



Brandenburgische
Technische Universität
Cottbus - Senftenberg

Faculty 1
Mathematics, Computer Science,
Physics, Electrical Engineering and
Information Technology

Institute of Computer Science

COMPUTER SCIENCE REPORTS

Report 02/16

December 2016

MARCIE Manual

Martin Schwarick
Christian Rohr
Monika Heiner

Computer Science Reports
Brandenburg University of Technology Cottbus - Senftenberg
ISSN: 1437-7969

Send requests to: BTU Cottbus - Senftenberg
Institut für Informatik
Postfach 10 13 44
D-03013 Cottbus

Martin Schwarick
Christian Rohr
Monika Heiner
marcie@informatik.tu-cottbus.de
<http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Marcie>

MARCIE Manual

Computer Science Reports
02/16
December 2016

Brandenburg University of Technology Cottbus - Senftenberg
Faculty of Mathematics, Natural Sciences and Computer Science
Institute of Computer Science

Computer Science Reports
Brandenburg University of Technology Cottbus - Senftenberg
Institute of Computer Science

Head of Institute:
Prof. Dr. Petra Hofstedt
BTU Cottbus - Senftenberg
Institut für Informatik
Postfach 10 13 44
D-03013 Cottbus

hofstedt@b-tu.de

Research Groups:
Computer Engineering
Computer Network and Communication Systems
Data Structures and Software Dependability
Database and Information Systems
Programming Languages and Compiler Construction
Software and Systems Engineering
Theoretical Computer Science
Graphics Systems
Systems
Distributed Systems and Operating Systems
Internet-Technology

Headed by:
Prof. Dr. H. Th. Vierhaus
Prof. Dr. H. König
Prof. Dr. M. Heiner
Prof. Dr. I. Schmitt
Prof. Dr. P. Hofstedt
Prof. Dr. C. Lewerentz
Prof. Dr. K. Meer
Prof. Dr. D. Cunningham
Prof. Dr. R. Kraemer
Prof. Dr. J. Nolte
Prof. Dr. G. Wagner

CR Subject Classification (1998): D.2.4, D.2.7, I.6.5, I.6.8, J.3

Printing and Binding: BTU Cottbus - Senftenberg

ISSN: 1437-7969

MARCIE Manual

<http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Marcie>

An analysis tool for extended stochastic Petri nets

December 7, 2016

Martin Schwarick, Christian Rohr, Monika Heiner
marcie@informatik.tu-cottbus.de

<http://www-dssz.informatik.tu-cottbus.de>
Data Structures and Software Dependability
Computer Science Department
Brandenburg University of Technology Cottbus-Senftenberg

Contents

1	Introduction	4
2	Background	6
2.1	Net classes	6
2.2	Model checking	7
2.3	The Abstract Net Description Language - ANDL	8
3	Engines	19
3.1	Qualitative Symbolic Engine	19
3.1.1	State space generation	19
3.1.2	Standard Properties	22
3.2	Exact Numerical Engine	23
3.3	Simulative Engine	30
3.4	Approximative explicit Engine	33
4	Model checking	34
4.1	CTL	34
4.1.1	Syntax	34
4.1.2	Patterns	36
4.2	CS(R)L	36
4.2.1	Syntax	36
4.2.2	Patterns	38
4.3	PLTLc	40
4.3.1	Syntax	40
4.3.2	Patterns	42
5	Templates	42
5.1	CTL	42
5.2	CSRL	43
5.3	PLTLc	45
6	Example Session	46
6.1	Model description	47
6.1.1	ANDL description	47
6.2	Analysis	48
6.2.1	State space	48
6.2.2	CTL	49
6.2.3	Printing the CTMC	51
6.2.4	Transient analysis	53
6.2.5	Steady state analysis	56
6.2.6	Reward analysis	57
6.2.7	PLTLc	58
	References	64

Index

65

1 Introduction

This manual gives an overview on MARCIE – *Model Checking And Reachability analysis done effiCIently*. MARCIE was originally developed as a symbolic model checker for stochastic Petri nets, building on its predecessor – IDDMC – Interval Decision Diagram based Model Checking – which has been previously developed for the qualitative analysis of bounded Place/Transition nets extended by special arcs. Over the last years the tool has been enriched to allow also quantitative analysis of extended stochastic Petri nets. We concentrate here on the user viewpoint. For a detailed introduction to the relevant formalisms, formal definitions and algorithms we refer to related literature.

A bit of history. The tool is the result of research activities at the chair for Data Structures and Software Dependability at the Brandenburg University of Technology in Cottbus supervised by Prof. Dr.-Ing. Monika Heiner.

A first prototype (EEMC) was based on a Zero Suppressed Binary Decision Diagram (ZBDD) engine developed by Andreas Noack [Noa99] and Jochen Spranger [Spr01]. It offered symbolic CTL and LTL model checking of safe Petri nets. The current version basically builds upon the concepts of this prototype implementation, such as shared decision diagrams, garbage collection, and heuristics for static variable ordering.

The Interval Decision Diagram (IDD) engine including the saturation algorithm for efficient state space generation, the CTL model checker and the user interface were implemented by Alexey Tovchigrechko [Tov08, HST09].

The IDD-based numerical stochastic analysis engine and the CSRL model checker were added by Martin Schwarick [Sch14].

Recently Christian Rohr integrated an approximative numerical engine and a stochastic simulation engine [HRSS10], which have been derived from our modelling and simulation tool Snoopy[HHL⁺12b].

Installation. MARCIE is written in C++. The current version is available for MAC OS X 10.8+ (64-bit) and Linux (32-bit and 64-bit). We provide statically linked binaries for non-commercial use on our website <http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Marcie>. Currently MARCIE comes as a command line tool. To install MARCIE, which is deployed as a simple archive, just copy the contained binary file `marcie` to a binary folder of your system, e.g. using the command `sudo cp marcie /usr/local/bin`.

Alternatively you can copy the file to an arbitrary folder of your choice, e.g. using the command `cp marcie your_favorite_folder`. In this case you have to add this folder to your PATH variable, e.g. by adding the line `export PATH=$PATH:your_favorite_folder` to the file `.bashrc` in your home folder. Now you can call `marcie` everywhere.

Features. MARCIE provides symbolic qualitative analysis techniques for Petri nets, and simulative, explicit approximative and symbolic exact numerical analysis techniques for stochastic Petri nets. Figure 1 shows the tool’s architecture. A closer look at its components and its In and Outs may give a first impression of what MARCIE can do for the user. The *symbolic engines* are

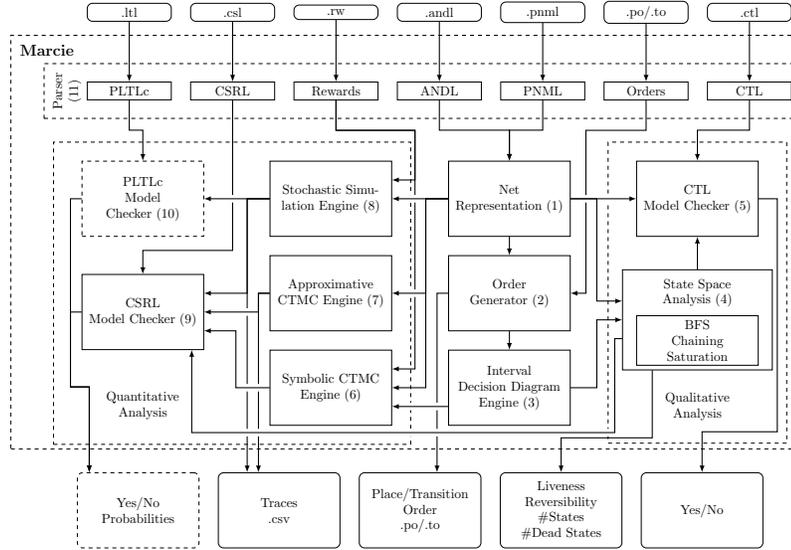


Figure 1: MARCIE’s architecture

based on *Interval Decision Diagrams* (IDD). State sets are represented by their characteristic interval logic functions which are encoded by IDDs. This allows to efficiently implement all qualitative state space based analyses.

For quantitative analysis MARCIE implements a *multi-threaded on-the-fly computation of the underlying CTMC*. It is thus less sensitive to the number of distinct rate values as approaches based on, e.g., Multi-Terminal Decision Diagrams.

Further, MARCIE applies *heuristics for the computation of static variable orders* to achieve small IDD representations.

In summary, MARCIE’s symbolic engines take care of all aspects affecting efficient symbolic analysis of bounded generalized stochastic Petri nets and outperforms comparable tools for technical case studies as well as for models from systems biology. For a comparison see [SH09, ST10, HST09, Sch10, Sch14]

The analysis of possibly unbounded stochastic Petri nets is supported by two engines. The *explicit approximative engine* builds on fast adaptive uniformisation and always computes a finite subset of the possibly infinite CTMC. The *simulative engine* provides statistical model checking of extended stochastic Petri nets. Therefore it creates paths through the CTMC, which could then be analysed.

2 Background

MARCIE is a model checker for qualitative Petri nets and several extensions of stochastic Petri nets. We assume that the reader is familiar with the basics of modelling and analysis of Petri nets. In this section we give a brief overview on the supported net classes, and their specification and the model checking capabilities the tool supports.

2.1 Net classes

The following Petri net classes are currently supported by MARCIE. They represent a subset of those supported by our modelling framework Snoopy [HHL⁺12b]. We recommend to use this tool to specify the models which should be analysed with MARCIE. Therefor Snoopy exports model descriptions to ANDL – a concise, but human-readable format, see Section 2.3.

Qualitative Petri nets (QPN). QPN comprise standard *Place/Transition nets* (P/T nets) and *extended Petri nets* ($\mathcal{X}PN$). They do not involve any timing aspects; so they allow a purely qualitative modelling of, e.g., biomolecular networks. Tokens may represent molecules or abstract concentration levels [HGD08]. $\mathcal{X}PN$ enhance standard Petri nets by four special arc types: read arcs (often also called test arcs), inhibitor arcs, equal arcs, and reset arcs, see [HRSR08], [RMH10] for details.

Stochastic Petri nets (SPN). This net class extends QPN by assigning to transitions exponentially distributed waiting times, specified by firing rate functions. A rate function is generally state-dependent; it can be an arbitrary arithmetic function deploying the pre-places of a transition as integer variables and user-defined, real-valued constants (often called parameters). Pre-places can be associated with transitions by special modifier arcs [RMH10]. They may modify the transition’s firing rate, but do not have an influence on the transition’s enabledness.

SPN have been extended to *Generalised Stochastic Petri Nets* ($GSPN$), which in turn have been extended to *Deterministic and Stochastic Petri Nets* ($DSPN$). Both are included in our net class of *Extended Stochastic Petri Nets* ($\mathcal{X}SPN$) [HLGM09]. $\mathcal{X}SPN$ supports the four special arc types as known for QPN , and three special transition types: immediate transitions (zero waiting time), deterministic transitions (deterministic waiting time, relative to the time point where the transition gets enabled), and scheduled transitions (scheduled to fire, if any, at single or equidistant, absolute time points).

Rewards. An SPN , $GSPN$ or $\mathcal{X}SPN$ can be augmented with a reward function – a function defining a numerical value for its transitions or its reachable states (markings). This value is accumulated continuously over time for states, and instantaneously with every firing for transitions. The reward function may

represent some kind of costs or gain. In the literature, the result of such an augmentation is called a Stochastic Reward Net (SRN), see e.g. [CBC⁺93].

Analysis. Depending on the net class, MARCIE offers different analysis methods. As a rule of thumb, it holds: as more general the net class of the investigated model, as more restricted are the analysis capabilities.

For instance, the use of modelling means specific for \mathcal{XSPN} destroys the Markov property of \mathcal{SPN} . This prohibits the use of established numerical standard algorithms. The same holds for \mathcal{SPN} if the investigated model possesses an infinite state space, and also for reward-augmented \mathcal{GSPN} models. MARCIE offers three different analysis engines which we present in Section 3. For each engine we outline the restrictions which have to be considered for its use.

2.2 Model checking

The mentioned net classes allow to apply a batch of established analysis methods which range from the state space construction of bounded (finite state space) \mathcal{QPN} to the generation of execution traces of even unbounded \mathcal{XSPN} . Based on these, lets say, common analysis methods MARCIE implements a couple of model checking techniques.

Model checking [CE82] denotes the application of an algorithm, the model checker, to evaluate for a formal model description, e.g. a Petri net, the truth of a model property formalized using some propositional temporal logic in a fully automatic fashion. Under a temporal logic we understand an extension of a proposition logic by so-called temporal or tense operators. This allows to express qualitative properties as “eventually p holds in a state”. Pnuelli considered temporal logics with a linear time semantics (LTL) to reason about the properties of reactive systems [Pnu77a]. In the linear time model all possible executions of the model have to fulfil the given specification.

A different approach is to consider branching time, where it is possible in each state to choose between different future evolutions. To specify properties with branching time semantics Clarke and Emerson introduced the Computation Tree Logic (CTL) [CE82].

For stochastic Petri nets and the mentioned extensions we can apply extensions as the Continuous Stochastic Logic (CSL) [ASSB00], and the Continuous Stochastic Reward Logic (CSRL) [BHHK00], or the Probabilistic Linear Time Logic with constraints (PLTLc) [DG08]. CSL and CSRL have branching time semantics and represent adaptations of CTL while PLTLc has linear time semantics and represents an adaption of LTL.

With these logics we are able to ask for instance for the probability that the quantitative property “within time t , p is true in a state” holds.

MARCIE supports model checking of the branching time logics CTL, CSL, and CSRL based on the finite state space; and the linear time logic PLTLc based on trace generation via stochastic simulation. We will give further explanations in Section 4.

P3 and P12.

To specify the input file use the option

```
--net-file=<filename> sets the file from which the net should
be loaded.
```

The given ANDL net specification must obey the following grammar:

```
net = [ netclass ] '[' STRING ']' '{'
      'functions:' (function)*
      'constants:' [valuesets] (constant)*
      'places:' (place)+
      'transitions:' transitions
    '}' .

netclass
= 'qpn' | 'spn' | 'gspn' | 'xspn' .
```

The net specification contains possibly empty sets of function and constant definitions and non empty sets of places and transitions. The set of arcs is implicitly encoded by the conditions and updates of the transition definitions. An ANDL file is allowed to contain comments in C/C++ style.

```
/*
   A multi line
   C style comment
*/

// A single line C++ style comment
```

Functions. The net can possess a set of function template definitions which can be used to realize more complex functions. The parameter list specifies the variables which are replaced by either values, constants, place names or even arithmetic functions, depending on the context in which the function template is instantiated.

```
function
= name '(' parameter_list ')' '=' arithmetic_function ';' .
```

The right hand side of a function definition can be an arbitrary arithmetic function.

Arithmetic Functions. Arithmetic functions must conform to the following grammar:

```

arithmetic_function
= expression .

expression
= term
| expression '+' term
| expression '-' term .

term
= factor
| term '*' factor
| term '/' factor
| term '^' factor .

factor
= VARIABLE
| REAL
| INT
| '(' expression ')'
| pow '(' expression ',' expression ')'
| unaryop '(' expression ')'
| naryop '(' nary_arg (',' nary_arg)* ')'
| FUNCTION '(' ( expression (',' expression)* ) ')' .

naryop
= 'min' | 'max' | 'prod' | 'sum' .

nary_arg
= expression | '@{' REG_EXPR '}' .

unaryop
= '-' | 'sqr' | 'sqrt' | 'floor' |
  'ceil' | 'abs' | 'log' | 'log10' |
  'exp' | 'cos' | 'acos' | 'sin' | 'asin' |
  'tan' | 'atan' .

```

VARIABLE must be a valid name.

FUNCTION must be a defined function name.

REAL must be a real number.

INT must be an integral number.

REG_EXPR is a regular expression with Perl syntax.

A valid variable or function name starts with a letter, followed by an arbitrary character sequence containing letters, digits or underscore (.). A variable may represent a model constant or a place name.

Functions can be used to specify initial token values, arc weights, parameters and rates of transitions, rewards functions, and parameters and atomic propositions in CTL/CSRL/PLTLc formulas. We distinguish arbitrary arithmetic functions (`arithmetic_function`) and constant functions (`const_function`). Arithmetic functions are allowed to contain also place names as variables and thus allow to define state-dependent functions.

A constant function syntactically equals an arithmetic function, but it is not allowed to use place names as variables. The function value will be computed at the time the function is parsed.

Regular expressions. The arguments of the n-ary functions `sum`, `prod`, `min` and `max` can be specified by means of regular expressions. A regular expression will be replaced by the set of variables whose name match the given expression. The implementation of regular expressions in MARCIE deploys the boost library with the Perl syntax, see <http://www.boost.org> for more details.

Constants. A constant must be decorated with a name and a type. Constants can be logically grouped by the group name that is prefixed. The group is optional, if a constant doesn't have a group it belongs to the same group as the constant above. The constant can have a single value or a set of values. A set of values must correspond to the `valuesets`, i.e., a vector of values must have the same number of entries as the `valuesets` and a map of values must refer to elements of the `valuesets`. In the net definition it is possible to omit the constant value. In this case it must be defined when starting the tool.

```
valuesets
= 'valuesets' '[' name (':' name)* ']' .

constant
= [ group':' ] ('int' | 'double') name [ '=' (const_function | const_values) ] ';' .

group
= name .

const_values
= '[' (const_values_vector | const_values_map) ']' .

const_values_vector
= const_function (':' const_function)* .
```

```
const_values_map
= name '=' const_function (':' name '=' const_function)* .
```

Example. We define the following two functions:

```
functions:
  f0(x1,x2,x3,x4) = x1 + x2 + x3 + x4 ;
  f1(x1,x2,x3,x4) = max( 1 , np / f0(x1,x2,x3,x4)) ;
```

These functions are used in the running example to specify state-dependent rate functions. Then we will replace parameters x_i by pre-places of the related transitions. The variable np is no parameter and will refer to a constant name.

We further define two integer constants and two value sets. Both constants belong to the same group. The value of np is computed from that of N . The default value of N is 2 and the second value is 4.

```
constants:
  valuesets[Main:VSet1]
  all:
    int N = [2:4] ;
    int np = 3*N/2 ;
```

The following option overrides constant values, given in the ANDL specification.

```
--const <c1=v1,c2=v2,. . .,cn=vn> sets the values for the
given constants.
```

The value of the constant N will be set to 10. Its default value of 2 will be overwritten.

```
--const N=10
```

A value set for a group of constants can be set in the following way.

```
--const all=VSet1
```

Places. The net must contain at least one place. A place must have a unique identifier, a name and an initial number of tokens (which can be zero). The initial number of tokens can be defined as a constant value or as an arithmetic function with the defined constants being used as variables.

```
place
= name '=' const_function ';' .
```

Example. We define four places. Place $P1$ will initially carry N tokens, with N being a constant introduced above.

places:

```
P1 = N ;
P1wM1 = 0 ;
P1M1 = 0 ;
M1 = 3 ;
```

Transitions. As for places MARCIE assumes that the set of transitions is not empty. MARCIE knows the following keywords to distinguish the different transition types of \mathcal{XSPN} :

- stochastic (the default)
- immediate
- deterministic
- scheduled.

The use of these keywords creates a partition of the actual transition set. Independent of its type, a transition must have a unique name. A transition may define a set of conditions which must be fulfilled in a marking to allow the transition to fire. The effect of the firing must be specified by a set of updates. Further a specific function may be associated to the transition, whose semantics is specified by the transition's type. **A transition without a function will get the constant value 1.**

transitions:

```
= (( transition_type ':' ) transition+ )+ .
```

transition_type:

```
= [ stochastic | immediate | deterministic | scheduled ] .
```

transition

```
= name ':' (conditions) ':' (updates) (':' transition_function) ';' .
```

conditions

```
= condition ('&' condition)* .
```

condition

```
= '[' PLACE '>=' const_function ']' //read arc
| '[' PLACE '<' const_function ']' //inhibitor arc
| '[' PLACE '=' const_function ']' //equal arc
| '[' PLACE ']' //modifier arc .
```

updates

```
= update ('&' update)* .
```

```
update
= '[' PLACE [ '+' | '-' | '=' ] const_function ']' .
```

Places which are given in the condition list are considered as the pre-places of the transitions. If a condition of a transition only specifies a place name, this place can be used in the function without affecting the enabledness of the transitions. Snoopy offers *modifier* arcs to describe this relation.

An upper bound for the tokens on a place in a given condition implicitly defines an inhibitor arc in the net.

A lower bound for the tokens on a place in a given condition implicitly defines a read arc in the net.

A specific number of tokens on a place in a given condition (in Snoopy an equal arc) implicitly defines a read arc and an inhibitor arc in the net.

Transition-related functions. There are different function types which can be used to specify the behaviour of a transition dependent of the transition's type.

1. The transition specific functions are generally specified by means of arithmetic functions.
2. For the analysis of biochemical networks MARCIE supports also special pattern to define for stochastic transitions special rate functions with **mass-action** or **bio-level** kinetics. The related kinetic constant can be specified by a constant function.

If the function field is empty the value is implicitly 1.

```
transition_function
= arithmetic_function
  | 'MassAction(' const_function ')'
  | 'BioLevel(' const_function ')'
```

Attention: The alert reader will have noticed, that in contrast to constant definitions, function templates do not have a type. This is reasonable as their result type depends on the context in which they are used. When a function is used to define a value within conditions, updates, in a place or a constant definition, the result type will be **integer**, **unsigned integer** in particular. In the context of transition related functions we deal with type **double**.

Example We define two stochastic transitions. The rate function of transition *tP3* is state-dependent and makes use of one of the specified function templates, the rate of transition *tP3M2* is a constant value of 0.5.

```

tP3 : [P2] & [P1] & [P12] & [0 < P3 ]
      : [P3M2 + 1] & [P3 - 1] : P3 * f1(P1 , P2 , P3 , P12 ));

tP3M2 : [M2] & [0 < M2] & [0 < P3M2]
        : [P3M2 - 1] & [P3s + 1] : 0.5 ;

```

Every definition of a function template, a constant, a place, or a transition must be closed by a semicolon.

Example An ANDL description of the FMS system given in Fig. 2. Due to the lack of state-dependent arc weights, we modelled all grey coloured transitions by $max = 10$ different transitions, each enabled for an unique number of tokens. The net is a correct model of the FMS system if $N \leq max$.

```

gspn [ FMS ] {

functions:

    f0(x1,x2,x3,x4) = x1 + x2 + x3 + x4 ;
    f1(x1,x2,x3,x4) = max( 1 , np / f0(x1,x2,x3,x4) ) ;
    rate(x) = x * f1(P1,P2,P3,P4) ;

constants:
    valuesets[Main:VSet1]
    all:
        int N = [2:4] ;
        int np = 3*N/2 ;

places:

    P12wM3 = 0;
    M3 = 2;
    P12M3 = 0;
    P3s = 0;
    P3 = N;
    P3M2 = 0;
    P12s = 0;
    P2s = 0;
    P2 = N;
    P1s = 0;
    P1 = N;
    M2 = 1;
    P2wM2 = 0;
    P2M2 = 0;
    P2d = 0;
    P2wP1 = 0;
    P1wP2 = 0;
    P12 = 0;
    P1wM1 = 0;
    M1 = 3;
    P1M1 = 0;
    P1d = 0;

transitions:

stochastic:

    tP3 : [P2 ] & [P1 ] & [P12 ] : [P3 - 1] & [P3M2 + 1] : rate(P3);
    tP3M2 : [1 <= M2 ] : [P3s + 1] & [P3M2 - 1] : 0.5 ;
    tP1 : [P3 ] & [P2 ] & [P12 ] : [P1 - 1] & [P1wM1 + 1] : rate(P1);
    tP2 : [P3 ] & [P1 ] & [P12 ] : [P2 - 1] & [P2wM2 + 1] : rate(P2);

```

```

P3s_eq_1 : [P3s = 1] : [P3s - 1] & [P3 + 1] : 1/60 ;
P3s_eq_2 : [P3s = 2] : [P3s - 1] & [P3 + 1] : 1/60 ;
P3s_eq_3 : [P3s = 3] : [P3s - 3] & [P3 + 3] : 1/60 ;
P3s_eq_4 : [P3s = 4] : [P3s - 4] & [P3 + 4] : 1/60 ;
P3s_eq_5 : [P3s = 5] : [P3s - 5] & [P3 + 5] : 1/60 ;
P3s_eq_2 : [P3s = 2] : [P3s - 2] & [P3 + 2] : 1/60 ;
P3s_eq_6 : [P3s = 6] : [P3s - 6] & [P3 + 6] : 1/60 ;
P3s_eq_7 : [P3s = 7] : [P3s - 7] & [P3 + 7] : 1/60 ;
P3s_eq_8 : [P3s = 8] : [P3s - 8] & [P3 + 8] : 1/60 ;
P3s_eq_9 : [P3s = 9] : [P3s - 9] & [P3 + 9] : 1/60 ;
P3s_eq_10 : [P3s = 10] : [P3s - 10] & [P3 + 10] : 1/60 ;
tP1M1 : : [M1 + 1] & [P1M1 - 1] & [P1d + 1] : P1M1 / 4 ;
tP2M2 : : [M2 + 1] & [P2M2 - 1] & [P2d + 1] : 1/6 ;
P1s_eq_1 : [P1s = 1] : [P1s - 1] & [P1 + 1] : 1/60 ;
P1s_eq_2 : [P1s = 2] : [P1s - 2] & [P1 + 2] : 1/60 ;
P1s_eq_3 : [P1s = 3] : [P1s - 3] & [P1 + 3] : 1/60 ;
P1s_eq_4 : [P1s = 4] : [P1s - 4] & [P1 + 4] : 1/60 ;
P1s_eq_5 : [P1s = 5] : [P1s - 5] & [P1 + 5] : 1/60 ;
P1s_eq_6 : [P1s = 6] : [P1s - 6] & [P1 + 6] : 1/60 ;
P1s_eq_7 : [P1s = 7] : [P1s - 7] & [P1 + 7] : 1/60 ;
P1s_eq_8 : [P1s = 8] : [P1s - 8] & [P1 + 8] : 1/60 ;
P1s_eq_9 : [P1s = 9] : [P1s - 9] & [P1 + 9] : 1/60 ;
P1s_eq_10 : [P1s = 10] : [P1s - 10] & [P1 + 10] : 1/60 ;

P2s_eq_1 : [P2s = 1] : [P2s - 1] & [P2 + 1] : 1/60 ;
P2s_eq_2 : [P2s = 2] : [P2s - 2] & [P2 + 2] : 1/60 ;
P2s_eq_3 : [P2s = 3] : [P2s - 3] & [P2 + 3] : 1/60 ;
P2s_eq_4 : [P2s = 4] : [P2s - 4] & [P2 + 4] : 1/60 ;
P2s_eq_5 : [P2s = 5] : [P2s - 5] & [P2 + 5] : 1/60 ;
P2s_eq_6 : [P2s = 6] : [P2s - 6] & [P2 + 6] : 1/60 ;
P2s_eq_7 : [P2s = 7] : [P2s - 7] & [P2 + 7] : 1/60 ;
P2s_eq_8 : [P2s = 8] : [P2s - 8] & [P2 + 8] : 1/60 ;
P2s_eq_9 : [P2s = 9] : [P2s - 9] & [P2 + 9] : 1/60 ;
P2s_eq_10 : [P2s = 10] : [P2s - 10] & [P2 + 10] : 1/60 ;

tP12 : [P3 ] & [P2 ] & [P1 ] : [P12wM3 + 1] & [P12 - 1] : rate(P12);
tP12M3 : : [M3 + 1] & [P12M3 - 1] & [P12s + 1] : P12M3 ;
P12s_eq_1 : [P12s = 1] : [P12s - 1] & [P2 + 1] & [P1 + 1] : 1/60 ;
P12s_eq_2 : [P12s = 2] : [P12s - 2] & [P2 + 2] & [P1 + 2] : 1/60 ;
P12s_eq_3 : [P12s = 3] : [P12s - 3] & [P2 + 3] & [P1 + 3] : 1/60 ;
P12s_eq_4 : [P12s = 4] : [P12s - 4] & [P2 + 4] & [P1 + 4] : 1/60 ;
P12s_eq_5 : [P12s = 5] : [P12s - 5] & [P2 + 5] & [P1 + 5] : 1/60 ;
P12s_eq_6 : [P12s = 6] : [P12s - 6] & [P2 + 6] & [P1 + 6] : 1/60 ;
P12s_eq_7 : [P12s = 7] : [P12s - 7] & [P2 + 7] & [P1 + 7] : 1/60 ;
P12s_eq_8 : [P12s = 8] : [P12s - 8] & [P2 + 8] & [P1 + 8] : 1/60 ;
P12s_eq_9 : [P12s = 9] : [P12s - 9] & [P2 + 9] & [P1 + 9] : 1/60 ;
P12s_eq_10 : [P12s = 10] : [P12s - 10] & [P2 + 10] & [P1 + 10] : 1/60 ;

immediate:
tM1 : : [P1wM1 - 1] & [M1 - 1] & [P1M1 + 1] : 1 ;
tP1e : : [P1s + 1] & [P1d - 1] : 0.8 ;
tP1j : : [P1wP2 + 1] & [P1d - 1] : 0.2 ;
tM2 : : [M2 - 1] & [P2wM2 - 1] & [P2M2 + 1] : 1 ;
tP2j : : [P2d - 1] & [P2wP1 + 1] : 0.4 ;
tP2e : : [P2s + 1] & [P2d - 1] : 0.6 ;
tx : : [P2wP1 - 1] & [P1wP2 - 1] & [P12 + 1] : 1 ;
tM3 : : [P12wM3 - 1] & [M3 - 1] & [P12M3 + 1] : 1 ;
}

```

Rewards. MARCIE allows to specify reward structures¹ to define reward functions. When analysing the model (Section 4) it can be augmented by one such reward function. The reward structures must be specified in a separate

¹syntactically in a similar way as the probabilistic symbolic model checker PRISM

reward file. A reward file is allowed to contain multiple reward structures. Each reward structure must have a unique name and comprises a non-empty list of state or transition reward items. Anonymous reward structures are forbidden. A reward file is allowed to contain comments in C/C++ style.

```

reward_structures
= (rewards_structure)+ .

rewards_structure
= 'rewards' '[' REWARD_NAME ']' '{' (reward_item)+ '}' .

reward_item
= transition_reward_item
| state_reward_item .

state_reward_item
= guard ':' reward_function ';' .

transition_reward_item
= '[' TRANSITION ']' guard ':' reward_function ';' .

reward_function
= arithmetic_function .

guard
= interval_logic_expression .

interval_logic_expression
= interval_logic_expression binop interval_logic_expression
| '!' interval_logic_expression
| 'true'
| 'false'
| PLACE cmp const_function
| const_function '<=' PLACE '<' const_function.

binop
= '&' | '|' | '->' | '<-' | '<->' .

cmp
= '=' | '!=' | '>=' | '>' | '<=' | '<' .

REWARD_NAME is a valid identifier

TRANSITION is a valid transition name

PLACE is a valid place name, or a valid constant name

```

which can be substituted with a place name

The guards of reward structure items are defined by means of interval logic expressions [Tov08].

Example. We define a reward structure representing the number of tokens on place P1

```
/* number of tokens on P1 */
rewards [P1]{
  true : P1;
}
```

We define a reward structure for the throughput of machine3.

```
/* throughput of machine3 */
rewards [throughput_m]{
  [tP3] true : 1;
}
```

We achieve the same with the following reward structure.

```
/* throughput of machine3 by means of state rewards!
   for each state, which enables tP3, the reward is the
   transition rate
*/
rewards [tP3]{
  P3 > 0 : rate(P3);
}
```

As it can be seen in the example, we can reuse the function templates from the model specification to define the reward measures. To load a set of reward structure definitions from a file, the following option has to be used.

```
--reward-file=<filename> sets the file from which the reward
structures should be read.
```

Place and transition-specific reward structures as "P1" and "tP3" are generated automatically and can be accessed by the name of the related place or transition.

The following option suppresses the implicit generation of reward structures.

```
--suppress-implicit-rewards no implicit reward structures
for places and transitions
```

To output certain registered reward structures MARCIE offers the option

```
--list-rewards=<r1:r2:...:rn> write the listed reward
structures.
```

The entry `ri` can be the concrete name of a reward structure or a regular expression. To e.g. list all reward structures type `--list-rewards=@.*@`

MARCIE offers a variety of qualitative and quantitative analysis methods. Most of these methods are based on the knowledge of the set of reachable states and can only be applied to bounded Petri nets. For the time being MARCIE does not implement a **boundedness check**. Just very simple cases, where transitions have no pre-places, will be considered. Thus, the user has to ensure the boundedness of the Petri net if he wants to deploy the related analyses.

3 Engines

MARCIE offers quantitative analysis of stochastic Petri nets and various extensions, which is based on the computation of the transient and the steady state probability distribution or of single traces. It provides an exact and an approximative numerical engine, and a simulative engine. The engines support different net classes and offer different types of analysis which are explained in the following.

3.1 Qualitative Symbolic Engine

The qualitative analysis is based on the set of reachable states. If not suppressed by starting the simulative (3.3) or the approximative (3.4) engine, the state space generation will be automatically started if a net-file has been specified.

3.1.1 State space generation

MARCIE provides three different base algorithms for this purpose.

`--rs-algorithm=<1,2,3>` allows to choose between the different state space generation algorithm.

1. Common Breadth-First Search (BFS): an iteration fires sequentially all transitions (according to the transition ordering) before adding the new states to the state space.
2. Transitions chaining: like BFS, but the state space is updated after the firing of each single transition.
3. Saturation algorithm (SAT): transitions are fired in conformance with the decision diagram, i.e. according to an ordering, which is defined by the variable ordering. A transition is saturated if its firing does not add new states to the current state space. Transitions are bottom-up saturated (i.e. starting at the terminal nodes and going towards the root). Having fired a given transition, all preceding transitions have to be saturated again, either after a single firing (single) or the exhausted firing (fixpoint) of the current transition.

The efficiency of the last two algorithms depends on the transition ordering. In most cases SAT outperforms the other algorithms. Thus it is the default algorithm. There are two further options to configure the SAT algorithm.

`--sat-single` triggers the SAT algorithm to saturate preceding transitions after a single firing. Per default this option is not activated.

To be efficient, saturation requires a suited transition order, which is obtained by the given place order; see [ST10]. To prevent the automatic calculation of such a transition order, the following option can be set.

`--sat-rs-noown` setting this option suppresses the computation of a Saturation specific transition order.

MARCIE represents the state space symbolically using Interval Decision Diagrams. There are several aspects which may influence the performance and which can be configured using the following options.

`--memory-dd=<1,2,3,4,5>` allows to set the available memory required to store IDD nodes and arcs.

The number of iterations and the number of IDD nodes which is required to generate the state space may be influenced by the order of transitions. Thus MARCIE implements pre-ordering of the transitions and offers options to handle the transition orders.

`--trans-order=<1,2,3,4,5,6,7,8>` allows to choose the transition ordering strategy

1. Plain order as given in the net file.
2. Randomly calculated order.
3. Order calculated according to Noack[Noa99] (default).
4. Order calculated according to Noack modified.
5. Saturation.
6. Reads a user-defined transition order from a file.
7. Lexicographic order
8. Natural alpha-numeric order

`--torder-read=<filename>` specifies the file *filename* from which the user-defined transition order should be read.

`--Torder-write=<filename>` writes the used order to the file *filename*.

A crucial point for the size of a decision diagram is the order of variables and thus the order of the places. MARCIE applies heuristics to compute static variable orders based on the Petri net structure. In most cases these orders result in very small-sized decision diagrams [SH09].

`--place-order=<1,2,3,4,5,6,7,8,9>` allows to choose a place ordering strategy

1. Plain order as given in the net-file.
2. Reversed order as in the net-file.
3. Randomly calculated order.
4. Order calculated according to Noack (default).
5. Order calculated according to Noack modified.
6. Order derived from the given transition order.
7. Reads a user-defined place order from a file.
8. Lexicographic order
9. Natural alpha-numeric order

```
--porder-read=<filename> specifies the file filename from
which the user-defined place order should be read.
```

```
--Porder-write=<filename> writes the used place order to the
file filename.
```

Priority of transitions. When applying qualitative analysis, MARCIE ignores the type of transitions, which in fact distinguishes immediate, deterministic, scheduled (higher priority) and stochastic transitions. Thus qualitative analysis ignores priorities and may be performed on a state space different of that when using quantitative analysis of an \mathcal{XSPN} .

Example. Table ?? shows the influence of treating immediate transitions with a higher priority for the FMS system. When doing qualitative analysis MARCIE computes the state space S_{np} . Quantitative analysis is done on the state space S_p .

3.1.2 Standard Properties

To determine the Petri net's standard properties, MARCIE expects a bounded Petri net. **ATTENTION. MARCIE does NOT perform a sophisticated boundedness check!**

Boundedness degree. Boundedness implies the existence of an upper tokens bound for each place of the net. The maximal bound determines the boundedness degree of the given net. To output its value you can use the option

```
--max-val analyze the maximal possible values of tokens.
```

You can also write the place specific bounds to a file using the option

```
--Max-vals=<FILE> write the maximal number of tokens for
each place to FILE.
```

Reversibility A net N is reversible if for all reachable states S there exists a path leading back to the initial state.

$$\forall s \in S : s \xrightarrow{*} s_0$$

This is the case, if the reachability graph forms a strongly connected component.

```
--rev-check triggers the check for reversibility.
```

Dead states A state s is a dead state if it does not enable any transitions. In the reachability graph the related node does not have outgoing arcs.

`--dead-check` triggers the check for dead states.

`--Dead-states=<filename>` writes all dead states to a file as interval logic expressions.

`--Dead-trace=<filename>` writes a trace as sequence of transition names to *filename*.

Liveness A transition t is called live in state s if for all states reachable from s it is possible to reach a state s' where t is enabled.

$$\forall s \in S, \exists s' : s \xrightarrow{*} s' \wedge s' \in \text{enabled}(t).$$

In this case all terminal strongly connected components contain at least one state where t is enabled. A net N is live if all its transitions are live.

`--live-net-check` triggers the check for liveness of the net.

`--Live-trans=<filename>` writes the set of live transitions to the file *filename*.

`--Not-live-trans=<filename>` writes the set of transitions which are not live to the file *filename*.

3.2 Exact Numerical Engine

MARCIE's exact numerical solvers are based on a symbolic on-the-fly multiplication of a matrix and a vector. The matrix entries will be recomputed anew when needed. This multiplication relies on a traversal of the state space-representing Interval Decision Diagram considering specific information of the Petri net transitions; see [Sch14] for more details. This approach becomes efficient when it is combined with a suitable truncation technique. We adapted the one introduced in [Par02]. The idea is to stop the traversal when reaching a node of a specified layer. Therefore it is necessary to store the needed information of indices and values of the potential path extension in these nodes. Computation speed and memory consumption increase when moving the truncation layer in direction of the decision diagram root as it is illustrated for the FMS system in Figure 3.

`--truncation-layer=<1, ..., |P| - 1>` sets the truncation layer to a value between 1, which is the layer on top of the terminal nodes, and the number of places minus one (root node). When not specified by the user, MARCIE sets the truncation layer to $\lceil |P| \cdot 0.50 \rceil$.

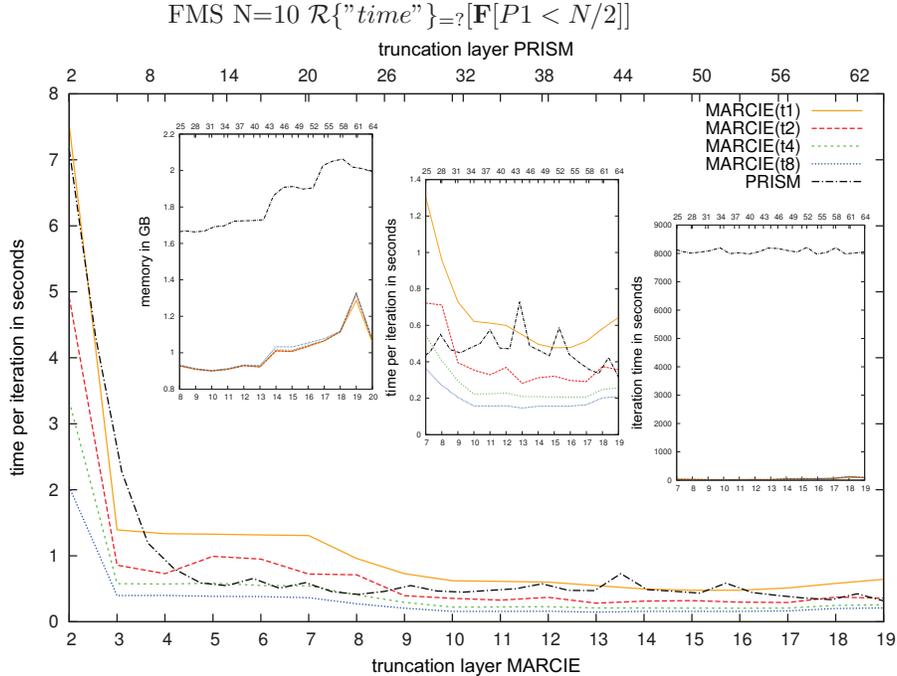


Figure 3: A comparison of MARCIE with the PRISM tool to illustrate the influence of an early truncation and multi-threading when doing model checking (taken from [Sch14]).

MARCIE uses the uniformization method [Ste94] to compute transient probabilities. The transient solver supports multi-threading. Figure 3 shows also the influence of this option.

`--threads=<n>` sets the number of threads to n .

Further it supports the following three iterative methods to solve a linear system of equations, for instance to compute steady state probabilities.

Jacobi method. The Jacobi method requires two vectors in the size of state space to store intermediate and final results. It **enables multi-threading**.

Gauss-Seidel method. MARCIE also offers a Gauss-Seidel solver. Unfortunately the implemented engine is not suited to realize an efficient Gauss-Seidel implementation. Thus the current Gauss-Seidel solver is very slow. It consumes more memory than the Jacobi solver although the method only requires one vector in the size of the state space. Currently, the solver does **not support multi-threading**.

Pseudo-Gauss-Seidel method. The Pseudo-Gauss Seidel method is an adaption of the Gauss-Seidel method, following the idea of [Par02]. The computation vector will be updated after the extraction of a consecutive block of matrix rows instead of each row. In general the method converges much faster than Jacobi and slower than Gauss-Seidel. It requires one computation vector and an additional subvector in the size of the biggest row block. The size of the row block depends on the truncation layer. Currently, the solver does **not support multi-threading**.

```
--lin-solver=<1,2,3> allows to choose the solver for a linear system of equations
```

1. Jacobi (default)
2. Pseudo-Gauss-Seidel
3. Gauss-Seidel

\mathcal{GSPN} support. MARCIE's numerical engine supports only stochastic and immediate transitions and actually does not like the immediate transitions. It has to consider all the vanishing states, since the on-the-fly multiplication does not allow a reduction. Thus, if possible, the \mathcal{GSPN} should be reduced to an \mathcal{SPN} as it is proposed in [MBC⁺95]. There are several options to print the CTMC.

```
--print-size prints number of states and state transitions.
```

```
--print-states prints the reachable states of the CTMC and their lexicographic index.
```

The lexicographic index of a state is used to access entries in a vector or a matrix which are related to the state. **The lexicographic index of a state depends on the used variable order!**

```
--list-rewards lists the defined reward structures.
```

```
--print-Reward-vector=<rs1>, ..., <rsN> prints a characterization of the specified reward structures in terms of states and their rewards.
```

The rewards structures must be specified by their names.

```
--print-ctmc prints the states, the rate matrix and the specified rewards.
```

Markovian approximation MARCIE's symbolic CSRL modelchecker needs to compute the joint distribution of time and accumulated reward and uses Markovian approximation, a technique to compute the joint distribution by applying standard techniques as transient analysis to a CTMC which approximates the behaviour of the actual Markov Reward Model (MRM). The accumulated reward will be encoded in the discrete states. This requires to replace the continuous accumulation of rewards by a discrete one. The discretized accumulated reward increases stepwise by a fixed value Δ . It requires y/Δ steps to approximately accumulate a reward of y . Currently MARCIE offers only an option to set the number of steps. The value for y is extracted from the given CSRL formula.

```
--steps=<number> sets the number of discretization steps for
the Markovian approximation.
```

Markovian approximation and thus symbolic CSRL model checking does **not support transition rewards** directly. But as mentioned earlier, it is possible to define a transition reward by means of state rewards.

Computation of probability distributions The symbolic engine of MARCIE's CSRL model checker can also be used to compute the following distributions:

1. the instantaneous transient probabilities

```
--transient=<t> computes the transient probabilities at time
point t.
```

2. the cumulative transient probabilities; as before but in addition one has to set option

```
--cumulative the transient solver computes the cumulative
transient probabilities.
```

3. the steady state probabilities

```
--steady-state computes the steady state probabilities.
```

4. performability

```
--performability=<rs>:<t>,<y> computes the joint distribu-
tion of time and accumulated reward for the reward structure
with name rs at time point t and for the reward bound y.
```

Evaluation and visualization of probability distributions As mentioned earlier the exact numerical engine considers all reachable states and thus computes a vector in the size of the reachable states. A result vector can be displayed using the option

```
--print-results prints the computed result vector.
```

Further it is possible to specify the formatter to be used to when printing the result vector.

```
--vector-format=<CSV,Compressed,Latex> prints a vector in the specified format.
```

```
--precision-results=<n> specifies the number of fractional digits. The default value is 2.
```

If the vector should be written to a file one can use the option

```
--export-results=<File> writes the result vectors to FILE.
```

Observers. To permit a further evaluation of the computed distributions MARCIE implements an observer concept. As an observer we understand every state-dependent function (defined by means of the marking of some places) whose value evolution is observed over time or at specific time points. The function defined by a reward structure can be interpreted as an observer. The following options enable to define and handle observer-based evaluation.

```
--observe=<r1:r2:...:rN> outputs the distribution of possible values of reward structures.
```

This option starts the computation of the distribution of the possible values (e.g. token values of places) for the specified reward structures. For each reward structure $r \in \{r1, \dots, rN\}$ and the given probability distribution $dist$ MARCIE computes the value $dist_{r=v}$ for $0 \leq v \leq \max\{i \mid s \in S \wedge r(s) = v\}$ with

$$dist_{r=v} = \sum_{s \in S: r(s)=v} dist_s.$$

There are three different ways to specify the reward structures to be observed:

1. Refer the name of a registered reward structure. For all places and all transitions MARCIE defines implicitly a reward structure representing the number of tokens, and the rate respectively.
2. Define an arbitrary function for which MARCIE creates an anonymous reward structure.
3. Define a regular expression which MARCIE expands to the set of matching reward structure names.

Example. We define the following observers,

```
--observe=P1:P1+P2:@{M.*}@
```

where P1 refers the implicit reward structure for place P1, P1+P2 specifies an anonymous reward structure representing the token sum on place P1 and P2, and the regular expression @{M.*}@ matches the reward structures M1, M2 and M3.

The computed distribution vectors can be exported again with `--export-results=<FILE>` possibly combined with `--vector-format=<CSV,Compressed,Latex>`.

It is possible to export the evolution of the expectation of observer values during a transient analysis.

```
--export-exp-evolution=<FILE> writes the evolution of the expectation in CSV format to FILE
```

```
--single-file writes the results into a single file instead of one file <FILE>_ri.csv per specified reward structure ri
```

If not suppressed by the option `--single-file`, MARCIE creates the csv-file `<FILE>_ri.csv`, if `ri` is the *ith* specified reward structure.

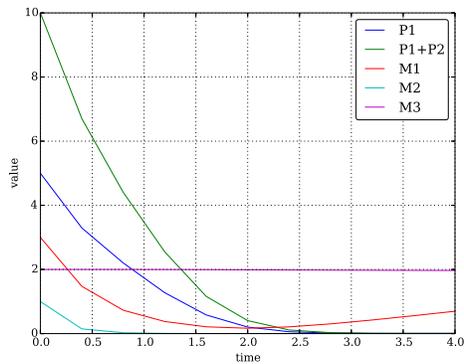


Figure 4: The evolution of the expected value of the observers specified by means of `--observe=P1:P1+P2:@{M.*}@`.

It is further possible to export the evolution of the distribution of the observer values during a transient analysis with

```
--export-dist-evolution=<FILE> writes the evolution of the distribution of observer values in the gnuplot format to FILE.
```

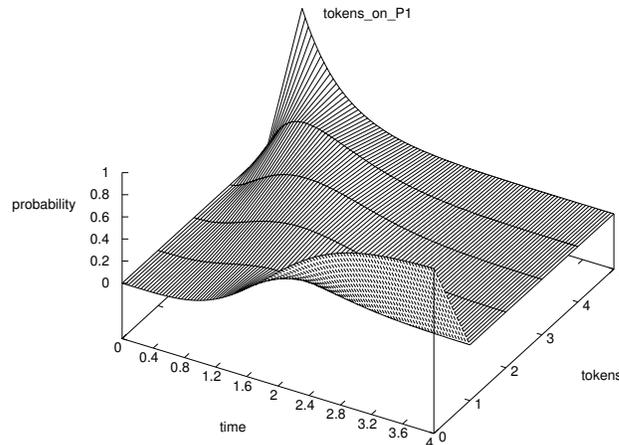


Figure 5: The evolution of the token distribution of place P1 up to time point four. The plot can be created with `gnuplot`'s `splot` command. MARCIE creates automatically the related `gnuplot` input file `<FILE>_ri.gpl`, where `ri` is the i th reward structure.

The resulting data files can be used to create diagrams as shown in Fig. 4 and Fig. 5 for the place $P1$ of the FMS model with $N = 5$.

If the values range of a reward structure is too large to be visualized it may be helpful to use the option

```
--normalize normalizes the reward values by the maximal value.
```

To plot the complete probability distribution you can additionally use the option

```
--export-state-dist-gnuplot writes the probability distribution evolution over time in the gnuplot format to FILE.
```

The generation of such kind of time dependent data requires to interrupt the transient analysis at certain time points, which can be specified with the option

```
--split-interval=<s1:t1,s2:t2,...,sn:tn> breaks the specified time interval [0,t] into n sub-intervals. Within the ith sub-interval the transient analysis is broken into si steps.
```

Strength The exact engine can be used to compute complete probability distributions.

Restrictions With regard to transient and steady state analysis engine can be deployed with **bounded SPN** and **GSPN** models. The computation of performability is restricted to **bounded SPN** models.

3.3 Simulative Engine

MARCIE includes a simulation engine.

```
--simulative switches the simulative engine on.
```

It incorporates three stochastic simulation algorithms (SSA).

Optimized direct method (ODM) The direct method was introduced by Gillespie in [Gil77]. It's an exact simulation method, i.e., it creates traces through the corresponding CTMC. The computational complexity is $\mathcal{O}(|T|)$. There where several improvements over the years, e.g., Cao et al introduced the dependency graph to minimize recomputing propensities in [CLP04]. Computing a discrete random variable for selecting the next transition to fire, can be done in different ways, e.g., using linear search as in the original algorithm by Gillespie, see [MS11] for a good overview. In MARCIE we use the 2-D search as presented in [MS11], which gives a good performance for smaller models, as well as for larger models. The computational complexity reduces to $\mathcal{O}(\sqrt{|T|})$.

Next reaction method (NRM) The next reaction method introduced by Gibson & Bruck in [GB00] is an adaptation of Gillespie's first reaction method. It uses the dependency graph to minimize recomputing propensities and computes the time at which each transition will fire directly. These times are stored in an indexed priority queue, implemented with a binary heap. The transition with the lowest time is on top of it. So selecting the next transition to fire has $\mathcal{O}(1)$ and the overall computational complexity is $\mathcal{O}(\log_2|T|)$.

τ -leaping method (TAU) The τ -leaping method [Gil01] is an approximative stochastic simulation algorithm, because it subsumes several steps in the simulation into one step if possible, i.e., some states may be *leaped*. This algorithm is useful for simulating stiff systems, i.e., some transitions are much faster than others.

Parameter-free method The parameter-free method isn't a stochastic simulation algorithm in the sense of the above. It aims at approximation of the widely used mass-action kinetics without knowledge of the kinetic parameters. It uses a fixed time step and at each step all transitions fire (if enabled) one after the other in random order. Each transition fires concurrently to itself depending on its enabledness degree.

```
--sim-algorithm=<1,2,3,4> allows to choose the stochastic
simulation algorithm
```

1. optimized direct method (default)
2. next reaction method
3. τ -leaping method
4. parameter-free method

The computation of results of a specific accuracy requires in general to create a huge number of simulation runs (traces) [HRSS10]. The number of required simulation runs can be determined using the confidence interval method [SM08]. The user can specify the confidence interval by defining the confidence level, usually 0.95 or 0.99, and the estimated accuracy, *e.g.*, 1×10^{-3} or 1×10^{-4} .

```
--sim-confidence=<c> sets the confidence to c, which can be
an arbitrary real value between 0 and 1 . The default value is
0.99.
```

```
--sim-accuracy=<a> sets the accuracy to a, which can be an
arbitrary real value. The default value is 0.01.
```

MARCIE calculates the required number of simulation runs to achieve this confidence interval. Of course the user can set the number of simulation runs manually.

```
--sim-runs=<n> sets the number of simulation runs to n. The
default value is 1.
```

Simulation further requires to specify a time bound - the simulation time. When the simulator reaches this time, the generation of the current trace will be stopped. **Attention:** We are speaking of simulation time, not runtime!

```
--sim-stop=<t> sets the simulation stop time t, which can be
an arbitrary positive real value.
```

It arises sometimes that one is in need of a forerun of t_0 time units, *e.g.*, an initialization phase which need not to be recorded. In such a case the simulation start time can be set to t_0 , at which $t_0 < t$ of course. Internally the simulation starts always at $t = 0$, but the recording of the values starts at t_0 .

```
--sim-start=<t0> sets the simulation start time t0, which can
be an arbitrary positive real value, the default value is 0.
```

The simulation output is characterized by the averaged number of tokens on each place over time. The generated traces can be written to a file in CSV format.

```
--export-results=<filename> writes the created traces to the
file filename.
```

If one is interested in a subset of places, the following option can be used to specify the desired places.

```
--sim-export-selection=<regex> all places that are matched
by the regular expression regex will be exported.
```

The simulative engine exports only the initial marking and the final marking at t in the default setting. In order to get more results there is an option to configure the number of steps for the output of results.

```
--split-interval=<s1:t1,s2:t2,...,sn:tn> specifies how
many simulation steps in the interval [start,stop] the simulator
should output for the results.
```

MARCIE is able to read and check simulation traces from a file in *CSV* format.

```
--sim-input-file=<filename> reads the simulation traces
from the file filename.
```

```
--sim-input-separator=<s> specifies the separator s (de-
fault: ',') used in the CSV file.
```

MARCIE uses the Mersenne Twister (MT) [MN98] pseudo-random number generator (PRNG). One can specify the seed for the MT-PRNG used in the simulative engine. If the option is not specified the seed is generated automatically by MARCIE.

```
--seed=<n> seed for pseudo-random number generator (0 = au-
tomatic seed generation) (default: 0)
```

The stochastic simulation engine supports multi-threading.

```
--threads=<n> sets the number of threads to n.
```

***GSPN* support** MARCIE's simulative engine supports generalized stochastic Petri nets and thus stochastic and immediate transitions.

Computation of probability distributions The simulative engine can be used to compute instantaneous transient probabilities, cumulative transient probabilities, steady state probabilities and performability. Therefore one has to use the corresponding options given in 3.2 ff. and combine them with `--simulative` to trigger the simulative engine. In general the simulative engine computes a subset of all states with a probability above a certain value defined by the number of simulation runs.

Strength The simulative engine can be used to compute averaged traces of token values over time. Furthermore it is able to compute probability distributions with a subset of states having a probability above a certain value. All this can be done to **bounded** and **unbounded** \mathcal{GSPN} models.

Restrictions The accuracy of the results of the simulative engine depends on the number simulation runs and increases exponentially with the accuracy.

3.4 Approximative explicit Engine

MARCIE provides another numerical algorithm for the computation of transient probabilities. This approximative algorithm was suggested in [DHMW09] and is based on the idea to prune states with a probability below a specified threshold combined with adaptive uniformization. Thus, the algorithm can be applied to unbounded models as well. The algorithm is implemented in an explicit fashion, see [HRSS10] for experimental results.

`--approximative` switches the approximative engine on.

`--appr-delta=<real>` sets the threshold, considered when pruning states, to a specific probability value. The default is $1.0E - 11$.

`--appr-epsilon=<real>` sets the value allowed to drop the birth process. The default is $1.0E - 7$.

`--appr-steps=<number>` sets the number of steps. An automatic determination of the number of steps is triggered by 0. The default is 0.

`--appr-lambda=<real>` sets the assumed maximal exit rate of the stochastic process. The value must be specified by the user. See [HRSS10] for more details.

Computation of probability distribution The approximative engine can be used to compute the instantaneous transient probabilities of **bounded** and **unbounded** \mathcal{GSPN} models.

`--transient=<t>` computes the transient probabilities at time point t .

\mathcal{GSPN} support MARCIE's approximative numerical engine supports generalized stochastic Petri nets and thus stochastic and immediate transitions. But due to its direct CTMC computation, it has to consider both tangible and vanishing states. So, if possible, the \mathcal{GSPN} should be reduced to a \mathcal{SPN} as it is proposed in [MBC⁺95].

Strength The approximative engine can be used to compute the instantaneous transient probabilities of **bounded** and **unbounded** \mathcal{GSPN} models.

Restrictions Currently, the approximative engine does not support multi-threading.

4 Model checking

Based on the present engines MARCIE implements model checking algorithms for different temporal logics, which we will presented in the following.

4.1 CTL

4.1.1 Syntax

MARCIE contains a symbolic model checker for the Computation Tree Logic (CTL) [CGP01]. The CTL formulas must be specified in a separate text file. Each formula must be closed by a semicolon. It is allowed to define multiple formulas in one single file. Comments can be given in C/C++ style. The formulas must be conform to the following grammar:

```

ctl_input
= (constant)* (ctl_formula ';' )+ .

constant
= 'const' type CONST_NAME '=' const_function ';' .

type
= 'string' | 'int' | 'double' .

CONST_NAME
= a constant name which does not exist as placename .

ctl_formula
= ap
| '[' ctl_formula ']'
| '!' ctl_formula
| ctl_formula binop ctl_formula
| ae '[' ctl_forumla 'U' ctl_formula ']'
| untemp ctl_formula .

ae
= 'A' | 'E' .

```

```

untemp
= 'AX' | 'EX' | 'AF' | 'EF' | 'AG' | 'EG' .

```

```

binop
= '&' | '|' | '->' | '<-' | '<->' .

```

```

ap
= arithmetic_function cmp arithmetic_function
| const_function '<=' PLACE '<' const_function
| true
| false .

```

PLACE
= a valid place name, or the name of a constant which can be substituted by a place name.

```

cmp
= '=' | '!=' | '>=' | '>' | '<=' | '<' .

```

`--ctl-file=<filename>` reads the specified file and activates the model checker.

`--ctl-show-it=<n>` triggers the tool to display CTL statistics after n iterations.

As for the state space generation, one can choose between three generation algorithms using the option

```
--ctl-rs-algorithm=<1,2,3>
```

1. Breath First Search
2. Transition Chaining
3. Saturation (default)

When choosing the saturation algorithm, there are two further options which have the same semantics as for state space generation.

`--ctl-sat-rs-noown` do not use own transition ordering in the saturation-based reachability set generation (default: false).

`--ctl-sat-single` enable single firing in the saturation-based reachability set generation (default: false).

A CTL formula is TRUE if it holds for the initial state.

4.1.2 Patterns

In this section we will specify some often used properties by means of CTL using the FMS system with $N = 5$ as illustration.

Reachability It is possible to reach a state where place `P1s` carries 5 tokens.

```
EF [P1s = 5];
```

It is possible to reach a state where the rate of transition `tP3` exceeds some defined bound `bnd`.

```
EF [ rate(P3) > bnd];
```

Again, `rate` is a user-defined function template given in the model description.

Safety. It is not possible to reach a state where place `P1s` carries more than 5 tokens.

```
AG [! P1s > 5];
```

Liveness of a transition. Is the transition `tP3M2` live?

```
AG [EF [M2>0 * P3M2>0]];
```

Precedes. Does `P1M1` always precede `P1s`?

```
!E [ P1M1 =0 U P1s > 0];
```

4.2 CS(R)L

MARCIÉ offers model checking of the Continuous Stochastic Logic (CSL) [ASSB00] as defined in [BHHK03] and the Continuous Stochastic Reward Logic (CSRL) [BHHK00]. CSL is a proper CSRL subset. In our definition we include further the reward extensions introduced in [KNP07], although they do not represent genuine CS(R)L formulas.

4.2.1 Syntax

A CSRL formula must be loaded from a separate file by means of the option

```
--csl-file=<filename> specifies the file containing the CS(R)L formula. When using this option the CS(R)L model checking starts immediately after the state space generation.
```

Currently there is the restriction to have one formula per file. The quantitative analysis must be triggered by specifying a valid CSRL formula, which must be conform to the following grammar:

```
csrl_input
= (constant)* csl_top_formula .

constant
```

```

= 'const' type CONST_NAME const_function ';' .

type
= 'string'
| 'int'
| 'double' .

CONST_NAME
= a constant name which does not exist as place_name .

csrl_top_formua
= state_formula
| 'P=? [' csrl_path_formula ']'
| 'S=? [' state_formula ']'
| 'P{" REWARD "}=? [' csrl_path_formula ']' .
| 'R{" REWARD "}=? [' reward_formula ']' .

state_formula
= 'P' ep '[' csrl_path_formula ']'
| 'P{" reward_name "}' ep '[' csrl_path_formula ']'
| 'S' ep '[' state_formula ']'
| 'R{" reward_name "}' ep '[' reward_formula ']'
| '!' state_formula
| state_formula binop state_formula
| ap .

ep
= cmp real
| closed_interval .

reward_formula
= 'I=' real
| 'C<=' real
| 'S'
| 'F' state_formula .

csrl_path_formula
= '[' state_formula ']' 'U' closed_interval '[' state_formula']'
| 'F' closed_interval '[' state_formula ']'
| 'G' closed_interval '[' state_formula ']'
| 'X' closed_interval '[' state_formula ']' .

csrl_path_formula
= '[' state_formula ']' 'U' closed_interval closed_interval '[' state_formula ']'
| 'F' closed_interval closed_interval '[' state_formula ']'

```

```

| 'G' closed_interval closed_interval '[' state_formula ']'
| 'X' closed_interval closed_interval '[' state_formula ']' .

closed_interval
= empty
| '[' real ',' real ']'
| '[' real ',' 'oo' ']' .

binop
= '&' | '|' | '->' | '<-' | '<->' .

ap
= arithmetic_function cmp arithmetic_function
| const_function '<=' PLACE '<' const_function
| true
| false .

cmp
= '=' | '!=' | '>=' | '>' | '<=' | '<' .

real
= const_function .

```

PLACE is the name of an existing place.

REWARD is the name of an existing reward structure.

In the standard setting the model checker starts in the exact numerical mode (3.1) where all states will be considered. This requires a bounded (Generalized) Stochastic Petri net. If the state space is too huge or even infinite the user can try the simulative (3.3) or approximative (3.4) engine.

4.2.2 Patterns

In this section we will specify some often used queries by means of CS(R)L using the FMS system with $N = 5$ as illustration.

Transient analysis. What is the probability to have 5 tokens on place P_{s1} at time 1.

$$P=? [F[1,1] [P_{s1} = 5]]$$

Steady state analysis. What is the probability to have 5 tokens on place P_{s1} at the steady state.

$$S=? [P_{s1} = 5]$$

Reward-based analysis. Given the reward structures P_{s1} and $tM1^2$ and

```
rewards [ P1notEmpty ] {
```

²Implicitly generated by MARCIE.

```
P1 > 0 : 1;
}
```

we compute

- **Performability measures** as the distribution of the accumulated reward. We compute the probability that the system is at time point 2 in a state satisfying $P1s = 5$ and up to this moment at most one time unit is spent in states where $P1$ is not empty.

$$P\{\text{"P1notEmpty"}\}=? [F[2,2] [0,1] [P1s = 5]]$$

We compute the probability that the system is at time point 2 in a state satisfying $P1s = 5$ and the transition $tM1$ fires at most three times.

$$P\{\text{"tM1"}\}=? [F[2,2] [0,3] [P1s = 5]]$$

- **Expected instantaneous reward measures** as the expected average of tokens on place $P1$ in the steady state

$$R\{\text{"P1"}\}=? [S].$$

or at time 1.5

$$R\{\text{"P1"}\}=? [I=1.5].$$

- **Expected cumulative reward measures** as the expectation of how often transition $tM1$ may fire up to time 1.5.

$$R\{\text{"tM1"}\}=? [C<=1.5].$$

or the expected cumulative sojourn time in states where at least one token is on $P1s$ up to time 1.5.

$$R\{\text{"P1notEmpty"}\}=? [C<=1.5].$$

MARCIE offers CS(R)L model checking based on the three quantitative engines presented in Section (3.1). However, depending on the net class, the type of property and the used engine a couple of **constraints** have to be considered.

1. The **exact engine** supports **full CSRL** but is restricted to **rate rewards** and \mathcal{SPN} models. For a general discussion on CSRL model checking we refer to [Clo06]. To evaluate genuine CSRL formulas the exact engine deploys the Markovian approximation as discussed in [Sch14]. \mathcal{GSPN} models can be analysed by means of **full CSL**.
2. The **simulative engine** supports **CSRL** model checking of bounded and unbounded \mathcal{SPN} models, but is restricted to unnested formulae. Bounded and unbounded \mathcal{GSPN} models can be checked using **CSL**, but with the same restriction as for **CSRL**.
3. The **approximative engine** supports **CSL** model checking of bounded and unbounded \mathcal{GSPN} models, but only for unnested and time-bounded **CSL** formulae.

4.3 PLTLc

Based on the simulative engine (3.3) MARCIE enables model checking of linear-time properties. For the specification of these properties we define the probabilistic extension of the Linear-time Temporal Logic (LTL) [Pnu77b] with numerical constraints [FR07] which we will call Probabilistic Linear-time Temporal Logic with numerical constraints (PLTLc) [DG08].

4.3.1 Syntax

A PLTLc formula must be read from a separate file by means of the option

```
--ltl-file=<filename> to specify the file where the PLTLc
formula is given.
```

In PLTLc each temporal operator may be decorated with a time interval. In contrast to CSL the \mathcal{P} operator is allowed to appear only at the top level of the formula. The following grammar defines the set of all PLTLc formulas and highlights the differences between CSL and PLTLc.

```
pltl_input
= (constant)* pltl_top_formula .

constant
= 'const' type CONST_NAME const_function ';' .

type
= 'string'
| 'int'
| 'double' .

CONST_NAME
= a constant name which does not exist as place_name .

pltl_top_formua
= 'P=? [' pltl_formula filter ']'
| 'P' cmp real '[' pltl_formula filter ']' .

pltl_formula
= '!' pltl_formula
| pltl_formula binop pltl_formula
| '[' pltl_formula ']' 'U' closed_interval '[' pltl_formula ']'
| 'F' closed_interval '[' pltl_formula ']'
| 'G' closed_interval '[' pltl_formula ']'
| 'X' '[' pltl_formula ']'
| ap .
```

```

closed_interval
= empty
| '[' real ',' real ']'
| '[' real ',oo]' .

binop
= '&' | '|' | '->' | '<-' | '<->' .

ap
= term cmp term
| term cmp '$FREE_VARIABLE'
| true
| false .

filter
= empty
| '{' sp '}' .

sp
= '!' sp
| sp binop sp
| term cmp term
| true
| false

cmp
= '=' | '!=' | '>=' | '>' | '<=' | '<' .

real
= arithmetic_function .

term
= arithmetic_function
| 'd(' PLACE ')'
| 'd2(' PLACE ')'
| time

```

FREE_VARIABLE

= a valid identifier which is neither a name of a constant nor of place.

We define the functions d and $d2$ which operate on each places value individually to return the places first and second derivative at each state, thus increasing and decreasing places value can be checked;

$$d(Place) > 0 \text{ and } d(Place) < 0$$

respectively.

We also define the variable *time* which holds the state time. State time values are the simulation time points such that we can, for example, express properties which occur after some simulation time has elapsed. This is especially useful for expressing a property before or after some event.

Free variables. Free variables can be used to compute the probability distribution of the domain $\mathcal{D}_\phi \subset \mathbb{N}^n$ of the free variables so that ϕ becomes true.

4.3.2 Patterns

Reachability. What is the probability to reach a state where place **P1s** has 5 tokens?

$$P=?[F [P1s = 5]]$$

What threshold v place **P1s** attain in the trace?

$$P=?[F [P1s \geq \$v]]$$

Stability. What is the probability that after time point 10 place **P1s** has 3,4 or 5 tokens?

$$P=?[G [time > 10 \rightarrow [P1s \leq 5 \ \& \ P1s \geq 3]]]$$

Influence. What is the probability that place **P3s** grows above 3 while **P2s** does not exceed 3.

$$P=?[[P2s \leq 3 \ U \ P3s \geq 3]]$$

5 Templates

The given grammars for CTL, CSRL and PLTLc allow to formulate generic queries. We can use

- **string** constants to define placeholders for places or reward structures
- **integer** constants to define placeholders for token values
- **double** constants to define placeholders for time and reward bounds or probabilities.

Thus we can define generalizations of frequently used patterns.

5.1 CTL

Generalization of some CTL patterns (4.1.2).

```
const string p1;
const string p2;
const int n;

/*
Reachability
of states wiht n tokens on place p
*/
EF [ p = n ];

/*
Safety its never possible to have more
than n tokens on place p
AG [ ! p > n ];
*/

/*
If p2 is eventually marked, p1 is marked before
*/

!E [ ![ p1 > 0 ] U p2 > 0 ];
```

5.2 CSRL

Generalizations of some CSRL patterns (4.2.2).

Transient anaylsis.

```
const string p;
const double t;
const int k;

/*
probability to be in a state
with k tokens on place p at time t
*/

P=? [ F[t,t] [ p = k ] ]
```

Steadystate analysis.

```

const string p;
const int k;

/*
probability to be in a state
with k tokens on place p after reaching the steady state
*/

S=? [ p = k ],

```

Performability analysis.

```

const string p;
const string rs;
const double y;
const double t;
const int k;

/*
probability to be in a state
with k tokens on place p at time t
and accumulating at most a reward of y
*/

P{"rs"}=? [ F[t,t][0,y] [ p = k ] ]

```

Expected cumulative rewards

```

const string r;
const double t;

/*
cumulative reward up to time t considering reward structure r,
which could be for instance P1s or tM1
*/

R{"r"}=? [ C<= t ]

```

5.3 PLTLc

Generalization of some PLTLc patterns (4.3.2). **Stability.**

```
const double t;
const string p;
const int n1;
const int n2;

/*
What is the probability that after
time point t place p has between n1 and n2 tokens?
*/

P=? [ G [ TIME > t -> [p <= n2 & p >= n1 ] ] ]
```

Influence.

```
const string p1;
const string p2;
const int n;

/*
What is the probability that place p
grows above n while p2 does not exceed n.
*/

P=?[ [ p2 =< n U p1 >= n ] ]
```

6 Example Session

MARCIE is a pure command line tool and it is often annoying to type the commands all the time you run the tool, especially when repeating the same experiments. Therefore MARCIE offers the option

```
--config=<FILE> load a configuration file.
```

to run with a specified configuration. A configuration contains just the options, as you would type them into the terminal. But in the configuration file only one option per line is allowed. If used, the option must be the first option. A configuration can be extended by further options as far as no option is specified several times.

Example. It is reasonable to define in a configuration file the model, the formula and the constants for an experiments.

`experiments.cnf`:

```
--net-file=example.andl  
--const N=20  
--csl-file=example.csl
```

In the terminal we can specify further to use the simulative engine as follows:
`marcie --config=experiments.cnf --simulative`

In the following we demonstrate an analysis of a small biological model with MARCIE. For all experiments we provide also the related configuration files in the sub folder `examples/erk`.

6.1 Model description

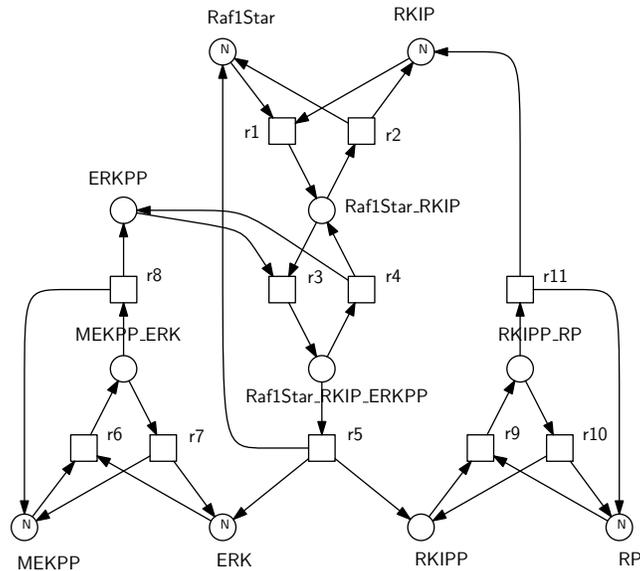


Figure 6: The RKIP inhibited ERK pathway.

As an example we use the RKIP inhibited ERK pathway (Fig. 6), published in [CSK⁺03], and discussed as qualitative and continuous Petri nets in [GH06], and as three related Petri net models in [GHL07]. Related files can be found here: <http://www-dssz.informatik.tu-cottbus.de/DSSZ/Examples/ERK>.

6.1.1 ANDL description

The given Petri net description has been created with our tool Snoopy [HHL⁺12a]. To begin we use Snoopy to export the net to the ANDL format(2.3).

```

sbn [ERK_N]
{
constants:
double c1 = 0.53;
double c2 = 0.0072;
double c3 = 0.625;
double c4 = 0.00245;
double c5 = 0.0315;
double c6 = 0.8;
double c7 = 0.0075;
double c8 = 0.071;
double c9 = 0.92;
double c10 = 0.00122;
double c11 = 0.87;

```

```

double N = 2;

places:
  Raf1Star = N;
  RKIP = N;
  Raf1Star_RKIP = 0;
  ERKPP = 0;
  MEKPP_ERK = 0;
  Raf1Star_RKIP_ERKPP = 0;
  RKIPP_RP = 0;
  MEKPP = N;
  ERK = N;
  RKIPP = 0;
  RP = N;

transitions:
  r1 : : [Raf1Star_RKIP + 1] & [Raf1Star - 1] & [RKIP - 1] : MassAction(c1);
  r2 : : [Raf1Star + 1] & [RKIP + 1] & [Raf1Star_RKIP - 1] : MassAction(c2);
  r3 : : [Raf1Star_RKIP_ERKPP + 1] & [Raf1Star_RKIP - 1] & [ERKPP - 1] :
MassAction(c3);
  r4 : : [Raf1Star_RKIP + 1] & [ERKPP + 1] & [Raf1Star_RKIP_ERKPP - 1] :
MassAction(c4);
  r6 : : [MEKPP_ERK + 1] & [MEKPP - 1] & [ERK - 1] : MassAction(c6);
  r7 : : [ERK + 1] & [MEKPP + 1] & [MEKPP_ERK - 1] : MassAction(c7);
  r9 : : [RKIPP_RP + 1] & [RP - 1] & [RKIPP - 1] : MassAction(c9);
  r10 : : [RP + 1] & [RKIPP + 1] & [RKIPP_RP - 1] : MassAction(c10);
  r5 : : [ERK + 1] & [RKIPP + 1] & [Raf1Star + 1] & [Raf1Star_RKIP_ERKPP - 1] :
MassAction(c5);
  r8 : : [ERKPP + 1] & [MEKPP + 1] & [MEKPP_ERK - 1] : MassAction(c8);
  r11 : : [RKIP + 1] & [RP + 1] & [RKIPP_RP - 1] : MassAction(c11);

}

```

6.2 Analysis

6.2.1 State space

We start with the computation of state space of the model with N set to 20. Further we check the model for reversibility and liveness using the provided options. Thus we call MARCIE as

```

marcie \
--net-file=erk_ma_N.andl \
--rev-check \
--live-net-check \
--const N=20

```

get

```

constant name N value 20;

(NrP: 11 NrTr: 11)

```

```

net check time: 0m0sec
place and transition orderings generation:0m0sec
init dd package: 0m0sec

RS generation: 0m0sec

-> reachability set: #nodes 211 (2.1e+02) #states 1,696,618 (6)

-> the net is reversible
check if the net is reversible: 0m0sec
-> the net is live
analysis based on terminal SCCs: 0m0sec

total processing time: 0m0sec

```

and can see that both properties are fulfilled by the model and the given initial marking.

6.2.2 CTL

In the next step we check whether it is possible to reach a state where on place p are n tokens. Thus we create the CTL template (5)

```

const integer n;
const string p;

EF [ p = n ];

```

and save it in the file `EF_p_eq_n.ct1`. We set N to 20 and n to 0 and define p as MEKPP. Calling MARCIE as

```

marcie \
--net-file=erk_ma_N.and1 \
--ctl-file=EF_p_eq_n.ct1 \
--const N=20,p=MEKPP,n=0

```

gives us

```
constant name N value 20;
constant name p value MEKPP;
constant name n value 0;

(NrP: 11 NrTr: 11)

net check time: 0m0sec

place and transition orderings generation:0m0sec

init dd package: 0m0sec

RS generation: 0m0sec

-> reachability set: #nodes 211 (2.1e+02) #states 1,696,618 (6)

starting CTL model checker
-----
checking: EF [p=0]
normalized: E [true U p=0]

abstracting: (MEKPP=0) states: 1,771 (3)
-> the formula is TRUE
mc time: 0m0sec

total processing time: 0m0sec
```

and we know that it is possible.

6.2.3 Printing the CTMC

At first we can have a look to the CTMC and the rewards in each state. We call MARCIE with the option `--print-ctmc` and assign to `N` the value 1. Further we load the reward structures defined in the file `erk.rew`. We call MARCIE as

```
marcie \
--net-file=erk_ma_N.andl \
--const N=1 \
--print-ctmc \
--list-rewards=MEKPP
```

and get

```
constant name N value 1;

(NrP: 11 NrTr: 11)

net check time: 0m0sec

place and transition orderings generation:0m0sec

init dd package: 0m0sec

RS generation: 0m0sec

-> reachability set: #nodes 21 (2.1e+01) #states 13

reward structures matching MEKPP:
rewards [MEKPP]{
true: MEKPP;
}

Starting Markov chain analysis
-----

Printing the CTMC
#States: 13
#State Transitions: 30

-----The states-----
Tangible states:
0 : (RKIPP_RP : 1 Raf1Star : 1 MEKPP_ERK : 1 )
1 : (RKIPP_RP : 1 Raf1Star : 1 ERK : 1 , MEKPP : 1 )
2 : (RKIPP_RP : 1 Raf1Star : 1 ERKPP : 1 MEKPP : 1 )
3 : (RP : 1 Raf1Star_RKIP_ERKPP : 1 MEKPP : 1 )
4 : (RP : 1 Raf1Star_RKIP : 1 MEKPP_ERK : 1 )
5 : (RP : 1 Raf1Star_RKIP : 1 ERK : 1 , MEKPP : 1 )
6 : (RP : 1 Raf1Star_RKIP : 1 ERKPP : 1 MEKPP : 1 )
7 : (RP : 1 RKIP : 1 , Raf1Star : 1 MEKPP_ERK : 1 )
8 : (RP : 1 RKIP : 1 , Raf1Star : 1 ERK : 1 , MEKPP : 1 )
9 : (RP : 1 RKIP : 1 , Raf1Star : 1 ERKPP : 1 MEKPP : 1 )
10 : (RP : 1 RKIPP : 1 Raf1Star : 1 MEKPP_ERK : 1 )
11 : (RP : 1 RKIPP : 1 Raf1Star : 1 ERK : 1 , MEKPP : 1 )
12 : (RP : 1 RKIPP : 1 Raf1Star : 1 ERKPP : 1 MEKPP : 1 )
```

```

Vanishing states:
allocate computation vector of size 0.01 KB
[0][1] = 0.007500 [0][2] = 0.071000 [0][7] = 0.870000 [0][10] = 0.001220
[1][0] = 0.800000 [1][8] = 0.870000 [1][11] = 0.001220
[2][9] = 0.870000 [2][12] = 0.001220
[3][6] = 0.002450 [3][11] = 0.031500
[4][5] = 0.007500 [4][6] = 0.071000 [4][7] = 0.007200
[5][4] = 0.800000 [5][8] = 0.007200
[6][3] = 0.625000 [6][9] = 0.007200
[7][4] = 0.530000 [7][8] = 0.007500 [7][9] = 0.071000
[8][5] = 0.530000 [8][7] = 0.800000
[9][6] = 0.530000
[10][0] = 0.920000 [10][11] = 0.007500 [10][12] = 0.071000
[11][1] = 0.920000 [11][10] = 0.800000
[12][2] = 0.920000
release vector of size 0.01 KB
-----The exit rates -----
allocate computation vector of size 0.10 KB
allocate computation vector of size 0.10 KB
allocate computation vector of size 0.01 KB
E:
Vector size: 13
@ [0]= 9.497200000000000E-01,
@ [1]= 1.671220000000000E+00,
@ [2]= 8.712200000000000E-01,
@ [3]= 3.395000000000000E-02,
@ [4]= 8.570000000000000E-02,
@ [5]= 8.072000000000000E-01,
@ [6]= 6.322000000000000E-01,
@ [7]= 6.084999999999999E-01,
@ [8]= 1.330000000000000E+00,
@ [9]= 5.300000000000000E-01,
@ [10]= 9.984999999999999E-01,
@ [11]= 1.720000000000000E+00,
@ [12]= 9.200000000000000E-01,
release vector of size 0.01 KB
release vector of size 0.10 KB
release vector of size 0.10 KB
allocate computation vector of size 0.10 KB
allocate computation vector of size 0.10 KB
-----The state rewards ( 23 reward structures) -----
reward structure " MEKPP":
#states: 13 #transitions: 1
reward function: MEKPP
Vector size: 13
@ [1]= 1.000000000000000E+00,
@ [2]= 1.000000000000000E+00,
@ [3]= 1.000000000000000E+00,
@ [5]= 1.000000000000000E+00,
@ [6]= 1.000000000000000E+00,
@ [8]= 1.000000000000000E+00,
@ [9]= 1.000000000000000E+00,
@ [11]= 1.000000000000000E+00,
@ [12]= 1.000000000000000E+00,
release vector of size 0.10 KB
release vector of size 0.10 KB

Total processing time: 0m1sec

```

6.2.4 Transient analysis

Now we compute the probability to eventually reach a state where we have n tokens on p . The following query is a stochastic adaption of the previously used CTL query.

We create the file `transient.csl` containing the CSL template. (5)

```
const integer n;
const string p;
const double t1;
const double t2;

P=? [F [t1,t2]p = n ]
```

The CTL operator `EF` has been replaced by the operator `P` and the operator `F` is now decorated with a time interval. We now call MARCIE as

```
marcie \
--net-file=erk/erk_ma_N.andl \
--csl-file=templates/transient.csl \
--const N=20,p=MEKPP,n=0,t1=1,t2=1 \
--threads=2
```

which triggers the computation of a distribution of transient probabilities at time 1. The computation is performed in multi-threaded mode on two CPU cores.

```
constant name N value 20;
constant name p value MEKPP;
constant name n value 0;
constant name t1 value 1;
constant name t2 value 1;

(NrP: 11 NrTr: 11)

net check time: 0m0sec

place and transition orderings generation:0m0sec

init dd package: 0m0sec

RS generation: 0m0sec

-> reachability set: #nodes 211 (2.1e+02) #states 1,696,618 (6)
```

```

Starting Markov chain analysis
-----

checking: P =? [ F [1,1] [p=0]]
normalized: P =? [ F [1,1] [p=0]]

abstracting: (MEKPP=0) states: 1,771 (3)
checking pathformula F [1,1] [MEKPP=0]
#absorbing states: 0/1,696,618 (6)
Computing exit rates ...
Starting numerical computation (simple multiplication, 2 Threads) ...
finished (2.207023 secs)

non zero states 1,771 (3)
zero states 1,694,847 (6)
maximum 688.000000 at position 1696407
split the interval with pattern:
time: 0/1

Uniformization:
lambda: 701.760000
time: 1.000000
left: 539
right: 928
-----
Starting numerical computation (Transient analysis, 2 Threads) ...
Iterations 928 ... finished (131.063297 secs)

Prob(m0) = 5.899615223309036E-02
number satisfying states: 1

-> the formula is TRUE
MC time: 2m13sec

Total processing time: 2m14sec

```

After nearly two minutes we get the result. MARCIE just prints the computed probability for the initial state, which is 5.899615223309033E-02. But we have computed the probability for the given formula for all reachable states. We could force MARCIE to output the complete result vector with the option `--print-results`.

For any unnested CSL formula we can alternatively use the simulative engine (3.3).

```

marcie \
--net-file=erk/erk_ma_N.andl \
--csl-file=templates/transient.csl \
--const N=20,p=MEKPP,n=0,t1=1,t2=1 \
--simulative \
--threads=2

```

We get



The implemented confidence interval method determines the number simulation runs. And we can see that the correct probability is between 0.05547475323 and 0.06016632706.

For this special case (time-bounded operators and a unnested CSL formula) we could also use the approximative engine (3.4).

```
marcie \
--net-file=erk/erk_ma_N.andl \
--csl-file=templates/transient.csl \
--const N=20,p=MEKPP,n=0,t1=1,t2=1 \
--approximative
```

We get

```
constant name N value 20;
constant name p value MEKPP;
constant name n value 0;
constant name t1 value 1;
constant name t2 value 1;

(NrP: 11 NrTr: 11)

place and transition orderings generation:0m0sec

checking: P =? [ F [1,1] [p=0]]
normalized: P =? [ F [1,1] [p=0]]

Options
threads: 1
runs: 1
start: 1.000000
end: 1.000000
split the interval with pattern:
output: 0
results:
delta: 1e-11
epsilon: 1e-07
max lambda: 0
using explicit fast adaptive uniformization engine
progress: 100%, elapsed time: 0m0sec
states: 8487
max states: 8561
avg enabled transitions: 7.47062259
lost probability: 7.58804871e-06

Prob(m0) = 5.899610241547879E-02
mc time: 0m0sec

Total processing time: 0m0sec
```

Although we can not use multi-threading the approximative engine is for the given settings the clear winner regarding the run-time. In general it is not

possible to predict which technique is the best choice. See [HRSS10] for a detailed discussion. The set of available options of the three implemented engines are described in detail here 3.1.

6.2.5 Steady state analysis

Now we compute the steady state probability to be in a state with no tokens on MEKPP. Therefore we create the file `steady.csl` containing the CSL template(5)

```
const integer n;
const string p;

S=? [p = n ]
```

We call MARCIE with

```
marcie \
--net-file=erk/erk_ma_N.andl \
--csl-file=templates/steady.csl \
--threads=2 \
--const N=20,p=MEKPP,n=0
```

to trigger the computation. We get

```
constant name N value 20;
constant name p value MEKPP;
constant name n value 0;

(NrP: 11 NrTr: 11)

net check time: 0m0sec

place and transition orderings generation:0m0sec

init dd package: 0m0sec

RS generation: 0m0sec

-> reachability set: #nodes 211 (2.1e+02) #states 1,696,618 (6)

Starting Markov chain analysis
-----
checking: S =? [p=0]
```

```

normalized: S =? [p=0]

abstracting: (MEKPP=0) states: 1,771 (3)

-> Found terminal SCC: #nodes 211 (2.1e+02) #states 1,696,618 (6)

#absorbing states: 0/1,696,618 (6)
Computing exit rates ...
Starting numerical computation (simple multiplication, 2 Threads) ...
finished (1.708038 secs)

Starting numerical computation (Jacobi method, 2 Threads) ...
Iterations 511 ...
finished (111.445278 secs)

Steady state probability: 1.809718286175109E-11

number satisfying states: 1

-> the formula is TRUE
MC time: 1m53sec

Total processing time: 1m54sec

```

and can see that the probability for the initial state is $1.809718286175109E-11$ (as for any other state since the CTMC is ergodic).

6.2.6 Reward analysis

We are interested in the most probable amount of tokens on MEKPP in the steady state. We can determine this amount using the already defined state rewards and the related reward operators (4.2.1).

Thus we define the CSL template

```

const string r;
R{"r"}=?[S]

```

Now we we specify the reward file and the name of the reward structure we want to use. Calling MARCIE as

```

marcie \
--net-file=erk/erk_ma_N.andl \
--csl-file=templates/RS.csl \
--threads=2 \
--const N=20,r=MEKPP

```

generates the output

```

constant name N value 20;
constant name r value MEKPP;

(NrP: 11 NrTr: 11)

net check time: 0m0sec

replacing :r --> MEKPPplace and transition orderings generation:0m0sec

init dd package: 0m0sec

RS generation: 0m0sec

-> reachability set: #nodes 211 (2.1e+02) #states 1,696,618 (6)

Starting Markov chain analysis
-----

checking: R{"MEKPP"}=? [S ]
normalized: R{"MEKPP"}=? [S ]

reward structure: 0x9a57ee8
States with non-zero reward : 1,694,847 (6)

-> Found terminal SCC: #nodes 211 (2.1e+02) #states 1,696,618 (6)

#absorbing states: 0/1,696,618 (6)
Computing exit rates ...
Starting numerical computation (simple multiplication, 2 Threads) ...
finished (1.719109 secs)

Starting numerical computation (Jacobi method, 2 Threads) ...
Iterations 511 ... finished (120.564538 secs)

yes 1,696,618 (6) no 0 maybe 0

Expected reward = 1.391160910143002E+01
number satisfying states: 1

-> the formula is TRUE
MC time: 2m3sec

Total processing time: 2m4sec

```

The expected instantaneous reward in the steady state is 1.391160910143015E+01.

6.2.7 PLTLc

PLTLc allows the P-operator to appear only at the top level of the formula. That's why and because PLTLc and CSL are syntactically comparable we are able to use such CSL formulas directly. At the moment model checking PLTLc is only supported by the simulative engine and therefore you can omit the parameter `--simulative`.

```
marcie \
--net-file=erk/erk_ma_N.andl \
--ltl-file=templates/transient.csl \
--const N=20,p=MEKPP,n=0,t1=1,t2=1
```

We get

```
constant name N value 20;
constant name p value MEKPP;
constant name n value 0;
constant name t1 value 1;
constant name t2 value 1;

(NrP: 11 NrTr: 11)

net check time: 0m0sec

place and transition orderings generation:0m0sec

function parsed:1
function parsed:1
simulative pltlc model checking
Options
runs: 65686
end: 1

checking: P =? [ F [1,1] [MEKPP = 0]]
normalized: P =? [ F [1,1] [MEKPP = 0]]

runs: 65686
result: P =? [ F [1,1] [MEKPP = 0]]

Expected probability(m0) = [2.8679112895e-02, 3.2128843779e-02]
with 99.000% confidence level

Size = 65686
Mean = 3.0356544774e-02
Variance = 2.9435024963e-02

mc time: 0m3sec

total processing time: 0m3sec
```

We see that the result is about the same as with the CSL model checker.

Now lets have a look at the “free variables” feature of PLTLc. Therefor we adapt the formula shown in section 4.3.2 and create the file `freevariable.ltl` containing the PLTLc template. (5)

```

const string p;
const double t1;
const double t2;

P=? [F [t1,t2]p >= $v ]

```

We call MARCIE with the following parameters

```

marcie \
--net-file=erk/erk_ma_N.andl \
--ltl-file=templates/freevariable.ltl \
--const N=20,p=MEKPP,t1=1,t2=1

```

and get this result.

```

constant name  N value 20;
constant name  p value MEKPP;
constant name  t1 value 1;
constant name  t2 value 1;

(NrP: 11  NrTr: 11)

net check time: 0m0sec

place and transition orderings generation:0m0sec

function parsed:1
function parsed:1
simulative pltlc model checking
Options
runs: 65686
end: 1

checking: P =? [ F [1,1] [v == 0] <= MEKPP]
normalized: P =? [ F [1,1] [v == 0] <= MEKPP]

runs: 65686
result: P =? [ F [1,1] [v == 0] <= MEKPP]

Expected probability(m0) = [1.0000000000e+00, 1.0000000000e+00]
with 99.000% confidence level
Size = 65686
Mean = 1.0000000000e+00
Variance = 0.0000000000e+00

v:
[0, 1]{3.1711475809e-02} [0, 2]{2.3767621715e-01} [0, 3]{3.9257984959e-01}
[0, 4]{2.4505983010e-01} [0, 5]{7.6287184484e-02} [0, 6]{1.4477970953e-02}
[0, 7]{1.9943366928e-03} [0, 8]{1.9791127485e-04} [0, 10]{1.5223944219e-05}

mc time: 0m3sec

total processing time: 0m4sec

```

It shows us the domain of the free variable v and the probability of each interval.

References

- [ASSB00] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Model checking continuous-time Markov chains. *ACM Trans. on Computational Logic*, 1(1), 2000.
- [BHHK00] Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. On the logical characterisation of performability properties. In *Automata, Languages and Programming*, volume LNCS 1853 of *Lecture Notes in Computer Science*, pages 780–792. Springer-Verlag, 2000.
- [BHHK03] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time markov chains. *IEEE Trans. on Software Engineering*, 29(6), 2003.
- [CBC⁺93] G. Ciardo, A. Blakemore, P. F. Chimento, J. K. Muppala, and K. S. Trivedi. Automated generation and analysis of markov reward models using stochastic reward nets. *IMA Volumes in Mathematics and its Applications: Linear Algebra, Markov Chains, and Queueing Models / Meyer, C.; Plemmons, R.J.*, 48:145–191, 1993.
- [CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.
- [CGP01] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. MIT Press 1999, third printing, 2001.
- [Clo06] L. Cloth. *Model Checking Algorithms for Markov Reward Models*. PhD thesis, University of Twente, 2006.
- [CLP04] Yang Cao, Hong Li, and Linda Petzold. Efficient formulation of the stochastic simulation algorithm for chemically reacting systems. *J Chem Phys*, 121(9):4059–4067, Sep 2004.
- [CSK⁺03] K.-H. Cho, S.-Y. Shin, H.-W. Kim, O. Wolkenhauer, B. McFerran, and W. Kolch. Mathematical modeling of the influence of RKIP on the ERK signaling pathway. In *CMSB 2003*, pages 127–141. LNCS 2602, Springer, 2003.
- [CT93] G. Ciardo and K. S. Trivedi. A Decomposition Approach for Stochastic Reward Net Models. *Performance Evaluation*, 18(1):37–59, 1993.

- [DG08] R. Donaldson and D. Gilbert. A Monte Carlo model checker for probabilistic LTL with numerical constraints. Technical report, University of Glasgow, Dep. of CS, 2008.
- [DHMW09] F. Didier, T. A. Henzinger, M. Mateescu, and V. Wolf. Fast Adaptive Uniformization for the Chemical Master Equation. In *HiBi*, 2009.
- [FR07] F. Fages and A. Rizk. On the analysis of numerical data time series in temporal logic. In *Proc. CMSB 2007*, pages 48–63. LNCS/LNBI 4695, Springer, 2007.
- [GB00] M. A. Gibson and J. Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry, A* 104:1876–1889, 2000.
- [GH06] D. Gilbert and M. Heiner. From Petri nets to differential equations - an integrative approach for biochemical network analysis. In *Proc. ICATPN 2006*, pages 181–200. LNCS 4024, Springer, 2006.
- [GHL07] D. Gilbert, M. Heiner, and S. Lehrack. A unifying framework for modelling and analysing biochemical pathways using Petri nets. In *Proc. CMSB 2007*, pages 200–216. LNCS/LNBI 4695, Springer, 2007.
- [Gil77] D.T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.
- [Gil01] Daniel T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *The Journal of Chemical Physics*, 115(4):1716–1733, 2001.
- [HGD08] M. Heiner, D. Gilbert, and R. Donaldson. Petri nets in systems and synthetic biology. In *SFM*, pages 215–264. LNCS 5016, Springer, 2008.
- [HHL⁺12a] M Heiner, M Herajy, F Liu, C Rohr, and M Schwarick. Snoopy – a unifying Petri net tool. In *Proc. Application and Theory of Petri Nets*, volume 7347 of *LNCS*, pages 398–407. Springer, 2012.
- [HHL⁺12b] M Heiner, M Herajy, F Liu, C Rohr, and M Schwarick. Snoopy – a unifying Petri net tool. In *Proc. PETRI NETS 2012*, volume 7347 of *LNCS*, page 398–407. Springer, June 2012.
- [HLGM09] M. Heiner, S. Lehrack, D. Gilbert, and W. Marwan. Extended Stochastic Petri Nets for Model-Based Design of Wetlab Experiments. pages 138–163. LNCS/LNBI 5750, Springer, 2009.

- [HRSR08] M. Heiner, R. Richter, M. Schwarick, and C. Rohr. Snoopy-a tool to design and execute graph-based formalisms. *Petri Net Newsletter*, 74:8–22, 2008.
- [HRSS10] M. Heiner, C. Rohr, M. Schwarick, and S. Streif. A comparative study of stochastic analysis techniques. In *Proc. CMSB 2010*, pages 96–106. ACM, 2010.
- [HST09] M. Heiner, M. Schwarick, and A. Tovchigrechko. DSSZ-MC - A Tool for Symbolic Analysis of Extended Petri Nets. In *Proc. Petri Nets 2009*, pages 323–332. LNCS 5606, Springer, 2009.
- [KNP07] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM’07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007.
- [MBC⁺95] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing, John Wiley and Sons, 1995. 2nd Edition.
- [MN98] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.
- [MS11] Sean Mauch and Mark Stalzer. Efficient Formulations for Exact Stochastic Simulation of Chemical Systems. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 8:27–35, January 2011.
- [Noa99] A. Noack. A ZBDD Package for Efficient Model Checking of Petri Nets (in German). Technical report, BTU Cottbus, Dep. of CS, 1999.
- [Par02] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.
- [Pnu77a] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.
- [Pnu77b] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, 1977.
- [RMH10] C. Rohr, W. Marwan, and M. Heiner. Snoopy - a unifying Petri net framework to investigate biomolecular networks. *Bioinformatics*, 26(7):974–975, 2010.

- [Sch10] M. Schwarick. IDD-MC - a model checker for bounded Stochastic Petri nets. In *Proc. 17th German Workshop on Algorithms and Tools for Petri Nets (AWPN 2010)*, volume 643 of *CEUR Workshop Proceedings*, pages 80–87. CEUR-WS.org, September 2010.
- [Sch14] M. Schwarick. *Symbolic On-the-fly analysis of stochastic Petri nets*. PhD thesis, BTU Cottbus, Dep. of CS, 2014.
- [SH09] M. Schwarick and M. Heiner. CSL model checking of biochemical networks with interval decision diagrams. In *Proc. CMSB 2009*, pages 296–312. LNCS/LNBI 5688, Springer, 2009.
- [SM08] W. Sandmann and C. Maier. On the statistical accuracy of stochastic simulation algorithms implemented in Dizzy. In *Proc. WCSB 2008*, pages 153–156, 2008.
- [Spr01] J. Spranger. *Symbolic LTL Verification of Petri Nets (in German)*. PhD thesis, Brandenburgische Technische Universität Cottbus, 2001.
- [ST10] M. Schwarick and A. Tovchigrechko. IDD-based model validation of biochemical networks. *Theoretical Computer Science*, 2010.
- [Ste94] W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton Univ. Press, 1994.
- [Tov08] A. Tovchigrechko. *Model Checking Using Interval Decision Diagrams*. PhD thesis, BTU Cottbus, Dep. of CS, 2008.

Index

--Dead-states=<filename>, 23
 --Dead-trace=<filename>, 23
 --Live-trans=<filename>, 23
 --Max-vals=<FILE>, 22
 --Not-live-trans=<filename>, 23
 --Porder-write=<filename>, 22
 --Torder-write=<filename>, 21
 --appr-delta=<real>, 33
 --appr-epsilon=<real>, 33
 --appr-lambda=<real>, 33
 --appr-steps=<number>, 33
 --approximative, 33
 --config=<FILE>, 46
 --const <c1=v1,c2=v2,...,cn=vn>, 12
 --csl-file=<filename>, 36
 --ctl-file=<filename>, 35
 --ctl-rs-algorithm=<1,2,3>, 35
 --ctl-sat-rs-noown, 35
 --ctl-sat-single, 35
 --ctl-show-it=<n>, 35
 --cumulative, 26
 --dead-check, 23
 --export-dist-evolution=<FILE>, 28
 --export-exp-evolution=<FILE>, 28
 --export-results=<File>, 27
 --export-results=<filename>, 32
 --export-state-dist-gnuplot, 29
 --lin-solver=<1,2,3>, 25
 --list-rewards, 25
 --list-rewards=<r1:r2:...:rn>, 19
 --live-net-check, 23
 --ltl-file=<filename>, 40
 --max-val, 22
 --memory-dd=<1,2,3,4,5>, 20
 --net-file=<filename>, 9
 --normalize, 29
 --observe=<r1:r2:...:rN>, 27
 --performability=<rs>:<t>,<y>, 26
 --place-order=<1,2,3,4,5,6,7,8,9>, 21
 --porder-read=<filename>, 22
 --precision-results=<n>, 27
 --print-Reward-vector=<rs1>,...,<rsN>, 25
 --print-ctmc, 25
 --print-results, 27
 --print-size, 25
 --print-states, 25
 --rev-check, 22
 --reward-file=<filename>, 18
 --rs-algorithm=<1,2,3>, 20
 --sat-rs-noown, 20
 --sat-single, 20
 --seed=<n>, 32
 --sim-accuracy=<a>, 31
 --sim-algorithm=<1,2,3,4>, 31
 --sim-confidence=<c>, 31
 --sim-export-selection=<regex>, 32
 --sim-input-file=<filename>, 32
 --sim-input-separator=<s>, 32
 --sim-runs=<n>, 31
 --sim-start=<t₀>, 31
 --sim-stop=<t>, 31
 --simulative, 30
 --single-file, 28
 --split-interval=<s1:t1,s2:t2,...,sn:tn>, 29, 32
 --steady-state, 26
 --steps=<number>, 26
 --suppress-implicit-rewards, 18
 --threads=<n>, 24, 32
 --torder-read=<filename>, 21
 --trans-order=<1,2,3,4,5,6,7,8>, 21
 --transient=<t>, 26, 33
 --truncation-layer=<1,...,|P|−1>, 23
 --vector-format=<CSV,Compressed,Latex>, 27