Andrey

A pedagogical implementation of Markov chains

Antoine Rauzy
Department of Mechanical and Industrial Engineering
S. P. Andersens veg 5, Valgrinda*3.306
Antoine.Rauzy@ntnu.no

Licenses & versions

Andrey is free software distributed by the AltaRica Association under GNU GPLv3 license.

Version	1.0.0
Date	23/02/2019

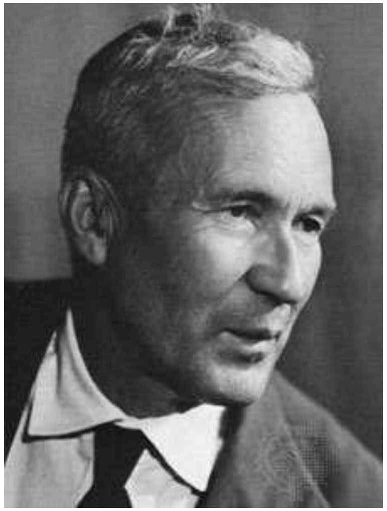
Andrey Andreyevich Markov was a Russian mathematician best known for his work on stochastic processes.
A primary subject of his research later became known as Markov chains and Markov processes

Andrey Andreyevich Markov
1856 - 1922

Source Wikipedia

Andrey Nikolaevich Kolmogorov was a 20th-century Soviet mathematician who made significant contributions to the mathematics of probability theory, topology, intuitionistic logic, turbulence, classical mechanics, algorithmic information theory and computational complexity.

In 1933, Kolmogorov published his book, Foundations of the Theory of Probability, laying the modern axiomatic foundations of probability theory and establishing his reputation as the world's leading expert in this field. In his study of stochastic processes, especially Markov processes, Kolmogorov and the British mathematician Sydney Chapman independently developed the pivotal set of equations in the field, which have been given the name of the Chapman–Kolmogorov equations.



Andrey Nikolaevich Kolmogorov
1903 - 1987

Later, Kolmogorov focused his research on turbulence, where his publications (beginning in 1941) significantly influenced the field. In classical mechanics, he is best known for the Kolmogorov–Arnold–Moser theorem, first presented in 1954 at the International Congress of Mathematicians. In 1957, working jointly with his student Vladimir Arnold, he solved a particular interpretation of Hilbert's thirteenth problem. Around this time he also began to develop, and was considered a founder of, algorithmic complexity theory – often referred to as Kolmogorov complexity theory.

Source Wikipedia

Table of contents

- Introduction
- Getting Started
- The S2ML+MRK modeling language
  - Basic components
  - Structuring constructs
- Commands
- References

Appendix

- Grammar of S2ML+MKRT models
- Grammar of Andrey commands
- Known bugs

INTRODUCTION

Rational

Markov chains are one of the fundamental stochastic models. They are used, implicitly, explicitly or as a reference in virtually all branches of science and engineering.

Andrey is a pedagogical implementation of discrete-time and continuous-time Markov chains with rewards:

- Models are written in the **S2ML+MRK domain specific modeling language**, which is the combination of S2ML (S2ML stands for system structure modeling language), a set of **object-oriented constructs** to structure models and Markov chains with rewards.
- It implements numerical solutions of Markov chains, in particular the **matrix exponentiation algorithm**.
- It comes as a **command interpreter**, making it possible to perform various studies.

This presentation specifies S2ML+MRK and presents the algorithms implemented by the tool as well as the commands to apply them.

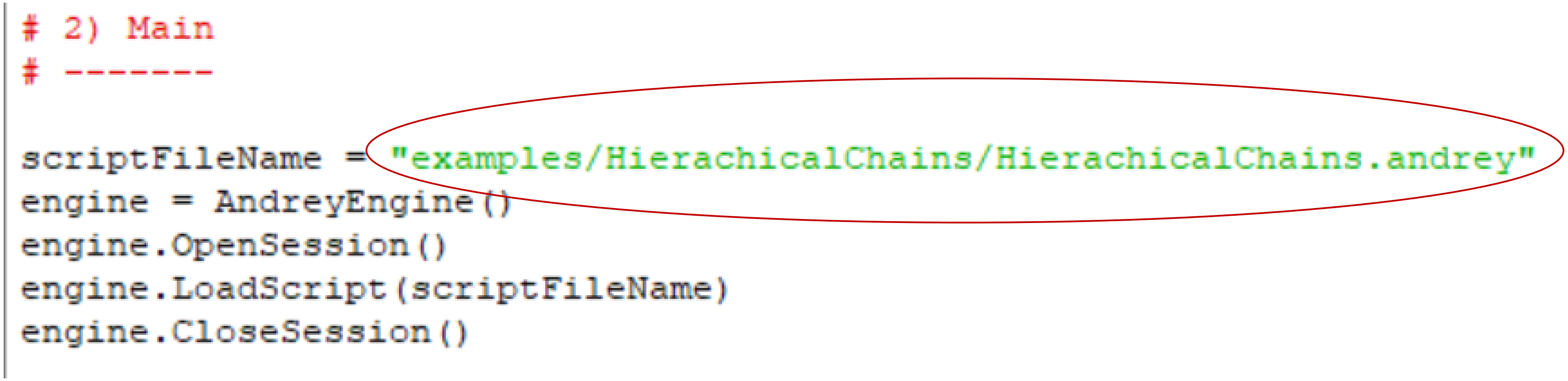
Andrey is developed in Python, for pedagogical purposes only. It is by orders of magnitude less efficient than available commercial tools.

The objective is to familiarize students with Markov chains as a **modeling language**.

Installing and Running Andrey

To install Andrey you just need to decompress the archive "Andrey1.0.0.zip" into local directory. Source files are the Python file "Andrey.py" as well as the directory "src" and its content.

To run Andrey you have to open the file Python file "Andrey.py" into your Python environment, set up the name of script file and run it.



```
# 2) Main
# -------

scriptFileName = "examples/HierachicalChains/HierachicalChains.andrey"
engine = AndreyEngine()
engine.OpenSession()
engine.LoadScript(scriptFileName)
engine.CloseSession()
```

Organization of this document

The remainder of this document is organized as follows.

- Section Getting started is a small introduction to Andrey.

The two next sections describe S2ML+MRK:

- Section Basic components presents the core of the language.
- Section Structuring constructs presents object-oriented constructs to structure models.

The next section describe the command interpreter:

- Section Commands describes Andrey commands.

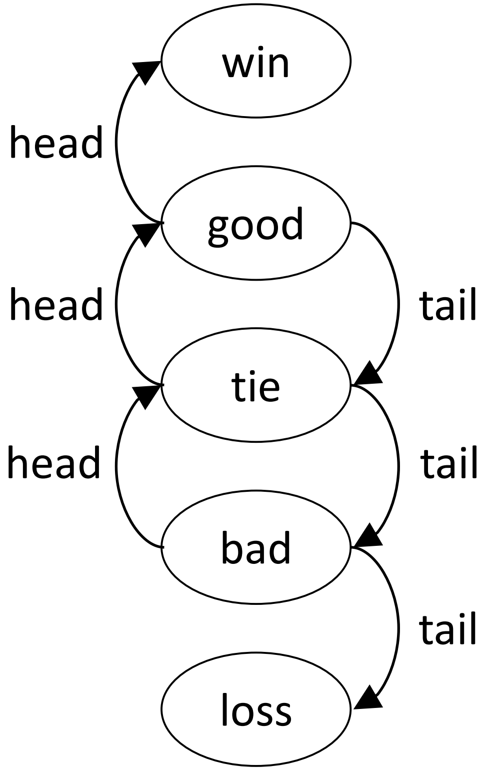
Finally, the appendix completes this document.

- Appendix S2ML+MRK gives the Backus-Naur form of the modeling language.
- Appendix Andrey gives the Backus-Naur form of Andrey commands.
- Appendix Known bugs reports know problems with the current version of Andrey.

GETTING STARTED

S2ML+MRK: discrete-time Markov chains

S2ML+MRK is a textual format to describe (hierarchical) Markov chains.
Here follows a example of discrete-time Markov chain.



```
block Main
    state win, good;
    state tie (probability = 1.0);
    state bad, loss;
    event head (probability = p);
    event tail (probability = (sub 1.0 p));
    parameter p = 0.51;
    transition
        head: good -> win;
        tail: good -> tie;
        head: tie -> good;
        tail: tie -> bad;
        head: bad -> tie;
        tail: bad -> loss;
end
```

S2ML+MRK: discrete-time Markov chains (bis)

```
block Main
    state win;
    …
    event head (probability = p);
    …
    transition
        head: good -> win;
        …
end
```

- Each model is described in a **block** which contains declarations of objects of the model. A block starts with keyword **block** followed by the name of the model (here `Main`) and ends with the keyword **end**.
- Five types of basic objects are used to define **Markov chains**: **states**, **events**, **parameters**, **transitions** and **rewards**. States and events must be declared before they are referred to in transitions.
- States and events have a name and possibly some attributes, which are used to define initial probabilities of states and probabilities and rates of transitions.

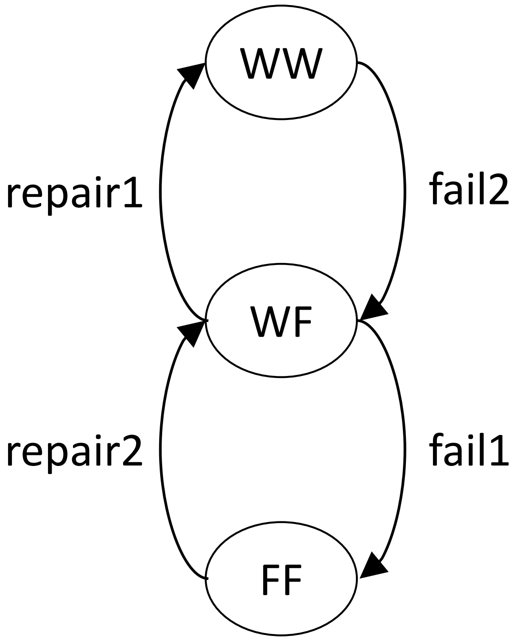
S2ML+MRK: discrete-time Markov chains (ter)

```
block Main
    state win;
    …
    event head (probability = p);
    …
    transition
        head: good -> win;
        …
end
```

- Attributes are given between after the name (of the node or the event). They are pairs (name, value), where value is an arithmetic expression, possibly involving parameters.
- Declarations of nodes, events, parameters, transitions and rewards are terminated with a ";".
- Although this is not mandatory, models are usually stored into text files with the extension ".mrk".

S2ML+MRK: continuous-time Markov chains

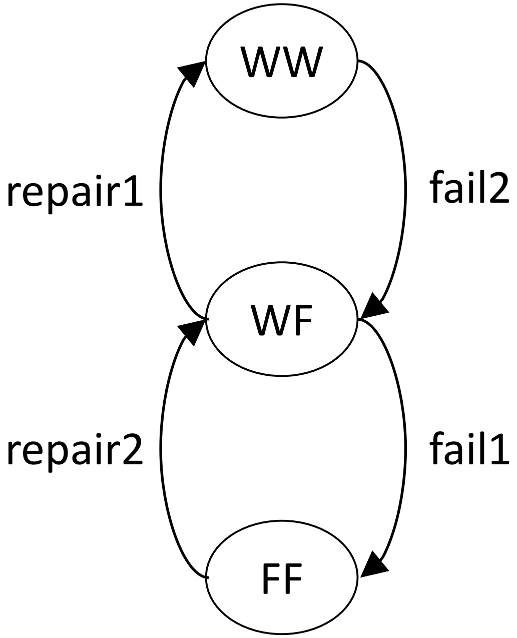
Continuous-time Markov chains are built in the same way. The only difference is that events are given rates and not probabilities. E.g.



```
block Main
    state WW (probability = 1.0);
    state WF, FF;
    event fail1 (rate = lambda);
    event fail2 (rate = (mul lambda 2.0));
    event repair1 (rate = mu);
    event repair2 (rate = (mul mu 2.0));
    parameter lambda = 1.23e-4;
    parameter mu = (div 1 12);
    transition
        fail2: WW -> WF;
        fail1: WF -> FF;
        repair1: WF -> WW;
        repair2: FF -> WF;
end
```

Rewards

In both discrete-time and continuous-time Markov chains it is possible to associate rewards, i.e. real valued quantities to states.



```
block Main
    state WW (probability = 1.0);
    state WF, FF;
    …
    reward Production =
        (add (in WW 100) (in WF 70));
end
```

Point-reward at time t:

$$100 \times p_{WW}(t) + 70 \times p_{WF}(t)$$

Mean-reward at time t:

$$100 \times \frac{sj_{WW}(t)}{t} + 70 \times \frac{sj_{WF}(t)}{t}$$

where $sj_s(t)$ is the sojourn-time in state *s* from time *0* to time *t*.

Assessment process

The assessment process of a model is typically made of the following steps:

1. The model is loaded from a text file.

2. The model is checked and rewritten in a form in which the calculation process can start. This steps is called instantiation in the S2ML jargon.

3. Calculations are performed. Results are printed out into text files.

Scripts

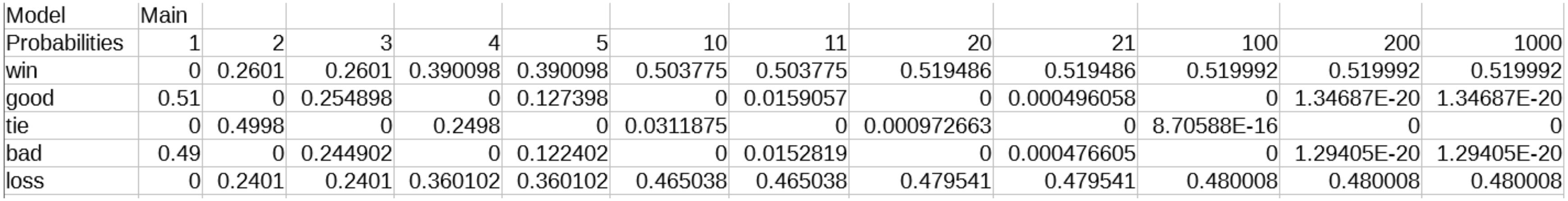
Andrey is a **command interpreter**: it reads commands into a text file and execute them. There are commands to perform each of the steps described in the previous slide.

```
# Step 1: the model is loaded
load "example/HeadAndTails/HeadAndTailsSmall.mrk"

# Step 2: the model is instantiated
instantiate model

# Step 3: the Markov chain is assessed
assess DTMC Main [1, 2, 3, 4, 5, 10, 11, 20, 21, 100, 200, 1000]
probabilities=true "results.csv" mode=write
```

Scripts are text files. Although this is not mandatory, models are usually stored into text files with the extension ".andrey".

Results

result.csv



Model	Main												
Probabilities	1	2	3	4	5	10	11	20	21	100	200	1000
win	0	0.2601	0.2601	0.390098	0.390098	0.503775	0.503775	0.519486	0.519486	0.519992	0.519992	0.519992
good	0.51	0	0.254898	0	0.127398	0	0.0159057	0	0.000496058	0	1.34687E-20	1.34687E-20
tie	0	0.4998	0	0.2498	0	0.0311875	0	0.000972663	0	8.70588E-16	0	0
bad	0.49	0	0.244902	0	0.122402	0	0.0152819	0	0.000476605	0	1.29405E-20	1.29405E-20
loss	0	0.2401	0.2401	0.360102	0.360102	0.465038	0.465038	0.479541	0.479541	0.480008	0.480008	0.480008

In result files, which are text files, items are separated with tabs so that results can be easily loaded into spreadsheets (Excel or equivalent).

S2ML+MRK:

BASIC COMPONENTS

Basic components

Basic components of S2ML+MRK models are:

- **Blocks** that contain declarations of other objects of a model.
- Declarations of **states**, **events**, **parameters**, **rewards** and **transitions**.
- All these declarations (but those of transitions) involve arithmetic expressions.

In the sequel, models are described using `this font`. Keywords are underlined using **`this font`**.

S2ML+MRK models must be written using ASCII characters.

Identifiers, i.e. names of elements obeys the following syntax:

- They start with a letter from a to z or from A to Z.
- They are made of any number of letters, digits, underscores "_".

E.g. `Plant`, `failed`, `R3151`, `this_is_a_valid_although_a_very_long_name`.

Comments

It is possible to add **comments** everywhere in a S2ML+MRK model.

- Any sequence of text between /* and */ is a comment, even if it spreads over several line.
- All characters after // until the end of the line is a comment.

In the sequel, we shall color comments *in italic and green*.

```
/*
 * This is a comment before a block declaration
 */
block Plant // This is a comment till the end of the line
    // declarations
end
```

Blocks

Blocks are the basic container of S2ML+MRK. They are **prototypes** in the sense of object-orientation theory. Blocks contain declarations of parameters, states, ports and sources (and other elements that will be described later).

A block declaration starts with the keyword "block", followed by the name of the block. It finishes with keyword "end". E.g.

```
block Task
   state work, test;
   event done(probability = doneProbability);
   event redo(probability = redoProbability);
   parameter doneProbability = 0.1;
   parameter badProbability = 0.2;
   transition
      done: work -> test;
      redo: test -> work;
end
```

Within a block, all elements must have a different name, even though they are of different types, e.g. a node and a parameter. Elements can be declared in any order.

States & events

States, and **events** can be declared either individually or several at a time. They can be associated attributes (multiple attributes are separated with commas. E.g.

```
state rainy (probability = 1.0);
state nice, snowy;
event r2n, r2s(probability = 0.25);
```

Both events `r2n` and `r2s` have the probability 0.25.

The attribute `probability` of states defines their initial probability.

The attribute `probability` of events defines their probability in case of discrete-time Markov chain or of immediate events in continuous-time Markov chains.

The attribute `rate` of events defines their rate in case of timed events in continuous-time Markov chains.

Parameters & rewards

Parameters are used in arithmetic expressions. They are declared together with the expression that defines or redefines them. E.g.

```
 parameter failureRate = 1.2e-5;
parameter B.repairRate = (mul 2 A.repairRate);
```

Rewards are used to associate random variables (real valued quantities) to states of a Markov chain. They are defined like parameters with arithmetic expressions. The value of a reward in a given state is defined by means of the built-in "in". E.g.

```
reward Production = (add (in WW 100) (in WF 70));
```

The reward `Production` takes the value 100 in the state `WW` and the value 70 in the state `WF`. Implicitly, it takes the value 0 in all other states.

Transitions

Transitions are labeled with an **event** and connect two states, the **source state** and the **target state**. Both the event and the two states must be declared before the transition is declared.

The declaration of one or several transitions must be preceded with the keyword "`transition`".

```
transition
   done: work -> test;
   redo: test -> work;
```

Arithmetic expressions

The current version of Andrey accepts the following arithmetic expressions.

Syntax	#arguments	Semantics
`Identifier`	0	Reference to a parameter
`Floating point number`	0	The number
`(add e1 … en)`	≥ 1	Sum of the arguments
`(sub e1 … en)`	≥ 1	First argument minus the others
`(mul e1 … en)`	≥ 1	Product of the arguments
`(div e1 … en)`	≥ 1	First argument divided by the others
`(neg f)`	1	Opposite
`(min e1 … en)`	≥ 1	Minimum of its arguments
`(max e1 … en)`	≥ 1	Maximum of its arguments

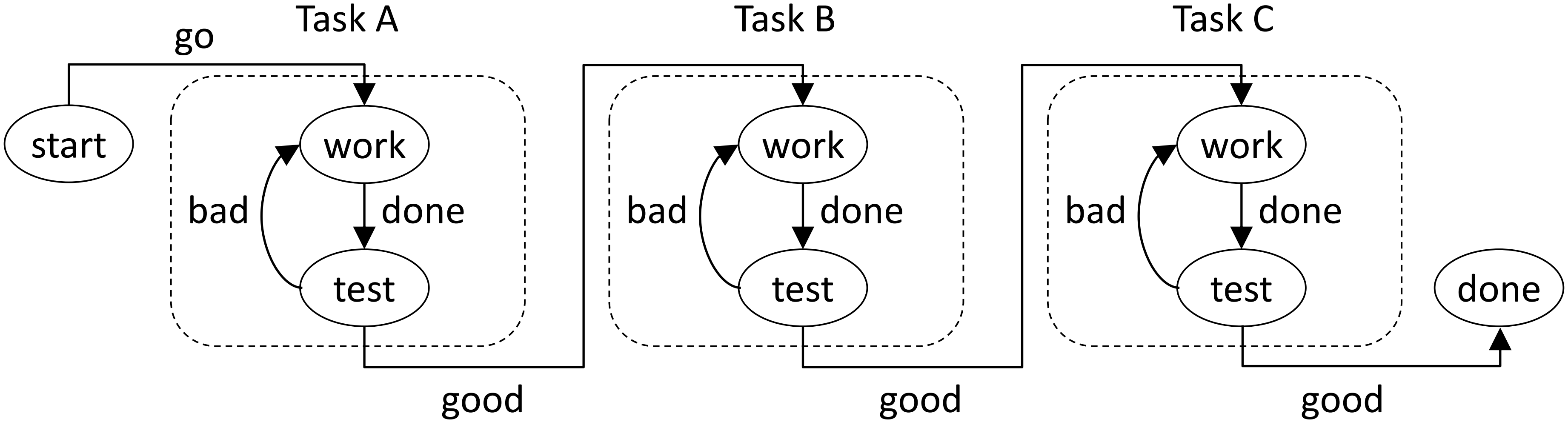
Examples:

```
    (mul 0.8 weight)          (max (sub f g) (sub f g) 1.0)
    (neg e)
```

S2ML+MRK:
STRUCTURING CONSTRUCTS

Hierarchical models

Consider a process made of 3 tasks in a row, each task requiring some iterations to be achieved. This process could be represented by the following Markov chain.



Strictly speaking the dashed rounded rectangle surrounding the states of each task are useless: they do not play any role in the mathematical definition of the chain. However, such "macro-states" prove to be very useful to design and to maintain models. S2ML provides several constructs to design such hierarchical models.

Blocks in blocks

Each task can be represented by a Markov chain. E.g.

```
block TaskA
   state work, test;
   event done(probability = doneProbability);
   event bad(probability = badProbability);
   parameter doneProbability = 0.1;
   parameter badProbability = 0.2;
   transition
      done: work -> test;
      bad: test -> work;
end
```

Blocks in blocks (bis)

The process can then be represented by the following hierarchical model:

```
block Process
   state start(probability = 1.0);
   block TaskA … end
   block TaskB … end
   block TaskC … end
   state done;
   event go(probability = 1.0);
   event good(probability = (sub 1.0 A.badProbability));
   transition
      go: start -> A.work;
      good: A.test -> B.work;
      good: B.test -> C.work;
      good: C.test -> done;
end
```

The block `Process` **composes** the blocks `TaskA`, `TaskB` and `TaskC`.

Instantiated form

The previous hierarchical model is equivalent to the following instantiated model:

```
block Process
   state start(probability = 1.0);
   state A.work, A.test;
   event A.done(probability = A.doneProbability);
   event A.bad(probability = A.badProbability);
   parameter A.doneProbability = 0.1;
   parameter A.badProbability = 0.2;
   transition
      A.done: A.work -> A.test;
      A.bad: A.test -> A.work;
   …
end
```

Cloning

Duplicating "by hand" blocks representing similar components would be both tedious and error prone in large systems studies. **Cloning** is the a first solution to this problem.

```
block Process
   block TaskA
      …
   end
   clones TaskA as TaskB;
   clones TaskA as TaskC;
   state done;
   event go(probability = 1.0);
   event good(probability = (sub 1.0 A.badProbability));
   transition
      go: start -> A.work;
      good: A.test -> B.work;
      good: B.test -> C.work;
      good: C.test -> done;
end
```

This model is equivalent to the first one: their instantiated form are the same.

Models as scripts

It is possible to change elements of clones in two ways.
Either directly in the clone directive:

```
clones TaskA as TaskB
   parameter badProbability = 0.3;
   end
```

Or later in the model:

```
clones TaskA as TaskB;
parameter TaskB.badProbability = 0.3;
```

This results of the "**model as script**" concept.

Paths

Thanks to absolute and relative paths, it is possible to refer to any element from any block of hierarchical model.

```
block Process
  block TaskA
    block SubTask1
      state work, test;
    end
  end
  block TaskB
    block SubTask1
      state work, test;
      transition
        FromA: main.TaskA.SubTask1.test -> work;
        ToA: test -> owner.owner.TaskA.SubTask1.work;
    end
  end
end
```

Absolute path: the keyword `main` denotes the top level block of the hierarchy.

Relative path: the keyword `owner` denotes the parent block in the hierarchy.

Classes & instances

Another solution consists in creating a on-the-shelf reusable component, via the notion of class. Then to instantiate this component as many time as needed. This makes it possible to create libraries of reusable modeling components.

```
class Task
   state work, test;
   event done(probability = doneProbability);
   event bad(probability = badProbability);
   parameter doneProbability = 0.1;
   parameter badProbability = 0.2;
   transition
      done: work -> test;
      bad: test -> work;
end

block Process
   Task A;
   Task B;
   Task C;
   …
end
```

Again, this model is equivalent to the first one: their instantiated form are the same.

Models as scripts (bis)

It is possible to change elements of instances in two ways.
Either directly in the instance declaration:

```
Task B
   parameter badProbability = 0.3;
   end
```

Or later in the model:

```
Task B;
parameter B.badProbability = 0.3;
```

Inheritance

Assume that we have to consider two kinds of tasks: simple tasks, which just perform some work, and full tasks that perform tests. It is possible to create first a class describing simple tasks, then to extends this class for full tasks. This mechanism is called **inheritance**.

```
class SimpleTask
   state work, test;
   event done(probability = doneProbability);
   parameter doneProbability = 0.1;
   transition
      done: work -> test;
end

class FullTask
   extends SimpleTask;
   event bad(probability = badProbability);
   parameter badProbability = 0.2;
   transition
      bad: test -> work;
end
```

Aggregation

Aggregation denotes a "uses" kind of a relation between blocks (or instances of classes).

```
block Process
    block DataBase
        parameter badProbability = 0.3;
    end
    block TaskA
        block SubTask1
            embeds main.DataBase as DB;
            state work, test;
            event bad(probability = DB.badProbability);
            transition
                bad: test -> work;
        end
    end
end
```

Aggregation: within the block `SubTask1`, `DB` becomes an alias for `main.DataBase`.

ANDREY COMMANDS

Role of commands

Categories of commands:

Andrey commands can be split into the following categories.

1. Commands to load, to check and to instantiate models.
2. Commands to assess models.
3. Commands to print out information, e.g. models.

Normally, commands of type 1, 2 and 3 are applied in sequence.

Order of arguments:

The order of arguments in a command matters, even though some arguments are optional. You have to follow strictly the syntax described in this section.

Optional arguments:

Optional arguments are given in a special form: `name=value`, where `name` is the name of the argument and `value` its value.

General considerations

One line commands:

Andrey commands spread normally over one line. It is however possible to write a command on several line by escaping the end of line is escaped with an anti-slash "\", e.g.

```
compute DTMC Main [1, 2, 3, 4, 5, 10, 100, 1000] \
    "statistics.txt" mode=append
```

Comments:

Comments can be introduced in Andrey scripts. Any character between the character # and the end of the line is a comment. We shall underline comment in green. E.g.

```
# this is a comment
```

File names and modes

File names:

Most of the commands require an input or an output file name as argument. File names can be given directly, e.g. `examples/LandOfOz.mrk` or surrounded with quotes, e.g. `"examples/LandOfOz.mrk"`.

The second form is mandatory when the file name or path includes spaces. It is wise to use it anyway.

Output file modes:

When opening a file to print out something, Andrey can do it in two modes: either the file is overwritten if it exists already -- this is the mode `write` --, or the new information is appended at the end of the existing file -- this is the mode `append`. If the file did not exists, it is created in both cases. By default, the mode is `write`. E.g.

```
compute DTMC Main 1000 "results.txt" mode=append
```

The above command calculate the shortest path from node `A` to node `B` and appends the result to the file `results.txt`.

Result files

As much as possible, result files are organized in a way they can be loaded into spreadsheets (Excel or equivalent). Items are separated with tabs. Methods to load such text files differ from one spreadsheet tool to another.

result.txt

```
Model   Main
Probabilities   1   2   3   4   5   100
A.ping  0.5 0.5 0.4375  0.390625    0.347656    0.196721
A.pong  0.5 0.375   0.34375 0.304688    0.271484    0.131148
B.ping  0.0 0.125   0.15625 0.195313    0.220703    0.196721
B.pong  0.0 0.0 0.0625  0.09375 0.121094    0.131148
C.ping  0.0 0.0 0.0 0.015625    0.0328125   0.213115
C.pong  0.0 0.0 0.0 0.0 0.00625 0.131148
```

Floating point numbers

Andrey implements numerical solutions of Markov chains, i.e. everything is done by multiplying vectors (which contains probabilities of states) by the Markov chain (which can be seen as a sparse matrix).

This approach makes it possible to assess very large models. But it has a drawback as well: probabilities are encoded by means of floating point numbers. Floating point numbers encode reals with a given precision. To implement Andrey's calculations, the Python package mpmath is used. This package makes it possible to encode floating point numbers with any precision, but this precision must be chosen beforehand.

The current implementation makes the choice of having a precision of about 10^{-30}, which corresponds roughly speaking to an encoding of floating point numbers on 128 bits. Moreover, we had to decide when to consider that two probabilities are almost equal. We chose 10^{-20} as a threshold. Both thresholds are arbitrary and can be easily changed in the code (they are defined in the module S2MLCore/S2MLReals.py).

This to say that the user should not be surprised to see numbers around 10^{-20} appear in the result files where he or she would expect 0.0.

Command to assess discrete-time Markov chains

```
assess DTMC blockName numbersOfIterations
    [probabilities=True] [point-reward=True]
    fileName [mode=(append|write)]
```

```
numbersOfIterations ::= Integer | "[" Integer ("," Integer)* "]"
```

This command computes the probabilities of states of the given discrete-time Markov chain after the given number(s) of iterations. It prints out (if demanded) these probabilities and point rewards into the given file.

Command to assess continuous-time Markov chains

```
assess CTMC blockName missionTimes
    [probabilities=True] [sojourn-times=True]
    [point-rewards=True] [mean-rewards=True]
    fileName [mode=(append|write)]
```

```
missionTimes ::= Float | "[" Float ("," Float)* "]"
```

This command computes the probabilities of states of the given continuous-time Markov chain at the given mission(s) times. It implements the matrix exponentiation algorithm in its pure numerical form, i.e. by only multiplying vectors (encoding probabilities of states) by the probability matrix associated with Markov chain (this matrix is encoded as a sparse matrix). It prints out (if demanded) these probabilities, sojourn-times, point rewards and mean rewards into the given file.

Commands to print out information

```
print model fileName [mode=(append|write)]
```

This command prints out the original model.

```
print instantiated-model fileName [mode=(append|write)]
```

This command prints out the instantiated model.

REFERENCES

References

Recommend books on Markov chains:

William J. Stewart. Introduction to the Numerical Solution of Markov Chains. Princeton University Press. Princeton, New Jersey, USA. ISBN 978-0691036991. 1994.

APPENDIX

GRAMMAR OF S2ML+MRK

Models

```
Model ::= (ClassDeclaration | BlockDeclaration)*

ClassDeclaration ::=
    class Identifier BlockField* end

BlockDeclaration ::=
    block Identifier BlockField* end

BlockField ::=
        StateDeclaration | EventDeclaration | ParameterDeclaration
    |   RewardDeclaration | TransitionDeclaration
    |   BlockDeclaration | InstanceDeclaration
    |   ExtendsDirective | EmbedsDirective | ClonesDirective
```

States, events & transitions

```
StateDeclaration ::=
    state Identifier ("," Identifier)* Attributes? ";"

EventDeclaration ::=
    event Identifier ("," Identifier)* Attributes? ";"

TransitionDeclaration ::=
    transition Transition*
Transition ::=
    Path ":" Path "->" Path ";"

Attributes ::=
    "(" Attribute ( "," Attribute )+ ")"
Attribute ::=
    Identifier "=" ArithmeticExpression
```

Parameters, rewards & arithmetic expressions

```
ParameterDeclaration ::=
    parameter Path "=" ArithmeticExpression ";"

ParameterDeclaration ::=
    reward Path "=" ArithmeticExpression ";"

ArithmeticExpression ::=
        Path     # reference to parameter
    |   Float
    |   "(" add ArithmeticExpression+ ")"
    |   "(" sub ArithmeticExpression+ ")"
    |   "(" mul ArithmeticExpression+ ")"
    |   "(" div ArithmeticExpression+ ")"
    |   "(" neg ArithmeticExpression ")"
    |   "(" min ArithmeticExpression+ ")"
    |   "(" max ArithmeticExpression+ ")"
```

Directives

```
InstanceDeclaration ::=
        Identifier Identifier ";"                # ClassName InstanceName
    |   Identifier Identifier BlockField* end    # idem

ClonesDirective ::=
        clones Path as Identifier ";"            # BlockPath CloneName
    |   clones Path as Identifier BlockField* end # idem

ExtendsDirective ::=
        extends Identifier ";"                   # ClassName

EmbdesDirective ::=
        embeds Path as Identifier ";"            # BlockPath LocalName
    |   embeds Path as Identifier BlockField* end # idem
```

Identifiers, paths, constants & comments

```
Identifier ::= [a-zA-Z_][a-zA-Z0-9_-]+

Path ::= Identifier ( "." Identifier )*

Integer ::= [0-9]+

Float ::= [+-]? [0-9]+ ("." [0-9]+)? ([eE] [+-]? [0-9]+)?
```

Comments can be added everywhere in the code.

- Single line comments introduced by `//`, which comment out the rest of the line.
- Multiline comments which comment out the text between `/*` and `*/`.

GRAMMAR OF ANDREY COMMAND

Scripts, commands load and instantiate

```
Script ::= Command*

Command ::=
        CommandLoad
    |   CommandInstantiate
    |   CommandAssess
    |   CommandPrint

CommandLoad ::=
        load model fileName
    |   load script fileName

CommandInstantiate ::=
    instantiate model
```

Commands assess

```
CommandAssess ::=
        CommandComputeDTMC
    |   CommandComputeCTMC

CommandComputeDTMC ::=
    compute DTMC blockName NumbersOfIterations
    (probabilities = True)? (point-rewards = True)?
    fileName [mode=(append|write)]

NumbersOfIterations ::=
        Integer
    |   "[" Integer ("," Integer)* "]"
```

Commands assess (bis)

```
CommandComputeCTMC ::=
     compute CTMC blockName MissionTimes
     (probabilities = True)? (sojourn-times = True)?
     (point-rewards = True)? (mean-rewards = True)?
     fileName [mode=(append|write)]

MissionTimes ::=
          Float
     |    "[" Float ("," Float)* "]"
```

Command print

```
CommandPrint ::=
        CommandPrintModel
    |   CommandPrintInstantiatedModel

CommandPrintModel ::=
    print model fileName [mode=(append|write)]

CommandPrintInstantiatedModel ::=
    print instantiated-model fileName [mode=(append|write)]
```

Comments

All characters comprised between a # symbol and the end of the line are considered as part of a comment.

KNOWN BUGS

Known bugs