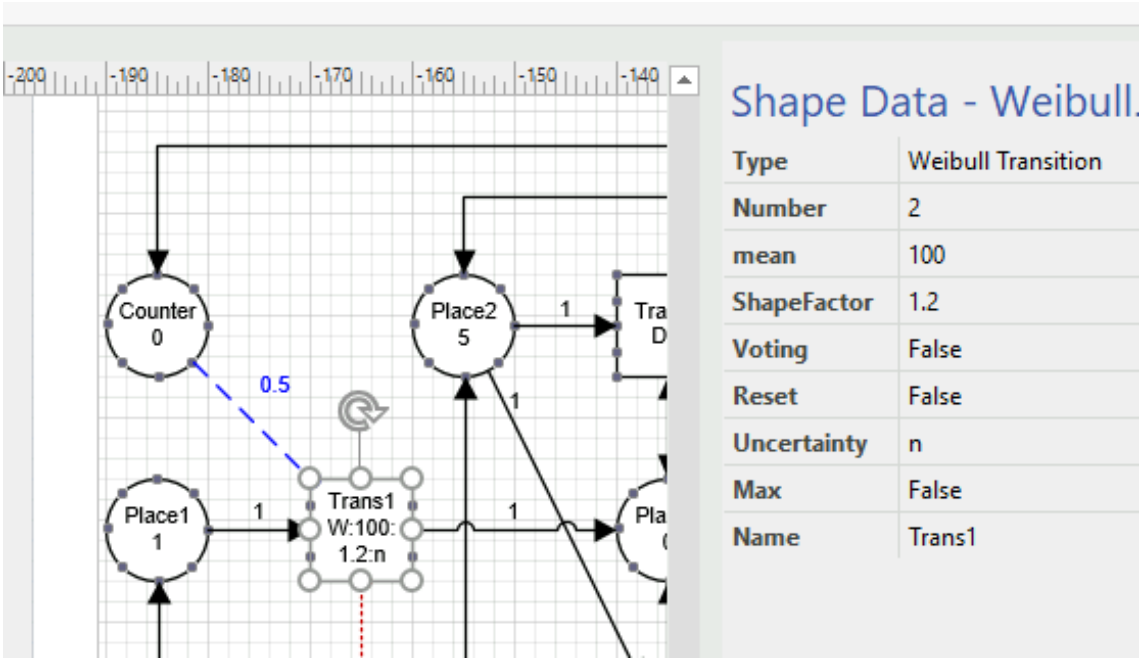
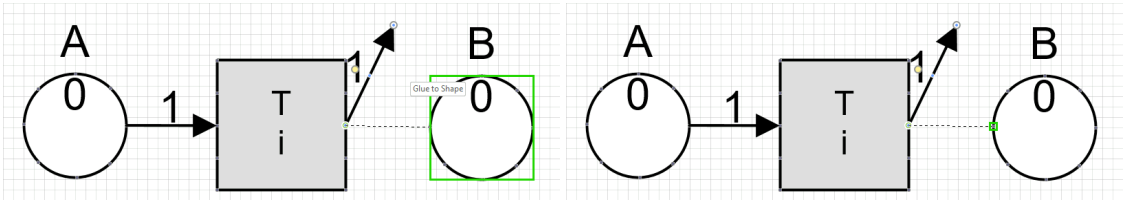


Figure 9: Object parameters are edited via the *"Shape Data"* window.



(a) Gluing an arc to an object (b) Affixing an arc to a connection point

Figure 10: An arc can be either glued to an object, which case it will align to its centre, or be affixed to a connection point on its edge.

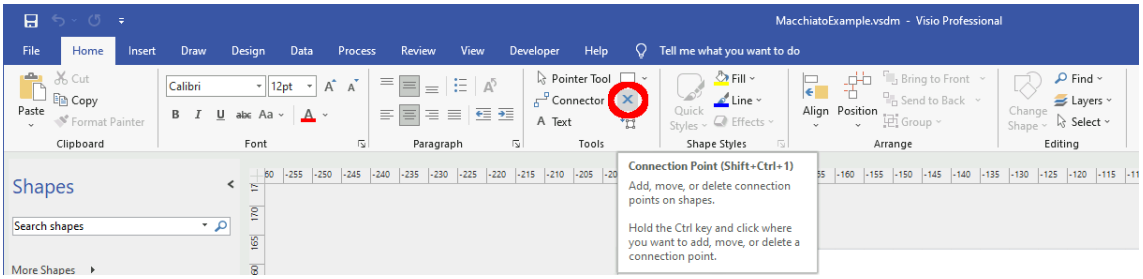


Figure 11: Connection points can be added via the circled button, or by pressing *"Ctrl+Shift+1"*.

reading and writing from `*.mpn` files. For example, if one wished to create a range of similar systems with varying parameters or varying number of duplicated sections, the scripting tools would enable the automation of this process. Documentation for these features is found either by running `help(Macchiato)` (or `help(Macchiato.SomeObject)` for a specific item) from within an interactive Python session, or by viewing the module files themselves, with descriptions provided for each object type, method, and

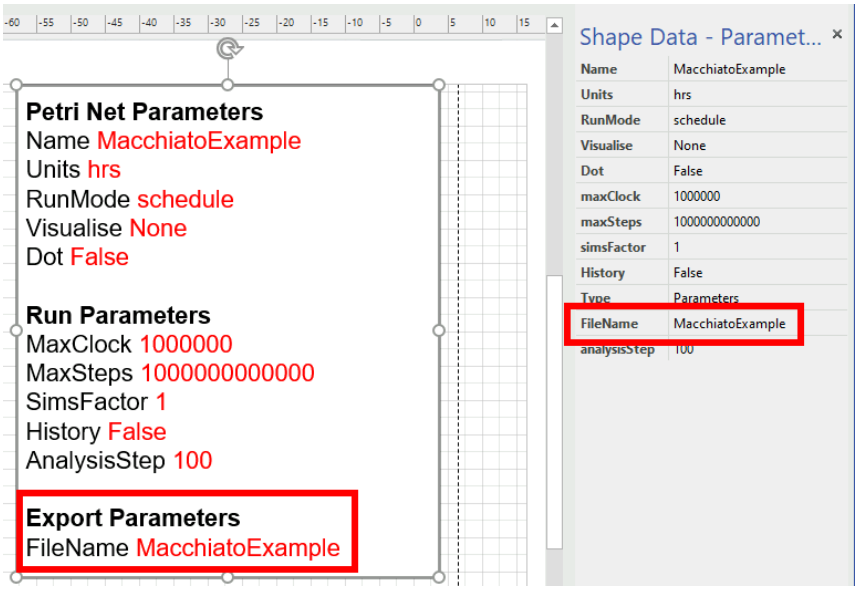


Figure 12: The file name used when the Petri Net is exported is set in the Parameters object.

function. A few examples are found below.

6.3.1 Reading and Writing `*.mpn` Files

Importing `Macchiato.py` gives access to the `read` and `write` functions, which allow the handling of `*.mpn` files. The function read takes a file path and returns a `PetriNet` object and list object containing the simulation parameters in the file.

```
# Import module
import Macchiato as MC

# Read *.mpn file
pn, simParams = MC.read('/path/to/PetriNet.mpn')
```

Conversely, `write` takes a `PetriNet` object and writes it to an `*.mpn` file.

```
# Import module
import Macchiato as MC

# Creat a Petri Net
pn = MC.PetriNet()

# Create Petri Net stucture
# ...
# ...
# ...

# Write Petri Net to *.mpn file
MC.write(pn, overwrite=False, rp=None, altName=None)
```

Note the parameters `overwrite`, `rp`, and `altName` are optional, and respectively control whether existing files can be overwritten (default value is `False`), a `list` object containing the simulation parameters written to the file (default value is `None`, resulting in the default parameters being used, see §6.1.2 *Simulation Parameters*), and the name given to the Petri Net in the file produced (default value is `None`, which results in no change in name).

6.3.2 Manipulating Petri Nets

To create a `PetriNet` object from scratch, import the module `Macchiato.py` and initialise an instance of the class, with any parameter not provided by the user defaulting to the value shown in the following example (a value should ideally always be given for `name`).

```
# Import module
import Macchiato as MC

# Create instance of PetriNet object (with default values
   shown - these may be ommitted)
pn = MC.PetriNet(name=None, units='hrs',
                 runMode='schedule', dot=False,
                 visualise=None, details=True,
                 useGroup=True, orientation=None,
                 debug=False, dotLoc=None)
```

The structure of a `PetriNet` object created by either of methods discussed may be constructed or edited by the methods, `addPlace`, `rmvPlace`, `addTrans`, and `rmvTrans`. The `PetriNet` object has the attributes `places` and `trans` (both of type `list`) which store objects of the class `Place` and `Trans` respectively. The `Trans` object has the methods `addInArc`, `rmInARc`, `addOutArc`, and `rmOutARc`, to link and unlink it from `Place` objects via `Arc` objects. In the following example, a simple loop is created from two places and two transitions.

```
# Import module
import Macchiato as MC

# Create instance of PetriNet object
pn = MC.PetriNet(name='Example')
# Add places
pn.addPlace('P1', tokens=2)
pn.addPlace('P2')
# Add transition T1 and its arcs
pn.addTrans('T1', weibull=[100000, 1.2])
pn.trans['T1'].addInArc('P1')
pn.trans['T1'].addOutArc('P2')
# Add transition T2 and its arcs
pn.addTrans('T2', delay=48)
pn.trans['T2'].addInArc('P2', weight=2)
```

```
pn.trans['T2'].addOutArc('P1', weight=2)
```

The optional parameters for `addPlace` are:

`tokens` Type: `int`. The initial number of tokens held (Default is 0)

`min` Type: `int`. Minimum limit for tokens held by the place (Default is None, i.e. no lower limit)

`max` Type: `int`. Maximum limit for tokens held by the place (Default is None, i.e. no upper limit)

`limits` Type: `list` of length two, containing `int` types. If the number of tokens held by the place is found to be less than the first entry or greater than the second, a simulation of the Petri Net is terminated.

`group` Type: `int`. Grouping label for Graphviz

The optional parameters for `addTrans` are:

- A timing parameter, pick one of the following or omit it to produce an instant transition:

  `rate` Type: `float`. Mean rate of fire per unit time description

  `uniform` Type: `float`. Uniformly distributed in 0 to `uniform`

  `delay` Type: `float`. Value of fixed delay

  `weibull` Type: `list` of `float` types. Two or three of parameters for a Weibull distribution (`<t>` and `β`, and optional uncertainty parameter `σ`).

  `beta` Type: `list` of `float` types. Two parameters for Beta distribution (`p` and `q`) and optional scale parameter (`k`).

  `lognorm` Type: `list` of `float` types. Two parameters for log-normal distribution (`μ`) and (`σ`).

  `cyclic` Type: `list` of `float` types. Two parameters for cyclic distribution (`c` and `ω`).

`maxFire` Type: `int`. Maximum number of times the transition can fire before the simulation terminates

`reset` Type: `list` of `str` types. The labels of places which are reset when the transition fires

`vote` Type: `int`. Voting threshold for the transition (Default is `None`, which results in no voting behaviour)

`group` Type: `int`. Grouping behaviour for Graphviz

The optional parameters for `addInArc` are:

weight Type: `int`. The weight of the arc (Default is 1)

type Type: `str`. The type of the arc. The available options are:

- **'std'** a normal arc (Default option)
- **'inh'** an inhibit arc
- **'pcn'** a place conditional arc

There is one optional parameter for `addOutArc`:

weight Type: `int`. The weight of the arc (Default is 1)

The methods `rmvPlace`, `rmvTrans`, `rmInArc`, and `rmOutArc` all remove objects of the corresponding type from the parent object (a `PetriNet` for `Place` and `Trans`, a `Trans` for `Arc`). They all only take only one argument, i.e. a `str` containing the relevant object label, which is the label of the object itself for `Place` and `Trans`, but is the label of the connecting `Place` object for `Arc` objects. Removing a place or transition from the Petri Net also removes all of its associated arcs.

The `PetriNet` object has the method `run` which will simulate the Petri Net specified up to a given maximum number of steps (`int`), and optionally, a maximum simulated time (`float`). In the example below, a `PetriNet` object, `pn`, is to be run for maximum of 100 steps or a maximum simulated time of $5{\times}10^5$ `units`.

```
pn.run(100, maxClock=500000)
```

This makes it possible to run any arbitrary code between steps for highly customisable simulation scripts. For example:

```
# Import modules
import Macciato as MC

# Create instance of PetriNet object
pn = MC.PetriNet(name='Example')
# Add places
pn.addPlace('P1', tokens=2)
pn.addPlace('P2')
# Add transition T1 and its arcs
pn.addTrans('T1', weibull=[100000, 1.2])
pn.trans['T1'].addInArc('P1')
pn.trans['T1'].addOutArc('P2')
# Add transition T2 and its arcs
pn.addTrans('T2', delay=48)
pn.trans['T2'].addInArc('P2', weight=2)
pn.trans['T2'].addOutArc('P1', weight=2)

# Write initial structure to *.mpn file
```

```
MC.write(pn, altName='%s_start'%pn.name)

# Write initial Petri Net marking to file
pfile, tfile, tlist = pn.writeNetStart(pn.runMode)

# Begin simulation process
while True:
    pn.run(1, verbose=False, fileOutput=False)
    pn.writeNet(pfile, tfile, tlist, pn.runMode)
    if fooCondition:
        # Modify the Petri Net or run some other code
        fooProcess(pn)
        # Record changes to the marking of the Petri Net
        if markingAltered:
            pn.step += 1
            pn.clock = newTime # if relevant
            pn.writeNet(pfile, tfile, tlist, pn.runMode)
    else:
        # Conditions for terminating simulation
        # ...
        break

# Finalise output files
pn.placesSummary(pn.runMode, tOut=True, pfile=pfile)
pfile.close()
tfile.close()
tlist.close()

# Write final structure to *.mpn file
MC.write(pn, altName='%s_end'%pn.name)
```

6.4 Analysis

Four Python scripts are available in the `Analysis` directory to aid in the extraction of results from Petri Net simulations, with a fifth providing some graphing tools. As the data produced by Macchiato is saved in `*.csv` format, it is fairly simple to produce new analysis tools and users are encouraged to do so.

6.4.1 `OutcomesData.py`

This script will provide information on the proportion of simulations ending in particular outcomes and the average durations of those sets, with standard error [15] given, as well as the 10^{th} and 90^{th} percentiles. This is achieved by inspection of the final states of a given list of places, with with labels delimited by `:`, e.g. `P1:P2:P3`. The script will also produce a set of image files containing histograms to represent the results, with the axis labelled *"Duration"* taking the same units as those specified in the simulated Petri Net. A separate file is also produced containing the simulation final clock values sorted according to their outcome(s).

Example:

```
$ python /path/to/OutcomesData.py /path/to/Results_Folder
   P1:P2:P3
```

6.4.2 `TransFireData.py`

This script produces statistics for transition firings, with standard error [15] given. Simply provide the directory containing the results for inspection and a plain text file will be produced in the current working directory.

Example:

```
$ python /path/to/TransFireData.py /path/to/Results_Folder
```

6.4.3 `ExtractPlacesEndings.py`

This script will extract the final state of every instance of a given set of places (delimited by `:`) within a directory and write them to file.

Example:

```
$ python /path/to/ExtractPlaceEndings.py /path/to/
   Results_Folder P1:P2:P3
```

6.4.4 `Places_wrt_Time.py`

This script will give the average number of tokens in each of a given set of places, plus standard error, sampling the simulation results at a user specified time interval. The total number of simulations continuing to run up to that point is also given. Image files containing graphs to illustrate these results are produced.

Example:

```
$ python /path/to/Places_wrt_Time.py /path/to/
   Results_Folder max_time interval P1:P2:P3
```

where `max_time` is the greatest time up to which the script will sample, `interval` is the gap between samplings, and `:` delimits the list of places to sample given at the end.

6.5 `HistogramTime.py`

This script will create histograms using the output file `{NAME}_OutcomeTimes.csv` from `OutcomesData.py`. It takes the name of this file as a command line argument.

Example:

```
$ python /path/to/HistogramTime.py /path/to/{NAME}
   _OutcomeTimes.csv
```

The fist entry of each line will be used as the title for the resulting graph, allowing the user to change it from the default (i.e. the label of the place) to something more verbose and human readable. However, take care when doing this if using a spreadsheet application, as the number of columns may exceed its maximum. Therefore, it is often safer to rename the entries using a text editor program.

If a number is provided as an additional option, this serves as a cut-off value, i.e. simulations end past this point will be excluded. Be aware that this value should be given in years and an assumption is made that the simulation units are in hours.

Example:

```
$ python /path/to/HistogramTime.py /path/to/{NAME}
   _OutcomeTimes.csv 1
```

If a percentage is given instead, this will be used as an upper cut-off for the proportion of simulations.

Example:

```
$ python /path/to/HistogramTime.py /path/to/{NAME}
   _OutcomeTimes.csv 50%
```

6.6 Visualisation

In the `Visualisation` directory, two scripts are available to produce images of Petri Nets from `*.mpn` files or Macchiato simulation output. These depend on Graphviz and are best suited for inspection and verification rather instead of creating report quality images. or this purpose, a dedicated graphical tool such as Microsoft Visio is better suited, see §6.2 *Graphical Petri Net Construction with Microsoft Visio*. By default, Graphviz will attempt to enforce a hierarchical structure on the Petri Net visualisation, but this is unsuitable in many cases, particularly with systems with multiple looping sequences. To compensate for this, one can add a place or transition to a grouping, which will force objects to appear next to those of the same assignment. This is achieved through the addition of the label GROUP to the end of the line on which the object is specified followed by a space and an integer, which serves as its group reference. Note that places and transitions have separate groupings, i.e. the places and transitions in `P1 GROUP 1`, `P2 GROUP 1`, `T1:instant IN P1 OUT P2 GROUP 1`, and `T2:instant IN P2 OUT P1 GROUP 1` will be organised as two independent groups.

6.6.1 `mpn_to_dot.py`

This script will produce a single `*.dot` file (readable by Graphviz) depicting a Petri Net in its initial state, as described in an `*.dot` file. Replacing `PetriNet.mpn` with the path of the target `*.mpn` , it is invoked with the following command:

```
$ python /path/to/mpn_to_dot.py /path/to/PetriNet.mpn
   {format1} {format2} {format3}
```

If multiple image file formats are supplied a file of each of the given types will also be produced. Formats should be specified by their file extension only, e.g. `svg` or `eps`.

Note: For best results, it is highly recommended to use vector image supporting formats (e.g. `svg`, `eps`, `pdf`) instead of raster images (e.g. `png`, `jpg`, `gif`, `bmp`).

6.6.2 `dot_to_image.py`

This script will read a `*.dot` file, or a directory of `*.dot` files, and produce image files of types from the given list of formats. Substituting `/path/to/target` for the directory or file to be read, the script is executed with the following command, with the desired image formats listed at the end:

```
$ python /path/to/dot_to_image.py /path/to/target
   {format1} {format2} {format3}
```

7 FMU Interface

Macchiato Petri Nets can also be used in conjunction with a physical model provided in FMU format [16], the tools for which are provided in the `FMUInterface` directory.

7.1 PN-FMU Dependencies

- Python 3 [1]
  - Macchiato
  - PyFMI [17]
    - FMI Library [18]
    - NumPy [2]
    - SciPy [19]
    - lxml [20]
    - Assimulo [21]
    - Cython [22]
    - Python-headers (normally, otherwise see python-dev packages for your operating system)

Where possible, install the PyFMI dependencies through Conda [23] as at the time of writing (March 2021), it seems that the version of the libraries available from pip [24] are outdated. Outdated libraries installed via pip should be uninstalled to prevent conflicts with the more up-to-date libraries from Conda. Once PyFMI is installed, the versions available to is can be found by running the following commands in a interactive Python session or script:

```
>>> import pyfmi
>>> pyfmi.check_packages()
```

Output akin to the below should be expected:

```
Performing pyfmi package check
==============================

PyFMI version ................ 2.8.5
```

```
Platform ..................... {your OS}

Python version ............... 3.8.5



Dependencies:

Package                       Version
-------                       -------
assimulo...................... 3.2.3
Cython........................ 0.29.22
lxml.......................... 4.6.2
matplotlib.................... 3.3.4
numpy......................... 1.20.1
scipy......................... 1.6.1
```

7.2 Overview

Hybrid Petri Net-Physical models are achieved through a number of steps. The Petri Net (PN) is handled solely through Macchiato itself, but the physical part of the model is loaded and run as an FMU (functional mock-up unit) via PyFMI. The FMUs in this work were exported from Modelica but in principle any suitably configured FMU is usable. The Bond Graph [25] (BG) methodology was used to describe the physical processes, achieved within Modelica via the BondLib library [26, 27]. The coordination between the two parts of the model is managed by the file `FMUInterface.py` in the `FMUInterface` directory. This file should not be edited by the user; instead, follow the guide found in §7.3 *Structure* which details how to set up the PN-FMU interface.

7.3 Structure

7.3.1 Interface Specification

Unlike the process for constructing purely PN models with Macchiato, a small amount of coding is required to specific the desired relationship between the PN and a BG components. The file `FMUInterface.py` imports the essential modules and provides the `pnfmu` object class used to execute the hybrid model. This class must be extended to fully specify the relationship. This should be done in an entirely separate script using a feature of Python known as inheritance [28]. In the inheriting class definition, all properties of the system that user wishes to be able to alter must be added as attributes in `__init__`. The method `inputFunction` must be updated to return all of the inputs expected by the FMU in the order in which they were specified and in the appropriate units (i.e. SI [29]). Conversely, the rules for changes made to the PN in response to the physical state of the BG (or other FMU model) should be specified in `netUpdate`. This method must return a Boolean value `update`, which should be `True` if any changes have been made and `False` otherwise. The methods, `newPN`, `endFiles`, `setPN`, `processResults`, and `run` are

generic to any system and should not be altered. Remember: Do not alter the file `FMUInterface.py` itself.

Having created the bespoke interface required for the hybrid model, it is recommend that the user create a function to the script to manage executions of the model and handle any command line arguments. It is likely that one should wish to run many iterations of the system to collect statistics. Make sure that the method `processResults` is always called directly after each call to run, otherwise the results from that iteration will not recorded to file.

Take care to ensure that consistent units of time, preferably seconds, are used between the components of the model.

7.3.2 General Parameters

The basic `pnfmu` object class takes the following parameters:

- Required:

  `name` A label for the model

  `pn` Petri Net to use in the hybrid model as either PetriNet object from Macchiato or path to an `*.mpn` file

  `fmuPath` Path to FMU to use in the hybrid model

  `inputs` A list object containing the input variable names as strings in the order expected by the FMU source code

- Optional:

  `wd` Alternative working directory for the hybrid simulation (Default is `None`, resulting in no change)

  `tMax` The maximum simulation clock in seconds

  `tStep` The sampling period for checking the physical model for states requiring updates to the Petri Net. For example, the temperature of a component reaching the critical value at which a reactive process is activated.

Additional parameters can be added to the inherited class in accordance with the requirements of the simulation.

7.3.3 Model Construction

For instruction relating to the creation of the PN component of the hybrid model, please refer to §6 *Usage*. When constructing a PN, consider how it is going to interact with the physical model and make sure that its places present the necessary information simply. The FMU can constructed by any means convenient to the user, but must to be able to accept the input of external variables. In Modelica, this is achieved by declaring variables of type `input Real` at the beginning of the model and using them as the parameterisation of input blocks etc. – for instance, in the case of the work presented here, variable components from the BondLib library have been

used. Modelica environments such as OpenModelica (OMEdit/OMShell) [30] or Dymola [31] can be used for the development of suitable models and to export them to in FMU form.

7.4 Example

An example system is presented [32], in which a PN describing the failure modes of a nuclear reactor is coupled with a representation of heat flow through its fuel rods in BG form. A schematic of the system is shown in figure 13 and the corresponding PN is shown in figure **??**. For the sake of the model, each

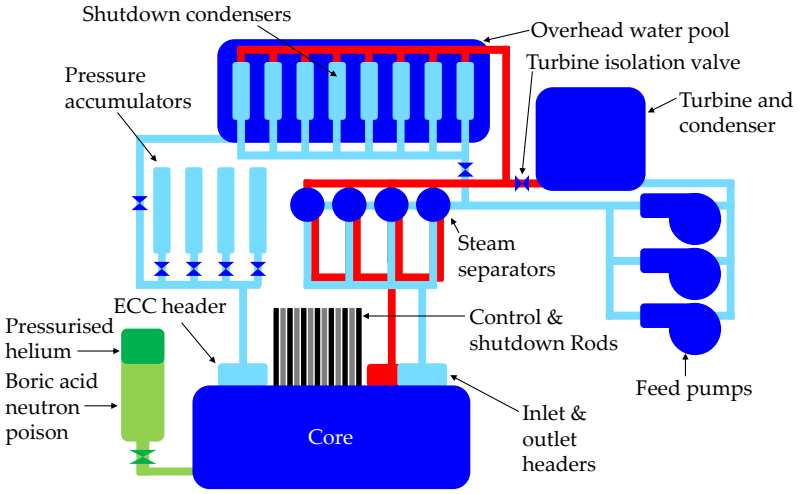


Figure 13: Schematic of reactor primary coolant, shutdown condensation, and emergency shutdown system.

fuel rod is treated as identical, so only one need be simulated. The rod is comprised of twelve clusters of uranium dioxide fuel pins in series, coated in zircaloy-2 cladding, and surrounded by light water coolant flowing down its length as a steam/liquid mix. Each cluster of pins is modelled in two parts, with heat transfer along its length, between each material, and in accordance with the mass flow of coolant. Figure 15 shows a schematic of the rod and figure 16 shows the BG model. The files for this model are available in the folder `PNFMU_Hybrid_Example` and its subdirectories. In `ATR_Model`, the files required to run the example are found, with `ATR_PetriNet.mpn` and `ATR_Bond_Graph_X.fmu` being the inputs for the PN and BG parts of the model respectively, where `{X}` is the operating system in use (a 64 bit version of Linux, macOS, or Windows 10 is required). These are read by `ATR_Hybrid.py` to achieve the hybridisation, discussed in detail below in §7.4.1 *`ATR_Hybrid.py`*.

In the folder `Petri_Net_Construction`, `ATR_PetrNet_Base.vsdm` is found which provides convenient graphical editing of the former via MS Visio (see §6.2 *Graphical Petri Net Construction with Microsoft Visio* for details of the usage of the macro). Note that as the reactor has forty shutdown rods, the function and failure of each of which is modelled individually, this part of the PN is not generated from `ATR_PetrNet_Base.vsdm`, but instead via the script `Rods.py`. The script will write an `*.mpn` file with the desired number of rods

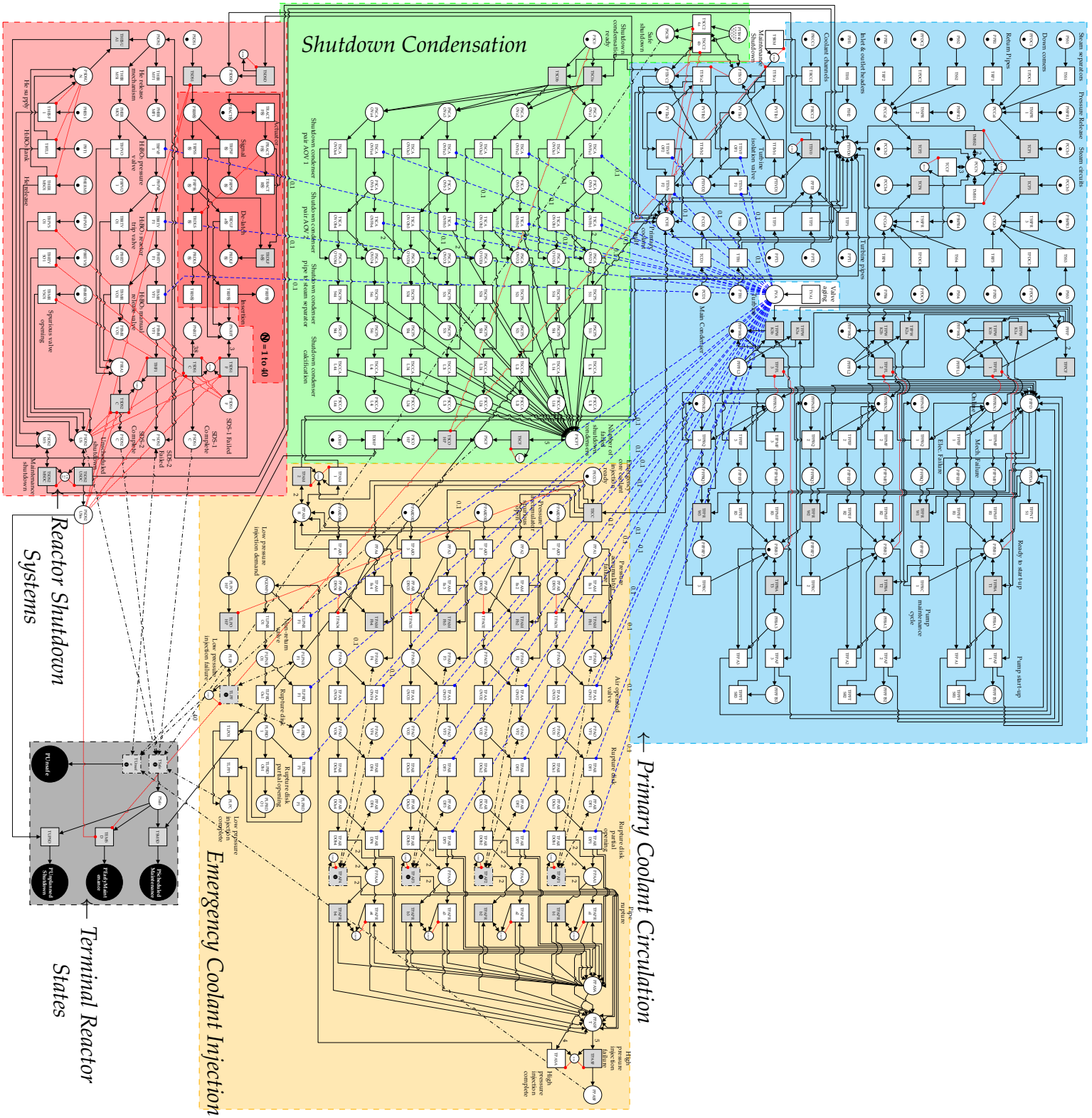


Figure 14: The Petri Net used to model the operational states and component failures of the reactor system in the FMU example. Its sections are coloured to mark their function: **blue** – primary coolant circulation, **green** – shutdown condensation, **yellow** – emergency coolant injection, **red** – reactor shutdown systems (shutdown rods and boric acid moderator injection), wherein the dark section is repeated 40 times in parallel, and **black** – terminal reactor states.

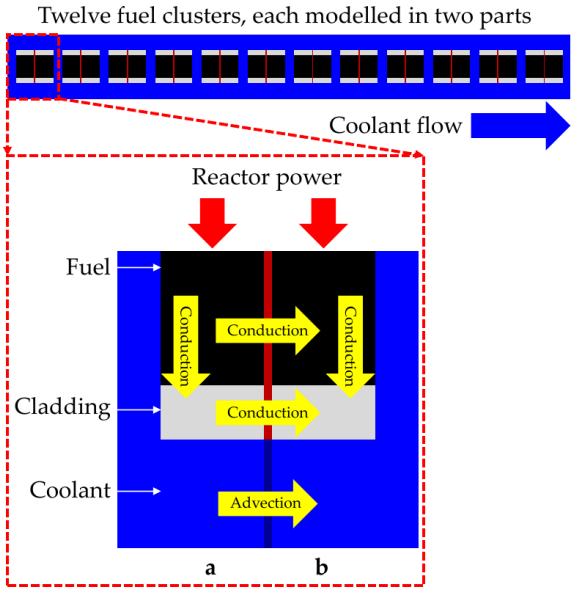


Figure 15: Schematic of heat transfer along a fuel rod.

which can be copied into the output of `ATR_PetrNet_Base.vsdm`.

The folder `Reactor_Bond_Graph` contains the Modelica package required to build the FMU, which can be viewed and edited from the aforementioned Modelica environments. A version of the model for FMU export is found as `ATR_BondGraph_FMU.mo`, as well as a stand-alone version as `ATR_Bond_Graph.mo`. When loading the Modelica codes, import BondLib first (available from ref.[26] or [33]), followed by `Reactor_Bond_Graph`. In both cases, this is done by loading the file named `package.mo` in their respectively top directories.

7.4.1 `ATR_Hybrid.py`

This file sets up the specific interactions for the example system and manages the execution of batches of simulation. Its structure is explained in the following sections It is assumed that the reader is already familiar with the concept of inheritance in Python and has read the in-file documentation within itself.

7.4.1.1 `atr`

The first step is to create an object class which inherits from the `pnfmu` class of `FMUInterface` (referenced locally as `fmui`).

```
class atr(fmui.pnfmu):
    def __init__(self, name, pn, fmuPath, inputs, wd=None,
                 tMax=3600.0, tStep=10.0):
        fmui.pnfmu.__init__(self, name, pn, fmuPath,
                            inputs, wd, tMax, tStep)
```

To this new class the attributes required to describe the features of the system

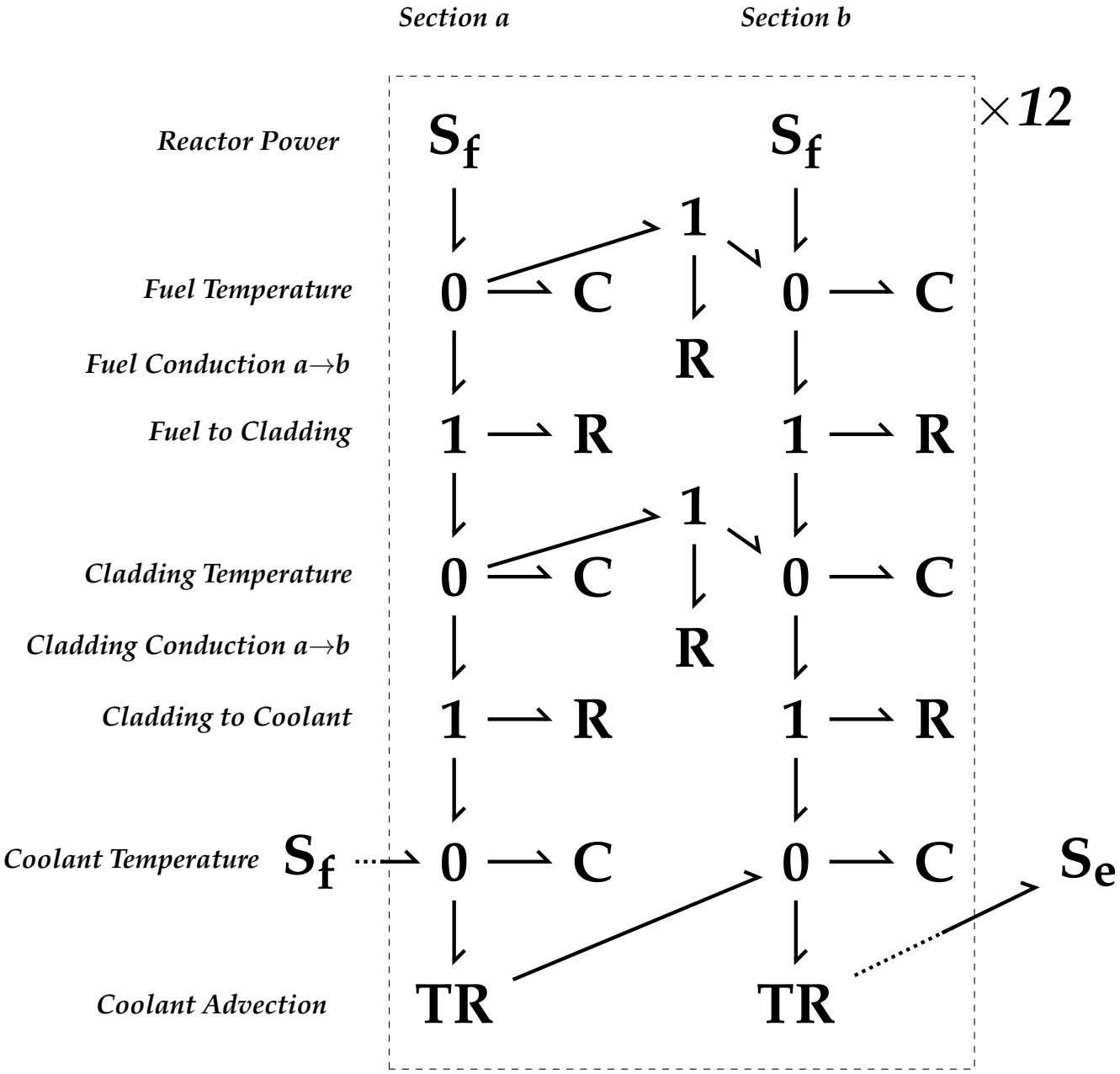


Figure 16: Bond Graph to model heat transfer in a nuclear reactor core from fuel pin clusters in a fuel rod to coolant fluid, separated by a layer of cladding. A fuel rod is comprised of twelve sets of 50 clusters, with each modelled in two sections, labelled ***a*** and ***b***, with longitudinal conduction. Note that the coolant heat sources and sink elements only connect to the first and last sections respectively, and intermediate sections' transformer elements attach to their successive neighbour. The coolant enters with an initial temperature and collects thermal energy from the fuel/cladding clusters, transferred advectively down the coolant channel by a transformer element in accordance with the mass flow rate, before leaving the system after the final cluster.

salient to the interface are added; in this case, the default parameters values to passed to the FMU and a number of Boolean tags to record some important system state toggles.

7.4.1.2 `inputFunction` & `netUpdate`

Next, an updated version of `inputFunction` must be supplied. This must read the state of the PN and the value of the simulation clock (supplied as a parameter, `t`, which must be given, even if irrelevant), and returns the values expected by the FMU. Likewise, the user must supply their version of `netUpdate` to read the last recorded state of the FMU and makes any necessary adjustment to the marking of the PN, returning a Boolean value to indicate if changes have been made. The state of the FMU is accessible from the attribute `self.results`, and to get the most recent value of variable one uses `self.results[-1]['NAME'][-1]`, where `'NAME'` is the label assigned to the variable in Modelica (or equivalent). In this example, the temperature of the fuel and cladding components are checked against their thresholds for damage and marks the places `PFuelOverheat` or `PCladOverheat` accordingly.

7.4.1.3 `run`

Separate from the `atr` object, the `run` function manages initial set of the hybrid model object and repeated executions of the simulation. If `ATR_Hybrid.py` is run from the command line directly, arguments passed to `run` from `main()`, but the file can also be imported as a module in a separate script, with the user passing the required parameters from there. The parameters need to be supplied as a `list` object in the following order:

1. Path to the `*.mpn` file containing the PN, or a `PetriNet` object
2. Path to the `*.fmu` file containing the physical model
3. Simulations to perform, which can take the form of a single integer, in which case, that many simulations will be performed, or two integers in ascending order separated by a colon, in which case, simulations with labels from the lower to upper bound will be executed. This allows a batch of simulations to be run in parts at the convenience of the user.
4. (Optional) Path in which a new folder is created to save simulation output. By default, the current working directory is used.

To facilitate the interaction between the PN and FMU, a number of transitions and places are added to the base PN (see §6.3 *Scripting Tools* for more details of the PN scripting tools in Macchiato).

```
    pn.trans['TUnsafe'].maxFire = None
    pn.addTrans('TUnsafeCounter', delay=25.0, maxFire=1)
```

```
    pn.trans['TUnsafeCounter'].addInArc('PUnsafe')
    pn.trans['TUnsafeCounter'].addOutArc('PUnsafe')
    pn.addPlace('PFuelOverheat')
    pn.addPlace('PCladOverheat')
    pn.addTrans('TFuelOverheat', maxFire=1)
    pn.addTrans('TCladOverheat', maxFire=1)
    pn.trans['TFuelOverheat'].addInArc('PFuelOverheat')
    pn.trans['TFuelOverheat'].addOutArc('PFuelOverheat')
    pn.trans['TCladOverheat'].addInArc('PCladOverheat')
    pn.trans['TCladOverheat'].addOutArc('PCladOverheat')
```

As the PN's parameters are given in hours, and those of the FMU are in seconds, the former must be converted.

```
    for t in pn.trans:
        if pn.trans[t].uniform is not None:
            pn.trans[t].uniform *= 3600.0
        if pn.trans[t].delay is not None:
            pn.trans[t].delay *= 3600.0
        if pn.trans[t].weibull is not None:
            pn.trans[t].weibull[0] *= 3600.0
            if len(pn.trans[t].weibull) == 3:
                pn.trans[t].weibull[2] *= 3600.0
        if pn.trans[t].cyclic is not None:
        pn.trans[t].cyclic[0] *= 3600.0
        pn.trans[t].cyclic[1] *= 3600.0
    pn.units = 'sec'
```

A `list` object, `inputs`, must be supplied which contains the input variable names anticipated by the FMU, matching the order in which they are returned by `inputFunction`.

```
    inputs = ['fuelThIn', 'coolTR', 'coolTin', 'fuelC',
              'fuelToCladR', 'cladC', 'cladToCoolR',
              'coolC', 'fuelToFuelR', 'cladToCladR',
              'spCap']
```

Next a `list` of the names of variables from the FMU that the user wishes to monitor must be given as a `list` called `interest`. Any variable that is required to be visible to `netUpdate` must be included. Only variables given in interest will be recorded in the simulation output files. In this example, the variables of interest are the thermal energy entering the system from the fuel (`fuelThIn`), the coolant mass flow rate (`coolTR`), the coolant temperature on entry to the rod `coolTin` and the temperatures of each of the 24 points along the rod at which the coolant, cladding, and fuel, are modelled (`Coolant1.e`, `Cladding1.e`, `Fuel1.e`, etc.).

```
        model = atr(pn.name, pn, fmu, inputs, wd=wd,
                    tMax=4.0*365.25*24*3600.0,
```

```
                    tStep=8766.0*3600.0)
        model.run()
        model.processResults(interest)
```

7.4.2 Execution

To run the the above example with a simulation batch size of 1000, one would enter the following:

```
$ python Example/ATR_Model/ATR_Hybrid.py Example/ATR_Model/
   ATR_PetriNet.mpn Example/ATR_Model/ATR_BondGraph_{OS}.
   fmu 1000
```

where `{OS}` is replaced by the appropriate system.

Acknowledgements

With thanks to Dr Robert *"Larus"* Lee for developing the original Macchiato stencil and macro for Microsoft Visio, and to Andrey Vasilyev for useful conversations regarding the Modelica language. Kind regards to John Andrews for supervisorial support throughout the development of Macchiato.

Title page image by Nathan Dumlao, used under the Unsplash Licence [34].

References

[1] Python. *www.python.org*.

[2] NumPy. *www.numpy.org*.

[3] Matplotlib. *www.matplotlib.org*.

[4] Graphviz. *www.graphviz.org*.

[5] Microsoft Visio. *www.microsoft.com/en/microsoft-365/visio/flowchart-software*.

[6] Scoop. *www.scoop.sh*.

[7] Homebrew. *www.brew.sh*.

[8] Carl Adam Petri. *Kommunikation mit Automaten* (In German). PhD thesis, Technical University Darmstadt, 1962.

[9] Winfrid G. Schneeweiss. *Petri Net Picture Book (An Elementary Introduction to the Best Pictorial Description of Temporal Changes)*. LiLoLe – Verlag GmbH (Publ. Co. Ltd.), 2004.

[10] Athanasios Papoulis and S. Unnikrishna Pillai. *Random Variables and Stochastic Processes*. McGraw Hill, 4th edition, 2002.

[11] Ionuţ Florescu and Ciprian A. Tudor. *Handbook of Probability*. John Wiley & Sons, 2013.

[12] G. M. Clarke and D. Cooke. *A Basic Cource in Statistics*. John Wiley & Sons Ltd., 2004, 5th edition, 1978.

[13] Brian Dennis and G. P. Patil. *Lognormal Distributions, Theory and Applications*. Marcel Dekker New York, 1987.

[14] Jack R. Benjamin and C. Allin Cornell. *Probability, Statistics, and Decision for Civil Engineers*. McGraw-Hill Book Company, 1970.

[15] John R. Taylor. *An Introduction to Error Analysis – The Study of Uncertainties in Physical Measurements*. University Science Books, 1997, 2nd edition, 1982.

[16] Functional Mock-up Interface. *www.fmi-standard.org*.

[17] modelon-community/PyFMI – GitHub. *www.github.com/modelon-community/PyFMI*.

[18] FMI Library. *www.jmodelica.org/FMILibrary*.

[19] SciPy. *www.scipy.org*.

[20] lxml. *www.lxml.de*.

[21] Assimulo. *www.jmodelica.org/assimulo*.

[22] Cython. *www.cython.org*.

[23] Conda. *www.docs.conda.io/en/latest*.

[24] pip. *www.pip.pypa.io*.

[25] Henry M. Paynter. *Analysis and Design of Engineering Systems*. The M.I.T. Press, 1961.

[26] BondLib. *www.build.openmodelica.org/Documentation/BondLib.html*.

[27] modelica-3rdparty/BondLib – GitHub. *www.github.com/modelica-3rdparty/BondLib*.

[28] Python Inheritance. *www.w3schools.com/python/python_inheritance.asp*.

[29] Si units -NPL. *www.npl.co.uk/si-units*.

[30] OpenModelica. *www.openmodelica.org*.

[31] Dymola. *www.3ds.com/products-services/catia/products/dymola*.

[32] Mark James Wootton, John D. Andrews, Ying Zhou, Roger Smith, A. John Arul, Gopika Vinod, M. Hari Prasad, and Vipul Garg. A Hybridised Petri Net-Pseudo Bond Graph Model for a Nuclear Reactor Coolant System. *Submission pending*, 2021.

[33] MJWootton-Resilience-Projects/BondLib – GitHub. *www.github.com/MJWootton-Resilience-Projects/BondLib*.

[34] Unsplash Licence. *www.unsplash.com/license*.