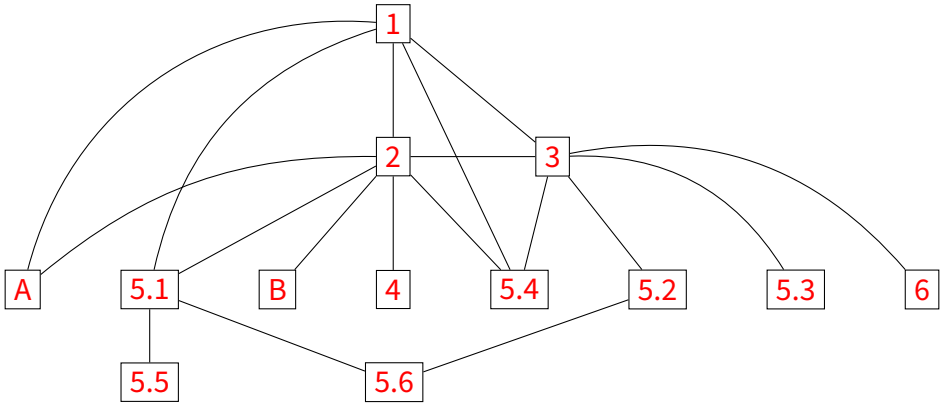
Maude strategy-aware, external, and quantitative model checkers

User manual

March 25, 2025

Maude [7] lets users specify concurrent and complex systems by means of terms, equations and rules, the language of rewriting logic [18]. These models can be directly executed or simulated by the interpreter, or analyzed by various tools. One of such tools is a built-in model checker [12] for linear temporal logic (LTL) properties. Since rewriting is just the application of rules one after the other, in any order and anywhere, a strategy language [10, 8, 11, 21, 19] was proposed and incorporated to the interpreter in order to gain some global control on the process. It is designed as a new specification layer on top of equations and rules, so that the same model can exhibit different behaviors when controlled by alternative strategies. These composed systems can be model checked with the strategy-aware model checker described in this document, which supports LTL properties as an extension of the Maude LTL model checker [22, 23], and CTL, CTL* and μ-calculus using external tools [27, 26]. These tools can also be applied to classical models. Moreover, by assigning probabilities on top of the Maude specifications, probabilistic properties can be checked and quantitative results calculated by probabilistic and statistical methods [24].

This manual starts with short explanations on how standard, strategy-aware, and probabilistic models are specified. The procedure for strategy-controlled systems does not differ much from the way it is done in the standard model checker described in [7, §12]. The extended LTL model checker can be used within Maude and a unified utility is provided to uniformly access the model checkers for all the supported logics. It also provides resources to better analyze its results, and it is described in Section 5.5. Users only interested in branching-time model checking can jump to Section 5. Those only interested in quantitative verification can jump to Sections 3, 5.2, 5.3 and 6. The Maude version including the model checkers, along with the unified model-checking utility, related documentation and examples can be found at maude.ucm.es/strategies.



Contents

1 Standard Maude models **3**
1.1 Problem preparation . . . 3

2 Strategy-controlled models **6**
2.1 Finite traces . . . 7
2.2 Parallel subterm rewriting . . . 8
2.3 Opaque strategies . . . 8
2.4 Model preparation . . . 8

3 Probabilistic models **9**

4 The strategy-aware LTL model checker within Maude **13**
4.1 Understanding the model checker output . . . 14
4.2 Running the model checker at the metalevel . . . 16

5 The unified Maude model-checking utility **18**
5.1 Standard model checking with check . . . 19
5.2 Probabilistic model checking with pcheck . . . 21
5.3 Statistical model checking with scheck . . . 23
5.4 Graph generation with graph . . . 26
5.5 A graphical interface . . . 27
5.6 External model checkers and their installation . . . 28

6 The statistical simulator for MultiVeStA **29**

A The Maude language extension for LTSmin **31**

B The Kleene-star semantics of the iteration **32**

1 Standard Maude models

Models in model checking are formalized as annotated state and transition systems known as Kripke structures. Rewriting systems can be naturally viewed as Kripke structures by identifying terms with states and adding a transition from one state to another if the first can be rewritten to the second by a rule. Like this, the executions of the model are sequences of rule applications. However, since temporal properties are usually only defined on infinite executions, the one-step rewrite relation should be completed by adding self-loops to all *deadlock* states, where no transition leaves. This is how the standard Maude LTL model checker works [7, §12].

Let us introduce an example for explaining, in the following sections, how Maude specifications are prepared for model checking. The classical problem of the *dining philosophers* is specified in the following modules: a group of *n* philosophers is gathered around a table to have dinner, for what they have to take the forks at both their sides. However, the table is round and there are only *n* forks, so they cannot eat all at the same time.

```
fmod PHILOSOPHERS-TABLE is      *** functional module
  protecting NAT .

  sorts Obj Phil Being List Table .
  subsorts Obj Phil < Being < List .

  op (_|_|_) : Obj Nat Obj -> Phil [ctor] .
  ops o ψ : -> Obj [ctor] .
  op empty : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: empty] .
  op <_> : List -> Table [ctor] .

  var L : List .
  ceq < ψ L > = < L ψ > if L =/= empty .
  op initial : -> Table .
  eq initial = < (o | 0 | o) ψ (o | 1 | o) ψ (o | 2 | o) ψ > .
endfm

mod PHILOSOPHERS-DINNER is      *** system module
  protecting PHILOSOPHERS-TABLE .
  var Id : Nat .
  var X : Obj .
  var L : List .
  rl [left] : ψ (o | Id | X) => (ψ | Id | X) .
  rl [right] : (X | Id | o) ψ => (X | Id | ψ) .
  rl [left] : < (o | Id | X) L ψ > => < (ψ | Id | X) L > .
  rl [release] : (ψ | Id | ψ) => ψ (o | Id | o) ψ .
endm
```

The philosophers can try three different moves: taking their `left` fork, their `right` fork, or `release` both of them at once. Doing it at their sole discretion may lead to some unwanted situations, like the starvation of some of them, or worse, of all of them. Strategies can prevent some of these problems by imposing additional restrictions, as explained in Section 2.

1.1 Problem preparation

In this section, we summarize the procedure for preparing any Maude specification for model checking. The following ingredients should be supplied:

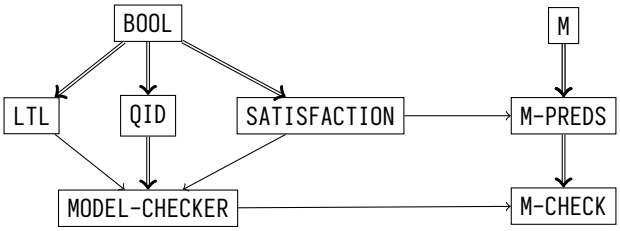


Figure 1: Structure of the model checker modules

- A model specified by a system module `M` with a designated sort of states.
- A set of atomic propositions and a satisfaction relation $\models$ that specifies which are satisfied in each state.
- A temporal formula built on top of the previous atomic propositions.
- An initial term.

For instance, in the dining philosophers example, the rewriting model is given by the system module `PHILOSOPHERS-DINNER`, whose sort `Table` should be designated as the state sort. The initial term can be the `initial` symbol, where all the forks are on the table.

The procedure involves some predefined and user-defined modules respectively depicted in the left and right sides of Figure 1. For the predefined modules, the `model-checker.maude` file in the Maude standard distribution should be loaded. The state sort is selected together with the declaration of the atomic propositions and their satisfiability. This is usually done in a new system module, say `M-PREDS`, that includes `M` and the predefined module `SATISFACTION`, where the `State` and `Prop` sorts, and the satisfaction relation symbol `_|=_` are declared.

```
fmod SATISFACTION is
  protecting BOOL .
  sorts State Prop .
  op _|=_ : State Prop -> Bool [frozen] .
endfm
```

The sort of states is selected by making it a subsort of `State`. `M-PREDS` must be a protected extension of `M` to ensure that the model is not altered in any way (although this is not checked by Maude). Finally, a system module `M-CHECK` merges the specification of the model and properties in `M-PREDS` with the predefined `MODEL-CHECKER` module. This module gives access to the `modelCheck` symbol and transitively imports the `LTL` module where the syntax for this temporal logic is defined.

```
*** primitive LTL operators
ops True False : -> Formula [...] .
op ~_ : Formula -> Formula [prec 53 ...] .
op _/\_ : Formula Formula -> Formula [comm prec 55 ...] .
op _\/_ : Formula Formula -> Formula [comm prec 59 ...] .
op O_ : Formula -> Formula [prec 53 ...] .
op _U_ : Formula Formula -> Formula [prec 63 ...] .
op _R_ : Formula Formula -> Formula [prec 63 ...] .

*** defined LTL operators
op _->_ : Formula Formula -> Formula [prec 65 ...] .
op _<->_ : Formula Formula -> Formula [prec 65 ...] .
op <>_ : Formula -> Formula [prec 53 ...] .
op []_ : Formula -> Formula [prec 53 ...] .
```

```
op _W_ : Formula Formula -> Formula [prec 63 ...] .
op _|->_ : Formula Formula -> Formula [prec 63 ...] . *** leads-to
op _=>_ : Formula Formula -> Formula [prec 65 ...] .
op _<=>_ : Formula Formula -> Formula [prec 65 ...] .
```

An optional LTL-SIMPLIFIER module can be included to simplify the LTL formula. In practice, there is no need to follow this module structure and M, M-PREDS, and M-CHECK can be written as a single module, but this is the usual convention.

Coming back to the dining philosophers problem, M would be PHILOSOPHERS-DINNER and the following PHIOSOPHERS-PREDS module can be its M-PREDS:

```
mod PHILOSOPHERS-PREDS is
  protecting PHILOSOPHERS-DINNER .
  including SATISFACTION .

  subsort Table < State .
  op eats : Nat -> Prop [ctor] .

  var  Id : Nat .
  vars L M : List .

  eq < L (ψ | Id | ψ) M > |= eats(Id) = true .
  eq < L > |= eats(Id) = false [owise] .
endm
```

A single parametric proposition eats is defined, all whose ground instances are the atomic propositions in the formal sense. The satisfaction relation is equationally defined to make eats(n) hold in any state where the n-th philosopher has both forks, and so is able to eat. Finally, the module collecting all the specification components is defined, playing the role of M-CHECK.

```
mod PHILOSOPHERS-CHECK is
  protecting PHILOSOPHERS-PREDS .        *** atomic propositions
  protecting PHILOSOPHERS-FORMULAE .
  including MODEL-CHECKER .
endm
```

In this module, we will be able to use the model checker as explained in Sections 4 and 5.1. We have imported another module, PHILOSOPHERS-FORMULAE, that has not been explained yet. Since LTL formulae are represented by terms in Maude, the user can construct and manipulate formulae within the language. For example, we can declare two formulae someoneEats and allEat, generic on the number of philosophers, to express that at least a philosopher or every philosopher can eat.

```
mod PHILOSOPHERS-FORMULAE is
  protecting PHILOSOPHERS-PREDS .
  protecting LTL .

  *** Parameterized formulae for a given number of philosophers
  ops someoneEats allEat : Nat -> Formula .

  var L : List .
  var X Y : Obj .
  var Id N : Nat .

  eq someoneEats(0) = False .
  eq someoneEats(s(N)) = someoneEats(N) \/ eats(N) .
```

```
  eq allEat(0) = True .
  eq allEat(s(N)) = allEat(N) /\ <> eats(N) .
endm
```

According to the equations, `someoneEats`(n) is the LTL formula $\bigvee_{k=0}^{n-1}$ `eats`(k) and `allEat`(n) is $\bigwedge_{k=0}^{n-1}$ $\Diamond$ `eats`(k).

Summing up, the script for preparing a problem to be model checked is:

1. Specify the model in a system module `M`.
2. In a protecting extension of `M` including the predefined `SATISFACTION` module (say `M-PREDS`) choose the sort of the model states by making it a subsort of the predefined `State` sort. Declare as many atomic propositions as desired as operators of range `Prop`, and define the satisfaction relation `|=` for all of them.
3. Write a system module (say `M-CHECK`) combining the specification of the model and properties in `M-PREDS` with the predefined `MODEL-CHECKER` module. Optionally, the `LTL-SIMPLIFIER` module may be included for LTL simplification.

2 Strategy-controlled models

The evolution of strategy-controlled rewriting systems does not only depend on the rules but also on the strategies that limit their application. States in those systems are not univocally associated to terms and their transitions are those rule rewrites allowed by the strategy. We also consider some special options for specific situations, but before describing them we will illustrate the normal behavior with an example.

Remember that unwanted situations may appear in the dining philosophers problem introduced in Section 1. Strategies can prevent some of them by imposing additional restrictions.

```
smod PHILOSOPHERS-PARITY is
  protecting PHILOSOPHERS-DINNER .

  strat parity @ Table .
  sd parity := (release
    *** The even take the left fork first
    | (amatchrew L s.t. ψ (o | Id | o) := L /\ 2 divides Id
             by L using left)
    | left[Id <- 0]
    *** The odd take the right fork first
    | (amatchrew L s.t. (o | Id | o) ψ := L /\ not (2 divides Id)
        by L using right)
    *** When they already have one, they take the other fork
    | (amatchrew L s.t. (ψ | Id | o) ψ := L by L using right)
    | (matchrew M s.t. < L (o | Id | ψ) L' > := M
            by M using left[Id <- Id])
    ) ? parity : idle .
endsm
```

With the `parity` strategy, even philosophers are compelled to take the left fork before the right one, and the odd should do the opposite. These additional rewriting restrictions produce a different model in which different properties are satisfied. For example, thanks to the `parity` strategy, the situation in which no philosopher can eat is avoided.

Notice that the strategy above is recursive and non-terminating. Even though non-terminating executions cannot be observed with the `srewrite` commands [7, §10.4], non-terminating strategies

are meaningful and useful to specify the behavior of non-stopping and reactive systems, which are the typical targets of model checking. Non-terminating rewriting paths are not an obstacle for the decidability of model checking as long as they repeat a finite number of states; in other words, as long as they are caused by a loop. The model checker is able to detect cycles also when strategies have parameters.

```
smod PHILOSOPHERS-TURNS is
  protecting PHILOSOPHERS-DINNER .

  strat turns @ Table .
  strat turns : Nat Nat @ Table .

  sd turns := matchrew M s.t. < L (o | Id | o) ψ > := M
                  by M using turns(0, s(Id)) .
  sd turns(K, N) := left[Id <- K] ; right[Id <- K] ;
                  release ; turns(s(K) rem N, N) .
endsm
```

The `turns` strategy in the module above makes the philosophers eat in turns. With 3 philosophers, the strategies `turns(0,3)`, `turns(1,3)`, `turns(2,3)` follow continuously in a loop. This is not a convenient solution to the philosophers problem due to the absence of parallelism, but it ensures that all of them eat infinitely often.

Strategies always select a subset of the model executions. Hence, all linear temporal properties satisfied by the unrestricted model will be satisfied by the model controlled by no matter which strategy. However, the model checker also allows considering some explicitly selected strategies as atomic steps, as described hereafter, breaking this rule.

2.1 Finite traces

Strategies are commonly used to specify finite rewriting sequences, those whose results are observed using the `srewrite` commands. Finite executions do not exactly fit in the model-checking setting, where properties are defined for infinite traces. However, they can be assimilated to infinite traces extending their last state forever. Intuitively, the modeled system has stopped after completing its execution and so it will continue in its idle state in perpetuity. Coming back to the philosophers example and considering the recursive strategy `sd free := all ? free : idle .` that coincides with the free execution of the rewriting rules, some finite traces are found:

```
                    < (o | 0 | o) ψ (o | 1 | o) ψ (o | 2 | o) ψ >
→right[Id <- 0]     < (o | 0 | ψ)   (o | 1 | o) ψ (o | 2 | o) ψ >
→right[Id <- 1]     < (o | 0 | ψ)   (o | 1 | ψ)   (o | 2 | o) ψ >
→right[Id <- 2]     < (o | 0 | ψ)   (o | 1 | ψ)   (o | 2 | ψ)   >
```

This is a finite trace because the strategy `all`, the application of any rule, cannot be executed in the last state. Then, the conditional operator will execute is negative `idle` branch and terminate. The philosophers, unable to do any other movement, will remain like this indefinitely.

The interpretation of finite traces is equivalent to that of terminating or *deadlocked* rewriting sequences in the standard model checker. In fact, the finite execution above is a deadlocked one, but this is not true in general. Suppose `free` were defined `all ; free` instead. Then, the trace above would be discarded since the strategy commits to execute `all` again in the fourth state, which is impossible. Conversely, the strategy `all*` admits all finite rewriting sequences, extended by repeating the last state forever, even if they can be continued otherwise with a genuine rule transition.

2.2 Parallel subterm rewriting

The semantics of a subterm rewriting combinator like the following (or its variants `xmatchrew` and `amatchrew`),

$$\texttt{matchrew } P(X_1,\ldots,X_n) \texttt{ by } X_1 \texttt{ using } \alpha_1, \ldots, X_n \texttt{ using } \alpha_n$$

is the *parallel* rewriting of its n subterms against their corresponding strategies [28]. When looking to the state evolution as a sequence of rewriting steps, this means that the next step may come from any of the subterms. In other words, the execution traces of the `matchrew` are all possible interleavings of all possible combinations of the execution traces of its subterms.

In general, generating all these combinations is computationally expensive. So, an optional and alternative form of *partial order reduction* is offered, which analyzes only those traces where the subterms are rewritten in order, i.e. the rewrites within the subterm k always occur before the rewrites within the subterm $k + 1$. It is the user responsibility to ensure by other means that this is enough to prove the correctness of the system.

2.3 Opaque strategies

The model transitions are fundamentally rule applications, both for the strategy-aware and the standard model checker, but sometimes having strategies as transitions is useful and convenient. We call these strategies *opaque* since the intermediate states occurring during its execution are invisible. Instead, a single transition is seen from the term where the strategy is applied to each of its results. An example of model checking with opaque strategies can be seen in Section 4.1.

2.4 Model preparation

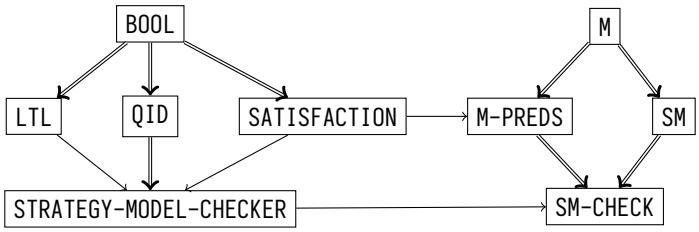


Figure 2: Structure of the strategy model checker modules

The problem setting and the usage of the strategy-aware model checker does not differ much of the standard procedures described in Section 1.1. The only difference is that, as part of the model specification, a strategy must be provided. The model is now specified by

- a Maude system module `M`, and
- a named strategy, defined in a strategy module `SM` that controls `M`. If using the interfaces of Sections 4.2, 5 and 6, an arbitrary strategy expression can be used instead of a named strategy, and the strategy module may be omitted for simple ones.

For instance, in the dining-philosophers examples, the base rewriting model `M` is the system module `PHILOSOPHERS-PREDS`, and the controlling strategy can be either `parity` in `PHILOSOPHERS-PARITY` or `turns` in `PHILOSOPHERS-TURNS`.

As illustrated in Figure 2, the collection of predefined and user-defined modules for strategy-aware model checking is very similar to that for standard model checking in Figure 1. In the typical setting, the rule-based model is specified in a system module `M`, one or more strategies controlling `M` are defined in an extension of this module that we will call `SM`, its atomic propostions and state are specified in a protected extension of `M` called `M-PREDS` that includes the predefined `SATISFACTION` module,

and everything is combined in a strategy module SM-CHECK. This latter module imports the predefined STRATEGY-MODEL-CHECKER module to allow access to the strategy-aware model checker. The standard MODEL-CHECKER module is compatible with the strategy-aware version and both can be imported and used at the same time. The model setup does not change when model checking using external tools in Section 5.1.

As a final summary, the script for preparing a problem to be model checked with strategies is:

1. Specify the model in a system module M, and define one or more named strategy strategies without parameters to control M in a strategy module SM. In the interfaces of Sections 4.2 and 5, an arbitrary strategy expression can be used instead of a named strategy without parameters.
2. In a protecting extension of M, say M-PREDS, choose the sort of the model states by making it a subsort of the State sort declared in SATISFACTION. Declare as many atomic propositions as desired as operators of range Prop, and define the satisfaction relation |= for all of them.
3. Declare a strategy module, say SM-CHECK, to combine the model M with the property specification in M-PREDS and the strategies SM. Import the STRATEGY-MODEL-CHECKER module too, and optionally the LTL-SIMPLIFIER module for LTL simplification.

3 Probabilistic models

Maude specifications, both standard and strategy-controlled, are intrinsically nondeterministic. However, this nondeterminism can be quantified to yield probabilistic systems that can be analyzed by probabilistic and statistical model-checking methods (see Sections 5.2 and 6). The tools described in this manual offer different alternatives to specify probabilities on top of rewriting systems specified in Maude. In the simplest case, uniform probabilities on the successors are considered for every state. In the most expressive one, probabilities are assigned with arbitrary complex programs in a probabilistic extension of the Maude strategy language. These *probability assignment methods* will be described in this section and illustrated with the following example of a simple coin.

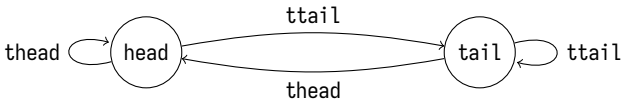
```
mod COIN is
  sort Coin .
  ops head tail : -> Coin [ctor] .

  vars C C' L R : Coin .

  rl [thead] : C => head [metadata "8"] .
  rl [ttail] : C => tail [metadata "5"] .

  op inertia : Coin Coin -> Nat .
  eq inertia(C, C') = if C == C' then 2 else 1 fi .
endm
```

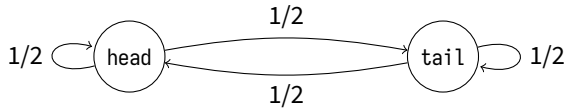
Coins can be either in head or tail state, and they can change from one to the other by the rules thead and ttail. Other aspects of this module will be explained in due time. The standard rewrite graph of this model is this:



The probability assignment methods are uniform, metadata, term, uaction, mdp-uniform, mdp-metadata, mdp-term, and strategy for the extension of the strategy language. Some of them derive discrete-time Markov chains (DTMC) out of the rewriting model and others produce Markov decision processes

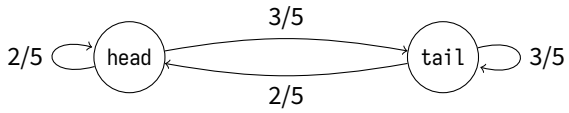
(MDP), which combine nondeterministic and probabilistic behavior. Continuous-time Markov chains (CTMC) can also be derived with the methods `ctmc-uniform`, `ctmc-metadata`, `ctmc-term`, `ctmc-uaction`, and `ctmc-strategy`. All methods except `strategy` are *local*, in the sense that they distribute the probability among the successors of every state separately. Most use weights to quantify the likeliness of the successors that are later normalized to obtain probabilities.

- `uniform` assigns the same probability to every successor of a term, i.e. successors are chosen uniformly at random. For example, the `COIN` module becomes the following DTMC under uniform probabilities:



- `uaction(`a_1`=`w_1`, …, `a_n`=`w_n`)` receives an assignment of weights to rule labels or actions of the rewriting system. First, the probability is distributed among the labels according to their weights. Then, for each rule label, the successors get an equal share of the assigned probability, in other words, they are then chosen uniformly at random. Instead of weights, actions can be assigned fixed probabilities with a_k`.p=`p_k instead of a_k`=`w_k. Fixed probabilities and weights for different actions can be combined, but fixed probabilities should never sum more than 1. In case no specification is given for an action, a unitary weight is assumed.

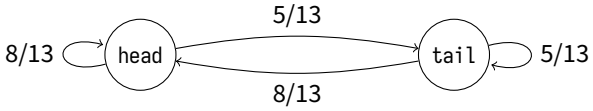
  On top of the `COIN` example, the method `uaction(ttail=3, thead=2)` produces the following discrete-time Markov chain:



  However, since there is a single successor for each action, the uniform distribution of probabilities per action is trivial in this case. The same result is obtained using a fixed probability with the `uaction(ttail.p=0.4)` method.

- `metadata` distributes the probability among the successors according to the weights written in the free-text `metadata` attributes of the rules that caused their transitions. The content of the `metadata` attribute must be a numeric literal or a Maude term of sort `Nat` or `Float` depending on the variables of the rule. A weight of 1 is assumed in case the attribute is missing. Whenever a successor has been generated by multiple rule applications, the weight of one of them is chosen in an implementation-defined way.

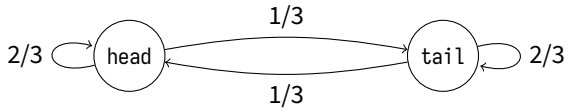
  In the rules of the `COIN` module, we can see an example of this kind of specification, where `thead` is assigned a weight of 8 and `ttail` is given a weight of 5. The DTMC produced by this method is the following:



- `term(`t`)` is given a Maude term t to calculate weights for every transition on a state and distribute the probabilities according to them. This term must reduce to a literal of numerical sort (`Nat` or `Float`) and it may contain the variables `L`, `R`, and `A` to be instantiated respectively with the left- and right-hand side of the transition, and the label of the rule that caused it, as a term of

sort `Qid` with `'unknown` for unlabeled rules. Whenever differently labeled rules produce the same transition, one label is chosen in an implementation-defined way.

In the `COIN` module, we have already defined a function `inertia` that gives twice as much weight to the current face as to the other one. The equivalent methods `term(inertia(L, R))` and `term(if L == R then 2 else 1 fi)` produce the following probabilistic model:



Since the variables `L` and `R` have already been declared in the module, we can directly write them in the term. Otherwise, we should have written `L:Coin` and `R:Coin`.

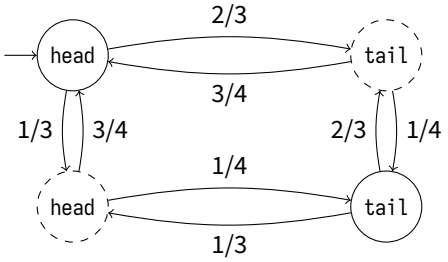
- `strategy` uses a probabilistic extension of the strategy language to both control and assign probabilities to the base rewriting system. This method produces a discrete-time Markov chain, a Markov decision process, or even an error depending on how much unquantified nondeterminism is left by the strategy. The new probabilistic combinators of the strategy language are:
  - `choice(`w_1 `:` α_1`, ...,` w_n `:` α_n`)` that selects one of the strategies α_k according to their weights w_k. These weights are terms in the `Nat` or `Float` sorts that may contain variables if they are defined in the outer scope. This is an evolution of the nondeterministic choice operator $\alpha_1 \mid \cdots \mid \alpha_n$, and similar constructs exist in a probabilistic extension of ELAN [3] and in Porgy [13].
  - `sample` X `:=` π`(`t_1`, ...,` t_n`)` `in` α that samples the variable X from a probabilistic distribution with parameters $t_1, \ldots, t_n$ that may contain variables defined in the outer contexts. The new variable X can be freely used in α. Both the variable X and the parameters must be of sort `Float`. The available distributions are `bernouilli(`p`)`, `uniform(`a`, `b`)`, `exp(`λ`)`, `norm(`μ`, `σ`)`, and `gamma(`α`, `β`)`.
  - An extension of the `matchrew`, `xmatchrew`, and `amatchrew` combinators of the standard strategy language to allow specifying the weight of every match and select one according to these weights. Syntactically, an optional infix `with weight` w is added to the original operators, like in

    `matchrew` $P(X_1,\ldots,X_n)$ `s.t.` C `with weight` w `by` X_1 `using` α_1`, ...,` X_n `using` α_n`,`

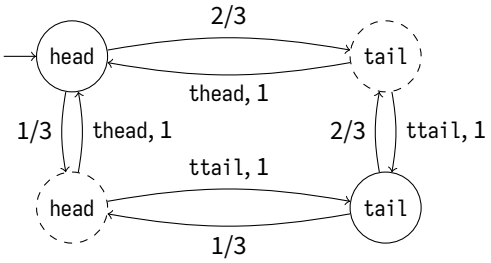
    where the weight w is a term of sort `Nat` or `Float` that may contain variables from the matching, the condition, and the outer scope.

The `sample` operator is not intended for discrete probabilistic model generation, since it samples continuous probabilistic distributions, but it can be used for statistical model checking and simulation.

For example, the strategy `(choice(2 : ttail, 1 : thead) ; choice(1 : ttail, 3 : thead))*` specifies the following discrete-time Markov chain if the initial term is `head`:



Notice that the probabilities are now non-local and the graph is made more complex since this strategy has memory. Probabilistic and nondeterministic behavior can be mixed in `(choice(2 : ttail, 1 : thead) ; (ttail | thead)) *` yielding a Markov decision process:



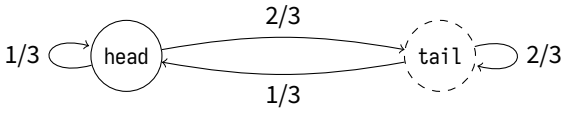
When the local methods are applied in a strategy-controlled model, the procedure does not change except that it is applied on the rewrite graph controlled by the strategy. The `mdp-` variants of the local assignment methods produce a Markov decision process where probabilities are only assigned once the rule label is nondeterministically chosen. There is no `mdp-uaction` method since it would not make any sense. These assignment methods are used for probabilistic model checking in Section 5.2 and for statistical model checking in Section 6.

For the `ctmc-` variants of the methods, weights are interpreted as firing rates of the corresponding transitions instead of unnormalized probabilities. In the `ctmc-uniform` method, the firing rate of every transition is unitary. In the case of `ctmc-uaction`, each transition is given the rate specified for its label, but weights are normalized if a fixed probability `.p` is given. For the `ctmc-strategy` method, the strategy must be free of unquantified nondeterminism, i.e. `strategy` should be able to generate a DTMC.

In addition to the previous general methods, there are some more for statistical model checking only. They can be used with the builtin statistical model checker in Section 5.3 and the simulator for MultiVeSta in Section 6.

- `step` considers the given probabilistic strategy as the atomic step of the model, i.e., the $(n+1)^{\text{th}}$ state of an execution is a solution of the strategy applied on the n^{th} state. For a sound statistical analysis, this strategy should not contain unquantified nondeterminism. In other words, it should provide a single (random) solution in every execution.

  For example, the `step` method with the strategy `choice(2 : ttail, 1 : thead)` yields the following discrete-time Markov chain, even though it will not be explicitly constructed.



  Sample operators can also be used.

- `pmaude` assigns probabilities to an APMaude model according to the conventions of that framework [1]. In particular, the Maude specification must define a function `tick` on the state sort and a function `getTime` from the state sort to `Float`. Every execution $(t_n)_{n=0}^{\infty}$ starts by rewriting the initial term with the rules of the module to obtain t_0, and then t_{n+1} is obtained by rewriting `tick`(t_n) exhaustively with the rules of the module. Randomness relies on the PMaude infrastructure and the `random` and `counter` operators in the Maude prelude, and random seeds are refreshed on each execution. If a strategy expression is provided, it will be ignored.

4 The strategy-aware LTL model checker within Maude

Once in a module that gathers all the required information, like SM-CHECK in Section 2.4, the model checker can be invoked by the modelCheck operator introduced in STRATEGY-MODEL-CHECKER. Its first two parameters are shared with the standard model checker: they are the initial term and the LTL formula. The third one is a quoted identifier with the name of the desired strategy, which can be chosen among the strategies without arguments defined in the current module.

```
op modelCheck   : State Formula Qid QidList Bool
                -> ModelCheckResult [special(...)] .
```

The last two parameters are optional, because the modelCheck symbol is overloaded to give them default values. The fourth parameter is a list of opaque strategy names. All strategies named here, regardless of their signature, will be considered opaque for the model checker as described in Section 2.3. The fifth parameter is a Boolean flag to activate the partial order reduction for the parallel matchrew described in Section 2.2. Their default values are the empty list and true respectively.

Once model checking is completed, the modelCheck operator reduces to true if the formula holds for the given model, or a counterexample of sort ModelCheckResult in case the model violates the property. A detailed description of the results is outlined in the Section 4.1, but first we will test the model checker in the known philosophers problem.

```
Maude> red modelCheck(initial,
        [] (<> eats(0) \/ <> eats(1) \/ <> eats(2)), 'parity) .
rewrites: 170 in 32ms cpu (50ms real) (5187 rewrites/second)
result Bool: true
```

This property says that at any moment one of the philosophers will eventually eat, i.e. that there are no deadlocks. The reader can check with the standard model checker (or with another strategy) that the property does not hold in the unrestricted model. However, not all desired properties are satisfied using this strategy.

```
Maude> red modelCheck(initial,
        [] (<> eats(0) /\ <> eats(1) /\ <> eats(2)), 'parity) .
rewrites: 139 in 22ms cpu (23ms real) (6054 rewrites/second)
result ModelCheckResult: counterexample(nil,
        {< (o | 0 | o) ψ (o | 1 | o) ψ (o | 2 | o) ψ >, 'left}
        {< (o | 0 | o) ψ (o | 1 | o) (ψ | 2 | o) ψ >, 'right}
        {< (o | 0 | o) ψ (o | 1 | o) (ψ | 2 | ψ) >, 'release})
```

Alternatively to the modelCheck operator, the command-line and graphical interface described in Section 5.5, and the meta-level entry point explained in Section 4.2 can be used.

Model checking is decidable when the model is finite and the transition relation is decidable itself. Both conditions may fail due to the base rewriting system or on account of the strategy. Strategies may be used to explore a finite portion of an infinite base model, but they may also make a finite system infinite using non-terminating recursive functions. The decidability requirements, adapted from those listed in Section 12.3 of the Maude manual, are:

1. The set of states that are reachable from the initial term following the strategy is finite.

   States refers to composed states, including both the term and the strategy execution progress. Clearly, if such composed states are finitely many, the base system terms are finite too. The converse depends on the concrete strategy, but different sufficient conditions are given in [28].

2. The rewrite theory $R = (\Sigma, E, \phi, R)$ specified by M plus the equations D defining the predicates Π are such that:

- both E and $E \cup D$ are (ground) Church-Rosser and terminating, modulo axioms if any, with $(\Sigma, E) \subseteq (\Sigma \cup \Pi, E \cup D)$ a protecting extension, and
- R is (ground) coherent with regard to E, module axioms if any.

4.1 Understanding the model checker output

The model checker output is a term of sort `ModelCheckResult`. When the model checker concludes that the property holds, the term reduces to the constant `true` of sort `Bool`. Otherwise, the property has been refuted and a counterexample trace is returned as witness, described by a loop and a path to it from the initial state in the rewrite graph. Like in the standard model checker, this is presented as a binary operator `counterexample(path, loop)`. Each of its components is a list of pairs `{state, trans}` whose first entry is the state term and whose second one is the description of a transition that connects it with the next state. Transition descriptions, of sort `RuleName`, are extended respect to the standard model checker according to what was told in Section 1:

- Rule applications are the most common transitions, and they are represented by its rule label as a quoted identifier, except in case of unlabeled rules that are signaled by the constant `unlabeled`.
- Solution transitions appear to signal the repetition of the last state in finite traces. They are written as `solution` and always occur at the end of the cycle. This is similar to the `deadlock` transition of the standard model checker, which is not used here.
- Opaque strategies are represented by a term `opaque(name)` where `name` is the name of the strategy as a quoted identifier.

In the second example execution of the previous section, we observe a cycle-only counterexample in which only the second philosopher is able to eat, because it repeatedly takes both forks and releases them in a loop. The transitions are the rules `left`, `right` and `release`.

To see other kinds of transitions, let us introduce another example:

```
mod RIVER is
  sort River Side Group .
  subsort Side < Group .

  op  _|_ : Group Group -> River [ctor comm] .
  ops left right : -> Side [ctor] .
  ops shepherd wolf goat cabbage : -> Group [ctor] .
  ops __ : Group Group -> Group [ctor assoc comm prec 40] .

  op initial : -> River .
  eq initial = left shepherd wolf goat cabbage | right .

  vars G G' : Group .

  rl [wolf-eats] : wolf goat G | shepherd G' => wolf G | shepherd G' .
  rl [goat-eats] : cabbage goat G | shepherd G' => goat G | shepherd G' .

  rl [alone]   : shepherd G | G' => G | shepherd G' .
  rl [wolf]    : shepherd wolf G | G' => G | shepherd wolf G' .
  rl [goat]    : shepherd goat G | G' => G | shepherd goat G' .
  rl [cabbage] : shepherd cabbage G | G' => G | shepherd cabbage G'  .
endm
```

The `RIVER-CROSSING` module specifies the well-known *river crossing* puzzle: a shepherd must conduct a wolf, a goat, and a cabbage to the other side of a river using a boat in which only two passengers fit.

The risks are that, unless the presence of the shepherd hinders it, the wolf would eat the goat or the goat would eat the cabbage. These animals will attack its meal as soon as they are left alone with it, so the `wolf-eats` and `goat-eats` rules should be applied with higher priority than the crossing ones. This is enforced using a strategy:

```
smod RIVER-CROSSING-STRAT is
  protecting RIVER-CROSSING .

  strats eating oneCrossing cross&eat eagerEating @ River .

  sd eating := wolf-eats | goat-eats .
  sd oneCrossing := alone | wolf | goat | cabbage .
  sd cross&eat := oneCrossing ; eating ! .

  sd eagerEating := match left | right shepherd wolf goat cabbage
                      ? idle : (cross&eat ; eagerEating) .
endsm
```

In the `RIVER-CROSSING-STRAT` strategy module, four strategies are defined: `eating` applies the eating rules until it cannot be applied further, `oneCrossing` runs any of the crossing rules, and `cross&eat` makes a single crossing followed by all necessary eating. Finally, the `eagerEating` strategy iterates `cross&eat` indefinitely. `River` is selected as the model-checking state and some atomic propositions are declared in the following module:

```
mod RIVER-CROSSING-PREDS is
  protecting RIVER-CROSSING .
  including SATISFACTION .

  subsort River < State .
  ops bad goal : -> Prop [ctor] .

  vars G G' : Group . var  R : River .

  eq left | right shepherd wolf goat cabbage |= goal = true .
  eq R |= goal = false [owise] .

  eq G cabbage | G' goat |= death = false .
  eq G cabbage goat | G' |= death = false .
  eq R |= death = true [owise] .
endm
```

A state is bad if an animal is left alone with its *prey*. These states may appear during the river crossing, but they must not be passed by without eating. A state is a `goal` if all the characters are on the right side of the river. We can use the model checker to find a solution to the puzzle by checking the property $\Box \neg goal$. First, we gather all the components in the module

```
smod RIVER-CROSSING-SCHECK is
  protecting RIVER-CROSSING-STRATS .
  protecting RIVER-CROSSING-PREDS .
  including STRATEGY-MODEL-CHECKER .
endsm
```

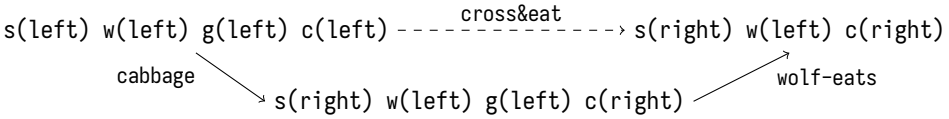
Then, we reduce the `modelCheck` operator:

```
Maude> red modelCheck(initial, [] ~ goal, 'eagerEating) .
rewrites: 204 in 33ms cpu (36ms real) (6170 rewrites/second)
```

```
result ModelCheckResult: counterexample(
        {left shepherd wolf goat cabbage | right,'goat}
        {left wolf cabbage | right shepherd goat,'alone}
        {left shepherd wolf cabbage | right goat,'wolf}
        {left cabbage | right shepherd wolf goat,'goat}
        {left shepherd goat cabbage | right wolf,'cabbage}
        {left goat | right shepherd wolf cabbage,'alone}
        {left shepherd goat | right wolf cabbage,'goat},
        {left | right shepherd wolf goat cabbage,solution})
```

The counterexample is a finite trace, whose cycle is a self-loop by a `solution` transition. This is a solution of the problem, and we are sure it is since no bad states are visited because eating always happens before moving again. This precedence can also be achieved with equations, but probably at the expense of coherence. Moreover, it is an optimal solution in length, although this may have not happened.

Because of the `eagerEating` strategy, we do not pass by bad states, but they are still visited and seen as a proper state by the model checker. Hence, the property $\Box \neg \mathit{bad}$ does not hold. If we make opaque the `cross&eat` strategy, by passing its name as the fourth argument of `modelCheck`, the transitions like



will be seen as a single action and the bad states will be hidden. Now $\Box \neg \mathit{bad}$ does hold,

```
Maude> red modelCheck(initial, [] ~ bad, 'eagerEating, 'cross&eat) .
rewrites: 108 in 0ms cpu (2ms real) (~ rewrites/second)
result Bool: true
```

and `cross&eat` transitions are printed instead of the particular rules that the strategy applies.

```
Maude> red modelCheck(initial, <> goal, 'eagerEating, 'cross&eat) .
rewrites: 40 in 0ms cpu (0ms real) (~ rewrites/second)
result ModelCheckResult: counterexample(
        {left shepherd wolf goat cabbage | right, opaque('cross&eat)}
        {left wolf cabbage | right shepherd goat, opaque('cross&eat)}
        {left shepherd wolf cabbage | right goat, opaque('cross&eat)}
        {left cabbage | right shepherd wolf goat, opaque('cross&eat)}
        {left shepherd goat cabbage | right wolf, opaque('cross&eat)},
        {left goat | right shepherd wolf cabbage, opaque('cross&eat)}
        {left shepherd goat | right wolf cabbage, opaque('cross&eat)})
```

The `ModelCheckResult` lacks information about the strategy execution, like the next strategy to be executed from each state, that is available via the umaudemc utility described in Section 5.

4.2 Running the model checker at the metalevel

To use the `modelCheck` operator, a named strategy without parameters must be defined in a strategy module. This may not be comfortable when model checking against a strategy with parameters, which may need to be adjusted. An alternative model-checking operator receiving a strategy expression at the metalevel is proposed in the `META-MODEL-CHECKER` module.

```
op metaModelCheck : Module Term Term Strategy QidList Bool ~> Term .
```

This operator does the same as `modelCheck`, but its arguments are the meta-representations of the module where to model check, the initial term, the LTL formula, and the controlling strategy. Only the list of opaque strategies and the biased-`matchrew` flag are given at the object level. Its result is the meta-representation of the `ModelCheckResult` produced by the model checker.

This function is indeed defined in Maude itself by adding the parameter strategy in an extension of the given module, and then reducing the usual model-checking operator.

```
op addStrategy : Module Qid Strategy -> Module .
op downResult  : ResultPair -> ModelCheckResult .
op upQidList   : QidList -> Term .

eq addStrategy(M, Q, S) = smod getName(M) is
     including 'STRATEGY-MODEL-CHECKER .  getImports(M)
     sorts getSorts(M) .
     getSubsorts(M) getOps(M) getMbs(M) getEqs(M) getRls(M)
     (strat Q : nil @ 'State [none] . getStrats(M))
     (sd Q[[empty]] := S [none] . getStratDefs(M))
   endsm .

eq metaModelCheck(M, T, F, S, L, B) = getTerm(metaReduce(
     addStrategy(M, '%metaModelCheck, S),
     'modelCheck[T, F, ''%metaModelCheck.Qid, upQidList(L),
                 if B then 'true.Bool else 'false.Bool fi]))
```

The function `upQidList` is also defined in the module to raise the list of opaque strategies to the meta-level[1]. Unlike in the object-level model checker, the `META-MODEL-CHECKER` module does not need to be included in an extension of the module under review. For instance, resuming the dining philosophers problem, we can execute

```
Maude> red in META-MODEL-CHECKER : metaModelCheck(
    upModule('DINNER-MCS, false),
    'initial['s_^7['0.Zero]],
    '`[`]_['allEat['s_^7['0.Zero]]],
    'turns[['0.Zero, 's_^7['0.Zero]]],
    nil, true) .
rewrites: 2092 in 29ms cpu (29ms real) (70041 rewrites/second)
result Constant: 'true.Bool
```

where `allEat`(n) is a function that produces the LTL formula $\Diamond\, eats(0) \wedge \cdots \wedge \Diamond\, eats(n-1)$. If we try the same command with `'parity[[empty]]` as its fourth argument, we obtain the metarepresentation of a counterexample term.

```
Maude> red in META-MODEL-CHECKER : metaModelCheck(
    upModule('DINNER-MCS, false),
    'initial['s_^7['0.Zero]],
    '`[`]_['allEat['s_^7['0.Zero]]],
    'parity[[empty]],
    nil, true) .
rewrites: 164 in 3ms cpu (2ms real) (49190 rewrites/second)
result GroundTerm: 'counterexample[
'`{_`,_`}['<_>['__['_|_|_['o.Obj,'0.Zero,'o.Obj],'ψ.Obj,...]],''left.Qid],
'__['`{_`,_`}['<_>['__['_|_|_['ψ.Obj,'0.Zero,'o.Obj], ...],''right.Qid],
     ...,
```

[1] The predefined `upTerm` cannot be used in this case because `META-LEVEL` defines subsorts of `Qid` that may not be defined in the target module.

```
'`{_`,_`}['<_['__['_|_|_['ψ.Obj,'0.Zero,'o.Obj], ...]],
          ''release.Qid]]]
```

5 The unified Maude model-checking utility

The unified Maude model-checking utility umaudemc is a uniform command-line, graphical, and programming interface to different model checkers operating on Maude specifications. On both standard and strategy-controlled Maude specifications, umaudemc can be used for

- checking LTL, CTL, CTL*, and μ-calculus properties (see Section 5.1),
- applying probabilistic model-checking methods (see Section 5.2), and
- exporting the rewrite graphs in different formats (see Section 5.4).

This program can be obtained alternatively from

- the strategy-aware model checker packages at maude.ucm.es/strategies,
- its source code repository at github.com/fadoss/umaudemc,
- or installed from the Python Package Index (PyPI) with `pip install umaudemc`.

The umaudemc tool requires Python 3.7 or newer to work and the maude package [20], which can also be installed with `pip install maude` (this is done automatically when installing `umaudemc` with `pip`). Moreover, some alternative external backends are required for checking branching-time (LTSmin [15], pyModelChecking [5], NuSMV [6], or Spot [9]) and probabilistic (PRISM [16] or Storm [14]) properties. Instructions for installing these backends are available in Section 5.6.

The command-line interface is organized in subcommands: `check` (Section 5.1), `pcheck` (Section 5.2), `graph` (Section 5.4), and `gui` (Section 5.5). Their syntax and options are listed by passing the `--help` flag. Every command except `gui` starts with the following arguments

`umaudemc` ⟨*subcommand*⟩ ⟨*Maude filename*⟩ ⟨*initial term*⟩

Some more options allow a more precise selection of the input Maude model:

`--module` ⟨*name*⟩ Selects the module specifying the system to be model checked. By default, as in the Maude interpreter, the last module will be used unless a module is explicitly selected with a `select` command in the file.

`--metamodule` ⟨*term*⟩ Selects the meta-module described by its argument as the module where to model check. The term will be reduced in the module indicated by the `module` option or selected by default.

`--opaque` ⟨*list*⟩ Indicates a comma-separated list of strategies to be considered opaque, as described in Section 2.3 (for strategy-controlled models).

`--full-matchrew` Enables the generation of the full set of traces for the rewriting of subterms operators, as described in Section 2.2 (for strategy-controlled models).

Moreover, the following arguments may be inserted between `umaudemc` and the subcommand:

`-v` Select the verbose mode so that more information is printed by the tool.

`--no-advise` Suppress debug advisories from Maude, like the `-no-advise` option of the Maude interpreter.

5.1 Standard model checking with check

The check command checks LTL, CTL, CTL*, and μ-calculus properties on standard and strategy-controlled Maude specifications. Several alternative backends can be used to check the satisfaction of the given formula, but the problem data is introduced and the verification result is shown in the same format for all of them. The basic syntax of the command is:

umaudemc check ⟨*filename*⟩ ⟨*initial state*⟩ ⟨*formula*⟩ [⟨*strategy*⟩]

The strategy argument can be any well-defined strategy expression, but it can also be omitted if model checking without strategies. Formulae are written in the language of the LTL module (see Section 1.1) extended with path quantifiers for CTL* and μ-calculus operators.

```
sorts @MCVariable@ @Action@ @ActionSpec@ @ActionList@ .
subsort @MCVariable@ < Formula .

*** CTL and CTL*
op A_ : Formula -> Formula [prec 53 ...] .
op E_ : Formula -> Formula [prec 53 ...] .

*** mu-calculus
op <.>_  : Formula -> Formula [prec 53 ...] .
op [.]_  : Formula -> Formula [prec 53 ...] .
op <_>_  : @ActionSpec@ Formula -> Formula [prec 53 ...] .
op [_]_  : @ActionSpec@ Formula -> Formula [prec 53 ...] .
op mu_._ : @MCVariable@ Formula -> Formula [prec 64 ...] .
op nu_._ : @MCVariable@ Formula -> Formula [prec 64 ...] .

subsorts @Action@ < @ActionList@ < @ActionSpec@ .
op opaque : @Action@ -> @ActionList@ [ctor] .
op ~_ : @ActionList@ -> @ActionSpec@ [ctor prec 50] .
op __ : @ActionList@ @ActionList@ -> @ActionList@ [ctor assoc] .
```

A and E are the universal and existential path quantifiers on CTL and CTL* formulae. The language of μ-calculus includes the existential modalities <.> and <_>, the universal modalities [.] and [_], and the fixed-point operators mu and nu, as well as all other propositional-logic operators defined in LTL. Modalities taking an @ActionSpec@ as argument must include a space-separated list of rule labels, which can be prefixed by the ~ symbol to indicate its complement. For opaque strategies (see Section 2.3), their names should be specified with the opaque constructor. The steps described by the modality are those labeled with one of these symbols, as usual.

$$\langle l_1 \cdots l_n \rangle \varphi \coloneqq \bigvee_{i=1}^{n} \langle l_k \rangle \varphi \qquad [l_1 \cdots l_n] \varphi \coloneqq \bigwedge_{i=1}^{n} [l_k] \varphi$$

The dot version of the same operators considers all possible transitions, as if the whole list of rule labels was written. Fixed-point operators are followed by a variable name that should be used in the nested subformula. Any identifier can be a variable as long as it does not conflict with other syntactical elements of the formula. The program will parse the given formula in this grammar,[2] and deduce the logic in which they are expressed and all required information to call the appropriate model checker. Formulae mixing μ-calculus and CTL* operators are not valid.

In addition to these positional arguments, other optional parameters can be set up that are listed when invoking the program with the --help flag. When branching-time properties are checked, the

[2] The sorts @Action@ and @MCVariable@ of the grammar are dynamically populated with the labels of the selected module and the candidate variables in the formula.

rewriting graph generated for LTL model checking must be applied some adaptations [27]. This utility will automatically choose the appropriate ones according to the input problem data, but the user may still overwrite the default settings. These options are meaningless without strategies, and so will be ignored in that case.

--merge-states ⟨*option*⟩ Merges successor states with a common term but different strategy continuations, if the option is `state` or `edge`. Moreover, with `edge` only successors by a common transition label are merged. Merging can be disabled completely with `no` and umaudemc can be instructed to set the appropriate configuration with `default` (`edge` for μ-calculus with edge labels, `no` for LTL, and `state` for the rest).

--purge-fails ⟨*option*⟩ Enables (`on`) or disables (`off`) the elimination of failed states. The `default` option is `off` for LTL and on-the-fly model checking algorithms, and `on` otherwise.

Other options are used to control the format of counterexamples. The `--slabel` option is specially useful to simplify counterexamples involving complex terms.

--show-strat Shows the next strategy to be executed from each state in the counterexample.

-c Prefers backends that provides counterexamples to those that do not provide them.

--slabel ⟨*format*⟩ Sets the format of state labels to a string that may contain special variables: `%t` for the current term, `%s` for the immediate strategy continuation, and `%i` for the internal state index. Moreover, arbitrary Maude terms containing `%t` can be written between curly brackets to be evaluated and replaced by their results. For example, an atomic proposition can be checked with `{%t |= aprop}`).

--elabel ⟨*format*⟩ Sets the format of edge labels to a string that may contain special variables: `%s` for the transition statement (rule, strategy declaration…), `%l` for its label, `%n` for its line number, and `%o` for `opaque` if the transition is caused by an opaque strategy.

--format ⟨*format*⟩ Determines how counterexamples are formatted. By default, they are printed as colored text (`text`), but they can also be written in `json` and `dot`.

Special variables can be truncated to a specified length *n* by writing `.`*n* between the `%` sign and the letter.

For example, we can check the CTL formula **A** □ **E** ◇ `eats(0)` with the following command:

```
$ umaudemc check philosophers.maude initial 'A [] E <> eats(0)'
The property is not satisfied in the initial state
(27 system states, 2734 rewrites)
```

The property is not satisfied, but we are not shown a counterexample. Among the supported backends, only NuSMV provides counterexamples for CTL properties, but it is not the first one in the ordered list of model-checking backends. We can use the `-c` flag to prefer a backend that provides counterexamples, if available.

```
$ umaudemc check philosophers initial 'A [] E <> eats(0)' -c
The property is not satisfied in the initial state
(27 system states, 98 rewrites)
| < (o | 0 | o) ψ (o | 1 | o) ψ (o | 2 | o) ψ >
v  rl < (o | Id | X) L ψ > => < (ψ | Id | X) L > [label left] .
| < (ψ | 0 | o) ψ (o | 1 | o) ψ (o | 2 | o) >
v  rl ψ (o | Id | X) => ψ | Id | X [label left] .
| < (ψ | 0 | o) ψ (o | 1 | o) (ψ | 2 | o) >
v  rl ψ (o | Id | X) => ψ | Id | X [label left] .
0 < (ψ | 0 | o) (ψ | 1 | o) (ψ | 2 | o) >
```

For branching-time properties, a counterexample (or an example for an existential property) cannot be a full execution, but the execution prefix *matched* by the first path quantifier can be obtained. For instance, the counterexample above shows a path to a state where no path satisfying `<> eats(0)` leaves. However, this property is satisfied if the system is controlled by the `parity` strategy in Section 1.1.

```
$ umaudemc check philosophers initial 'A [] E <> eats(0)' parity
The property is satisfied in the initial state
(12 system states, 328 rewrites)
```

To alternatively see whether the `release` rule is eventually executed, we can check the following μ-calculus property:

```
$ umaudemc check philosophers initial \
    'mu X . (< release > True \/ (<.> True /\ [.] X))' parity
The property is satisfied in the initial state
(18 system states, 104 rewrites, 129 game states)
```

Moreover, the tool can be used to verify linear-time properties as well, like the one we have checked in Section 4. In this case, we add the `elabel` and `slabel` options to simplify the printed counterexample.

```
$ umaudemc check philosophers.maude initial \
   '[] (<> eats(0) /\ <> eats(1) /\ <> eats(2))' parity \
   --elabel %l --slabel 'eats(1) = {%t |= eats(1)}'
The property is not satisfied in the initial state
(5 system states, 136 rewrites, 4 Büchi states)
| | eats(1) = false
| v  left
| | eats(1) = false
| v  right
| | eats(1) = false
| v  release
< v
```

The strategy argument admits arbitrary strategy expressions to control the system, unlike the interface in Section 4, which only accepts strategy names.

```
$ umaudemc check philosophers.maude initial '[] allEat(3)' 'turns(0, 3)'
The property is satisfied in the initial state
(10 system states, 128 rewrites, 4 Büchi states)
```

5.2 Probabilistic model checking with `pcheck`

Probabilistic model checking is available through the `pcheck` subcommand:

> `umaudemc pcheck` ⟨*filename*⟩ ⟨*initial state*⟩ ⟨*formula*⟩ [⟨*strategy*⟩] [`--assign` ⟨*method*⟩]

The *formula* argument can be either `@steady` to calculate steady-state probabilities, `@transient(`*n*`)` for the transient probabilities at the *n*-th step, or a formula in LTL, CTL, or PCTL in the following syntax extending the standard `LTL` module:

```
*** CTL
op A_ : Formula -> Formula [ctor prec 53] .
op E_ : Formula -> Formula [ctor prec 53] .

*** bounded step operators
op _U__ : Formula Bound Formula -> Formula [ctor prec 63 ...] .
op _R__ : Formula Bound Formula -> Formula [ctor prec 63 ...] .
```

```
op <>__ : Bound Formula -> Formula [prec 53 ...] .
op []__ : Bound Formula -> Formula [prec 53 ...] .
op _W__ : Bound Formula Formula -> Formula [prec 63 ...] .

*** bounded probability operator (PCTL)
op P__ : Bound Formula -> Formula [ctor prec 65 ...] .
```

A formula $\mathbf{P}_I\ \varphi$ in PCTL holds when the probability that φ is satisfied is in the interval $I \subseteq [0, 1]$. In the `P` operator of the Maude-based syntax, this interval is defined by a term of sort `Bound` built with

```
op <=_ : Float -> Bound [ctor] .        op <=_ : Nat -> Bound [ctor] .
op <_  : Float -> Bound [ctor] .        op <_  : Nat -> Bound [ctor] .
op >=_ : Float -> Bound [ctor] .        op >=_ : Nat -> Bound [ctor] .
op >_  : Float -> Bound [ctor] .        op >_  : Nat -> Bound [ctor] .

op [_,_] : Nat Nat -> Bound [ctor] .
op [_,_] : Float Float -> Bound [ctor] .
```

These bounds can also be attached to the temporal operators, although not all combinations are admitted by the backends. In addition to the general options described at the beginning of Section 5, some relevant modifiers are specific to this subcommand:

`--assign` ⟨*method*⟩ Sets the probability assignment method to one of those explained in Section 3. Instead of the literal description of the method, a filename prefixed by `@` may be entered to load it from file. If the selected method is `strategy`, the *strategy* argument must be filled. Otherwise, this argument is optional and the strategy would control the system in the standard way.

`--steps` For a reachability formula, calculates the expected number of steps instead of its probability.

`--reward` ⟨*term*⟩ For a reachability formula, calculates the expected reward for the given term instead of its probability. The sort of this term should be a numerical one (`Int`, `Float`, `Integer`, `Real`, ...) and it must contain at most one variable, which will be instantiated with every state to evaluate the reward. The `--steps` option can be seen as a shortcut for `--reward 1`.

`--raw-formula` The *formula* argument is directly passed to the backend, although it is scanned to find atomic propositions that should be evaluated in the model.

`--fraction` Results are printed as approximated fractions instead of decimal floating-point numbers.

For example, we can calculate steady-state probabilities for the philosophers example, which are only non-zero for the deadlock states.

```
$ umaudemc pcheck philosophers.maude initial '@steady' --backend storm
 0.5                < (ψ | 0 | o) (ψ | 1 | o) (ψ | 2 | o) >
 0.5                < (o | 0 | ψ) (o | 1 | ψ) (o | 2 | ψ) >
```

We have forced Storm as backend because the probabilities obtained from PRISM are 0.49999880...due to approximation errors. Transient probabilities can also be obtained with

```
$ umaudemc pcheck philosophers.maude initial '@transient(1)' --fraction \
    --assign 'uaction(left=2, right=3)'
 1/5                < (o | 0 | o) ψ (o | 1 | o) ψ (o | 2 | ψ) >
 1/5                < (o | 0 | ψ) (o | 1 | o) ψ (o | 2 | o) ψ >
 1/5                < (o | 0 | o) ψ (o | 1 | ψ) (o | 2 | o) ψ >
 2/15               < (ψ | 0 | o) ψ (o | 1 | o) ψ (o | 2 | o) >
 2/15               < (o | 0 | o) ψ (o | 1 | o) (ψ | 2 | o) ψ >
 2/15               < (o | 0 | o) (ψ | 1 | o) ψ (o | 2 | o) ψ >
```

The `--fraction` modifier has been used to see the probabilities as fractions, and `--assign` has selected the `uaction` method for assigning probabilities.

As examples of temporal properties, we can check the following:

```
$ umaudemc pcheck philosophers.maude initial '<> eats(0)'
Result: 0.4999996389661516 (relative error 7.3569252716515036e-06)
$ umaudemc pcheck philosophers initial '<> <= 3 eats(0)'
Result: 0.24999999999999997
```

The unbounded one holds with probability 1 under the `parity` strategy, so we can also compute the expected number of steps until `eats(0)` is satisfied.

```
$ umaudemc pcheck philosophers initial '<> eats(0)' parity --steps
Result: 4.799996844606567 (relative error 5.952947354729354e-06)
```

Assuming a function `eatCount` has been defined that counts how many philosophers are eating in a given state, we can also calculate the expected value of the reward:

```
$ umaudemc pcheck philosophers.maude initial '<> eats(0)' parity \
    --reward '2 * eatCount(M)'
Result: 1.599999367157242 (relative error 7.855416516353985e-06)
```

5.3 Statistical model checking with scheck

Statistical model checking is available by means of the scheck subcommand:

umaudemc scheck ⟨*filename*⟩ ⟨*initial state*⟩ ⟨QuaTEx *file*⟩ [⟨*strategy*⟩] [`--assign` ⟨*method*⟩]

The arguments shared with the `pcheck` subcommand specify the Maude model and the probabilistic assignment method to be applied on it. Some particularities should be taken into account:

- The assignment methods `step` and `pmaude` described in Section 3 are allowed in addition to those admitted by `pcheck`. However, methods starting with `mdp-` are not allowed since Monte Carlo simulations do not make sense on Markov decision processes. Moreover, methods with `ctmc-` behave exactly as their base methods.
- Failures in a strategy execution (either explicit with `fail` or implicit) may discard previous steps. Hence, deciding whether a rewrite is admissible under a strategy is undecidable and may require expanding its whole state space. Moreover, failed entries in `choice` or weighted `matchrew` combinators are not taken into account for distributing the probability, so this is not possible until every branch has been expanded. For all these reasons, the `strategy` method may be inefficient and it is not suitable for infinite state systems. Under the assumption that the strategy is free of failure, a new method `strategy-fast` is available to decide steps locally and irrevocably for a greater efficiency. Warnings will be shown in case a nondeterministic construct or conditional is used in the strategy, since their semantics may not be respected by this efficient execution mode. However, failures are allowed in the condition of a conditional expression consisting of tests only and the negative branch will be executed in that case.
- Strategies used in simulation are supposed to be free of unquantified nondeterministic choices. Nevertheless, if they are present, they would be resolved in an implementation-defined way for the `strategy-fast` and `step` methods. The `strategy` will shown an error message if unquantified nondeterministic behavior is detected.

The third argument specifies the path of a file including one or more queries in the language of Quantitative Temporal Expressions (QuaTEx) of the Vesta tool family [1]. Those general options at the beginning of Section 5 can also be given. Each query in the input file will be estimated an expected value and confidence interval by the Monte Carlo method on the executions of the probabilistic model. The simulation is controlled by the following parameters:

--alpha ⟨*number*⟩ Required significance level for the confidence interval, i.e., the long-run proportion of computed intervals that would not contain the true value of desired parameter. Its complement 1 – α is known as the confidence level of the interval and it must satisfy $0 \leq \alpha \leq 1$.

--delta ⟨*number*⟩ Maximum admissible radius for the confidence interval around the mean.

--nsims ⟨*range*⟩ Fixed number or bounds for the number of single simulations or samples. It can be either a single number, a pair *m*-*M* of a minimum *m* and maximum *M* number of executions, or a half-opened range. When the number of simulations is bounded above, the confidence level and interval radius in the previous arguments may not be attained. Its default value is `30-`.

--block ⟨*number*⟩ Number of simulations before checking the confidence interval again. The first round of samples can be larger if the minimum number of simulations is greater than this block size, and the last round may be shorter if the maximum number of simulations is reached.

--seed ⟨*number*⟩ Seed for the random number generator. The default value is `0`.

--jobs ⟨*number*⟩ Number of parallel simulation processes. By default, a single process will be used and `--jobs 0` will start as many jobs as CPU units in the machine.

--format ⟨*name*⟩ Output format for the simulation result, either `text` (the default) or `json`.

--assign ⟨*number*⟩ The probability assignment method, as explained above. Instead of the literal description of the method, a filename prefixed by `@` may be entered to load it from file. The default method is `step` when a strategy is provided and `uniform` otherwise.

--plot Plots the results of all parametric queries in the input file: a line chart will display the mean of each parametric query for the input value with the radius of the confidence interval highlighted around that line. Matplotlib is required for using this option.

Short versions of every option are defined with their first letter, like `-a` for `--alpha`. Simulations will be executed until the radius of the confidence interval with significance level α is below δ or when the maximum number of simulations is reached. If the SciPy package is installed, the reference distribution for computing the confidence interval will be the Student's t with as many degrees of freedom as the number of simulations less one. Otherwise, Python 3.8 is required and a normal distribution will be used. When the statistical model checker finishes, the estimated mean, variance, and confidence interval radius will be printed for each query. The temporary values of these parameters after each simulation block can be shown with the verbose `-v` flag.

For example, the following fragment defines a function `StepsFor` in QuaTEx that calculates the number of steps until the given number of heads have appeared. Two queries are introduced with `eval` for the expected value of this function under 10 and 20 steps.

```
StepsFor(n) = if (n == 0) then s.rval("steps")
              else if (s.rval("C␣==␣head") == 1) then #StepsFor(n - 1)
              else #StepsFor(n) fi fi;

eval E[StepsFor(10)];
eval E[StepsFor(20)];
```

The *next* operator # evaluates the function in the next step of the simulation, and `s.rval` reduces the given string as a Maude term of sorts `Int`, `Float`, `Integer`, `Real`, or `Bool` and returns the result as a floating-point number, where `true` and `false` are respectively converted to 1 and 0.[3] This term is called an *observation* and may contain a single variable, no matter whose name, that will be instantiated with the current state term. Moreover, the strings `time` and `steps` will be directly interpreted as the current time and number of steps. The current time is calculated as in a CTMC, regardless of whether the

[3] For the `pmaude` method, integer arguments *n* to `s.rval` are admitted as equivalent to the observation `val(n, C)`.

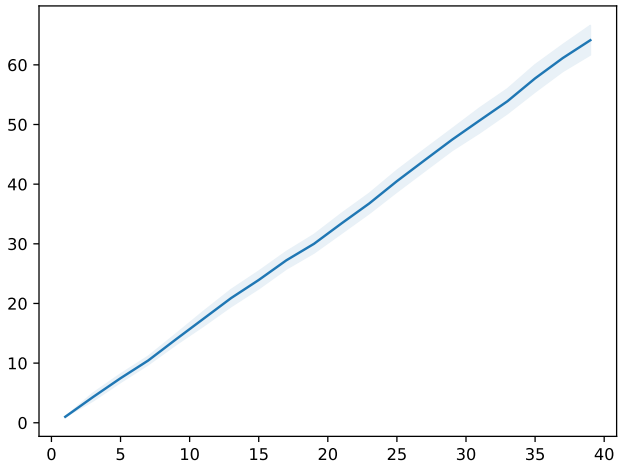


Figure 3: Plots of the confidence intervals, generated by scheck.

prefix `ctmc-` is used, for all methods but `strategy`, `step` (where `time` behaves like `steps`), and `pmaude`. Assuming the previous QuaTEx query is stored in a file `coin.quatex`, the following command evaluates that expression with scheck.

```
$ umaudemc scheck coin head coin.quatex -a 0.01
Number of simulations = 990
Query 1 (line 5:1)
  μ = 18.94242424242424     σ = 4.044209454647704    r = 0.3317202659896303
Query 2 (line 6:1)
  μ = 39.977777777777774    σ = 6.012416467669121    r = 0.49315951912519634
```

The exact expected values are 20 and 40.

Parametric queries are also supported using the syntax of MultiQuaTEx [29]. For example, instead of the two `eval` statements of the previous `coin.quatex` file, a parametric query can be written

```
eval parametric(E[StepsFor(x)], x, 1, 2, 40);
```

to evaluate `StepsFor` every two units in the interval [1, 40]. In the command below, we use the `choice` strategy for assigning probabilities to the model with the default `step` method, and fix the maximum admissible radius to 5 steps.

```
$ umaudemc scheck coin.maude head coin.multiquatex \
    'choice(2 : ttail, 3 : thead)' -d 5 --plot
Number of simulations = 30
  x = 1.0   μ = 1.0             σ = 0.0              r = 0.0
  x = 3.0   μ = 4.3             σ = 1.7449434335263  r = 0.651572586374
...
  x = 37.0  μ = 61.13333333333  σ = 6.009953429926   r = 2.244153492364
  x = 39.0  μ = 64.13333333333  σ = 6.600592101618   r = 2.464701596981
```

In addition to the text output, the `--plot` flag causes the confidence intervals to be represented graphically using Matplotlib, as shown in Figure 3. Moreover, the option `--format json` can be useful to obtain the model-checking results in a reusable format for more advanced analyses and visualizations.

The following is the grammar of the QuaTEx language admitted by scheck, where *Id* represents an identifier, *Var* is a variable name, and *Lit* a floating-point, integer, Boolean, or string literal.

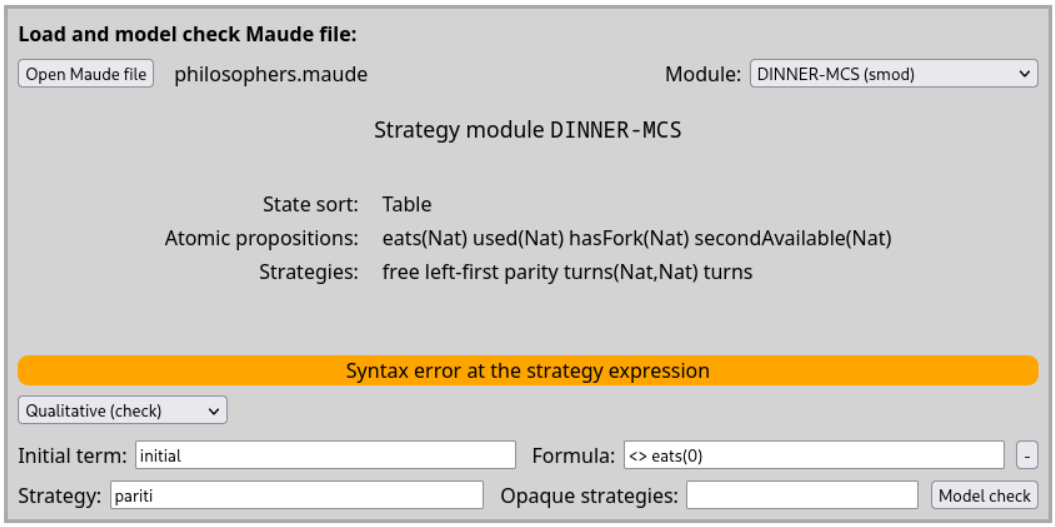


Figure 4: Model selection screen

```
   Q ::= Def * E +
   E ::= eval E[ PExp ]; | eval parametric(E[ PExp ], Var, Lit, Lit, Lit);
 Def ::= Id(Var, ..., Var) = PExp ;
PExp ::= SExp | # Id(SExp, ..., SExpr) | if SExp then PExp else PExp fi
SExp ::= Lit | SExp Op SExp | Var | s.rval(SExp)
  Op ::= + | - | * | == | > | >= | < | <= | ! | && | ||
```

Line comments are introduced with `//`. Admissible *PExp* expressions should represent floating-point or integer values.

5.4 Graph generation with `graph`

The program umaudemc can also be used to output the rewrite graph and strategy-controlled rewrite graph used for model checking in GraphViz's DOT format with the subcommand `graph`.

`umaudemc graph` ⟨*filename*⟩ ⟨*initial state*⟩ [⟨*strategy*⟩]

The common arguments for selecting modules, `merge-states`, `purge-fails`, the formatting options `elabel` and `slabel` can be passed too. Moreover, the subcommand has some specific arguments:

`--depth` ⟨*depth*⟩ Limits the graph to the states that are reachable from the initial state by at most the given number of steps.

`--passign` ⟨*method*⟩ Specifies a probability assignment method for generating graphs of probabilistic models. This is equivalent to the `assign` option of `pcheck` (see Section 5.2).

`--aprops` ⟨*list*⟩ Comma-separated list of atomic propositions to be written as state annotations in the output file (for SMV, PRISM, and JANI output only).

`--format` ⟨*name*⟩ Selects the output format of the graph, among `dot`, `tikz`, `nusmv`, `spin`, `prism`, and `jani`.

`-o` ⟨*filename*⟩ Outputs the graph to a file instead of the standard output. If the file extension is `pdf`, the `dot` command will be called if available to directly produce the PDF file. If the extension is `smv`, `pm`, `pml`, or `jani`, a SMV, PRISM, Spin (Promela), or JANI model will be generated instead of DOT, respectively. If `--format` is present, the output format will not be guessed.

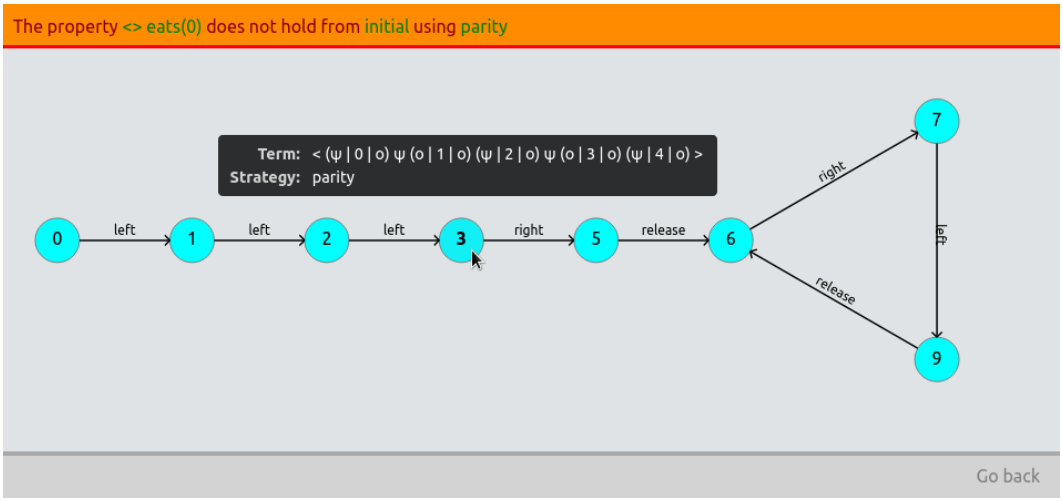


Figure 5: Counterexample view screen

5.5 A graphical interface

Alternatively to the command-line interfaces described in the previous sections, the model checker can be used from a graphical user interface, included in the umaudemc program[4].

`umaudemc` [`gui` [`--web`] [`--backends=`⟨*list of backends*⟩]]

When executing umaudemc as above, a local server will start and a web-browser window will be opened in its home page. As depicted in Figure 4, users should select the source file and Maude module they want to verify. Relevant information about the state sort, the atomic propositions, and the available strategies is shown. An initial state and a formula in any supported logic must be entered in their corresponding fields. The strategy field can be filled not only with a strategy name but with an arbitrary strategy expression, and it can also be left blank for model checking without strategies. Opaque strategies are introduced as a space-separated list of names. Syntax errors will be reported when the *Model check* button is activated.

A message will indicate that model checking has started, offering the possibility of cancelling the operation. As soon as the model checker finishes, the result will appear in place of that message. In case a counterexample is available, it will be shown as in Figure 5. By hovering over any of the states, more information is printed, like the current term and the next strategy to be executed from the state. Various configuration options like the precedence of model-checking backends can be set in the command-line invocation, as described with `umaudemc gui --help`.

Probabilistic model checking is also available within the web interface, under the advanced options panel that can be expanded with the + sign button above the *Model check* button. The leftmost dropdown list in the panel allows choosing between qualitative and quantitative model checking, and another dropdown list will enumerate all possible assignment methods once the quantitative variant is selected. Instructions are shown for filling the probabilistic data.

Notice that the web-based interface is intended for local use, and it will be attached to a local address by default. However, the server address and port can be changed with the `--address` flag. Anyone opening the webpage will be able to access the whole filesystem and initiate model-checking tasks. An option `--rootdir` is available to limit filesystem access to a specific directory, but neither the builtin Python server nor this interface actively try to prevent non-legitimate use.

[4] The graphical interface included in umaudemc is based on a previous discontinued program called `smcview` without support for branching-time properties.

Backend	LTL	CTL	CTL*	μ-calculus
Maude	on-the-fly			
LTSmin	on-the-fly	✓	✓	✓
pyModelChecking	tableau	✓	✓	
NuSMV	tableau	✓		
Spot	automata			
Spin	automata			
`umaudemc`'s		✓		✓

Table 1: Temporal logics supported by each backend

5.6 External model checkers and their installation

The umaudemc utility does model checking by alternatively calling the builtin Maude model checker, some algorithms implemented as part of it, and some external model checkers. Without anything else, LTL, CTL and μ-calculus properties can be directly checked using the first two options. However, for using other model checkers like LTSmin and NuSMV their corresponding programs have to be installed or downloaded in a location where the utility is able to find them.

- For LTSmin, download the model checker at ltsmin.utwente.nl, extract the package, and set the environment variable `LTSMIN_PATH` to its binary directory. The environment variable `MAUDE-MC_PATH` should also be set to the full path of the Maude-LTSmin plugin included in the download package. However, for model checking μ-calculus properties containing both atomic propositions and rule labels, a modified version of LTSmin is required as well as the `pbespgsolve` tool from mCRL2 [4]. A ready-to-use package[5] can be downloaded from maude.ucm.es/strategies. More details are given in Appendix A.
- For NuSMV, download this model checker from nusmv.fbk.eu, extract the package, and set the environment variable `NUSMV_PATH` to the path where the `NuSMV` binary resides.
- For `pyModelChecking`, install this Python library with `pip install pyModelChecking` or equivalent method.
- For Spot, download this library from spot.lrde.epita.fr. There are installation instructions in its website. Once the Python package is installed, it will be available for umaudemc.
- For Spin, download this tool from www.spinroot.com, extract the package, and set the environment variable `SPIN_PATH` to the path containing the `spin` binary. Alternatively, there are Spin packages that can be directly installed in the repositories of Debian/Ubuntu, MacPorts, etc.

Some backends and their connections may be more efficient than others (in general, the builtin Maude model checker and LTSmin should be the best choices), some easier to install, and not all of them support all temporal logics, as shown in Table 1. In particular, the renowned model checkers NuSMV and Spin do not have convenient interfaces to communicate the low-level Kripke structure derived by the Maude models, so they are likely less efficient and scalable than others. Given a temporal property, umaudemc will choose the first supported backend available to model check it. They are chosen in the order they appear in the table, which can nevertheless be modified with the following option:

`--backend` ⟨*list*⟩ Indicates a comma-separated list of model-checking backends that will be used to check the given properties, among `maude`, `ltsmin`, `pymc`, `nusmv`, `spin`, and `builtin`.

All arguments passed to umaudemc after a pair `--` of dashes will be passed directly to the backends, in case they are external programs (LTSmin, NuSMV, and Spin).

For probabilistic model checking, either PRISM [16] or Storm [14] are required.

[5] The modification has been proposed to be included in the upstream LTSmin.

- For PRISM, download it from www.prismmodelchecker.org and set the `PRISM_PATH` environment variable to the path containing the `prism` binary. Since PRISM is partially written in Java and starting its virtual machine takes a non-negligible amount of time, the model checker can alternatively be executed in server mode with `prism -ng` and the client `ngprism` be called for every subsequent verification task. For making umaudemc call `ngprism`, set `PRISM_PATH` to the full path of `ngprism`.
- For Storm, read the installation instructions in www.stormchecker.org and set the `STORM_PATH` enviroment variable to the path of the `storm` binary.

Again the `--backend` command receives a comma-separated list of model-checking backends, in this case, among `prism` and `storm`. All arguments passed to umaudemc after a pair `--` of dashes will be passed directly to the backends.

6 The statistical simulator for MultiVeStA

MultiVeSta [29] is a statistical analysis tool that extends the earlier Vesta [1] and PVesta [2]. The tool follows a client-server architecture where discrete-event simulators with a simple interface are executed by the statistical engine to estimate quantitative temporal expressions in the QuaTEx language, already introduced in Section 5.3. We have written a simulator for probabilistic Maude models specified as in Section 3 and a helper script `mvmaude` to use MultiVeSta with a syntax similar to umaudemc (see Section 5.3).

`mvmaude` ⟨*filename*⟩ ⟨*initial state*⟩ ⟨QuaTEx *file*⟩ [⟨*strategy*⟩] [`--assign` ⟨*method*⟩]

In addition to the standard `--module`, `--metamodule`, and `--opaque` arguments, `mvmaude` may be passed an `--assign` option with the probability assignment method from those in Section 3 with the particularities of Section 5.2.

The QuaTEx or MultiQuaTEx file (third argument) defines the value to be estimated by the Monte Carlo simulation. Moreover, the strings `time`, `steps`, and `completed` are directly interpreted by MultiVeSta as the current simulation time, the number of steps, and whether the simulation is completed.

```
$ mvmaude coin.maude head coin.multiquatex -- -d1 0.5
MultiVeStA client: analysis completed.
 Samples generated for query 0 (StepsFor(20.0)): 2370

MultiVeStA client: Results:
 result of query 0 (StepsFor(20.0)):
     39.017721518987344 [var: 37.98533376792006, ci/2: 0.24813149455606678]
```

The result is the computed expected value of the expression, the variance of the observations, and the radius of a confidence interval for the value. This radius has been limited with the `-d1 0.5` option that is directly forwarded to MultiVeSta, as any option before the first occurrence of `--` in the command line. Parametric queries are also supported by MultiVeSta and plots are generated for them. For instance, using the same `StepsFor` definition, the instruction

```
eval parametric(E[StepsFor(x)], x, 1, 2, 40);
```

evaluates `StepFor` in the partition of the interval [1, 40] whose steps are separated by 2 units. Figure 6 shows the result of this query for the coin example with the strategy `choice(2 : ttail, 3 : thead)` and the default `step` method.

For configuring the simulation parameters, as shown before with the `-d1` option, any option passed to `mvmaude` after its own arguments and two consecutive dashes `--` will be directly forwarded to MultiVeSta. The most relevant parameters are the following:

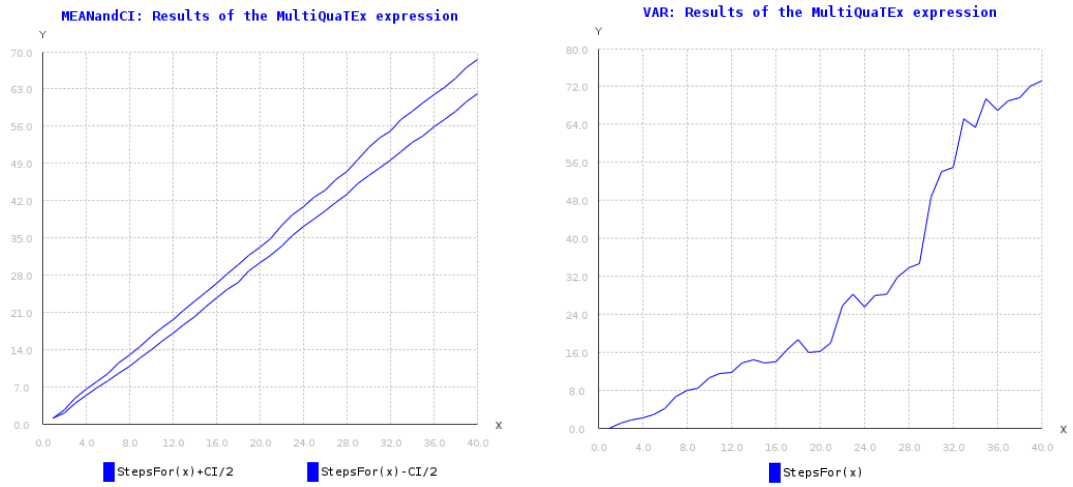


Figure 6: Plots of the confidence interval limits and variance, generated by MultiVeSta.

-a ⟨*number*⟩ The α coefficient of the confidence interval (that is, the probability that the estimated parameters falls outside the calculated interval).

-d1 ⟨*number*⟩ A δ bound for the diameter of the confidence interval (multiple distinct deltas may be given with -ds where there are multiple queries).

-l ⟨*number*⟩ Number of parallel simulation processes.

-bs ⟨*number*⟩ Number of simulations before checking the confidence interval.

-ms ⟨*number*⟩ Maximum number of simulations.

-sots ⟨*number*⟩ Seed of the generator of random seeds for the simulator instances. By default or when passing -1, the current time is used.

-op ⟨*path*⟩ Output path for the CSV and plot files produced by MultiVeSta.

-help Show the help message with the enumeration of these and more command-line options.

More information on MultiVeSta can be found in its repository github.com/andrea-vandin/MultiVeStA and in [29].

Appendices

A The Maude language extension for LTSmin

LTSmin [15] is a language-independent model checker with support for a wide range of logics. Using LTSmin and a language module that connects it with Maude, it is possible to check branching-time properties expressed in CTL* and μ-calculus. The plugin, a shared library called `libmaudemc.so` for Linux or `libmaudemc.dylib` for macOS, is shipped in the packages available at maude.ucm.es/-strategies. The recommended way of using this model checker is through the umaudemc utility, which is simpler and avoids learning the concrete syntax of its temporal logics and other cumbersome configuration details. However, it is still possible to use LTSmin tools directly.

The Maude language module implements the *Partitioned Next-State Interface* (PINS) that allows LTSmin's tools to communicate with Maude specifications. The suitable programs are `pins2lts-seq` for sequential or `pins2lts-dist` for distributed explicit-state model checking, and `pins2lts-sym` for symbolic-state model checking. The multi-core tool `pins2lts-mc` only works with a single process (`--procs=1`). The options of the different tools are described in their documentation [17] and can be printed by the command with the `--help` option. They include flags to indicate the formulae to be checked like `invariant`, `ltl`, `ctl`, `ctl-star`, `mu` and `mucalc`. These formulae may use the state labels (atomic propositions) and edge labels (rule labels) of the model, which are listed by passing the `--labels` option to the command. The path of `libmaudemc.so` must be indicated with the `--loader=`⟨*path*⟩ option.

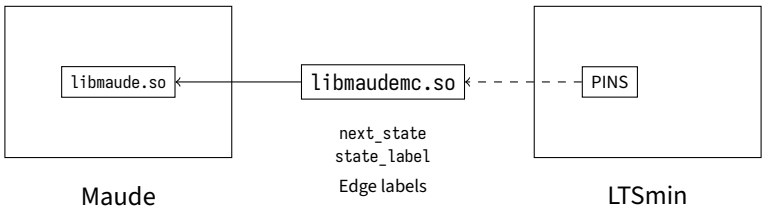


Figure 7: Architecture of the Maude LTSmin plugin

While not being recommended, because umaudemc facilitates these tasks, the LTSmin toolset can be used directly if desired. A typical model-checking invocation will have this form:

```
pins2lts-seq --loader=libmaudemc.so --initial=⟨term⟩ --strat=⟨sexpr⟩ \
             --aprops=⟨atomic props⟩ --ctl=⟨formula⟩
```

where `ctl` should be replaced to any other supported logic. The appropriate tool for the required logic must be chosen manually by changing the initial command suffix `seq` accordingly. In addition to the parameters of the own LTSmin tools, these commands accept other options that are specific to our language module. They are also listed in a separate section if the `--help` option is present:

`--module` ⟨*name*⟩ As in umaudemc.

`--metamodule` ⟨*term*⟩ As in umaudemc.

`--aprops` ⟨*list*⟩ Provides a comma-separated list of atomic propositions that would appear in the formulae. The full Maude term for the proposition must be written unchanged here, but they must have all non-alphanumeric characters escaped with a backslash when they appear in formulae.

Even though this information could be obtained in principle from the formula itself, it is not communicated to language plugins. Since propositional terms in Maude may take arguments and so be infinitely many, those used in the formula must be provided explicitly here. This disadvantage is avoided by using umaudemc.

--merge-states ⟨*option*⟩ The same values that in umaudemc are admitted except `default`. The `no` option is `none` here, and it is the default. This value will not be chosen automatically, and it should be set manually to respect the expected semantics on branching-time properties.

--purge-fails Enables the elimination of failed states, which is usually convenient for logics other than LTL. These states do not lead to a solution or execution loop, so they are not considered part of the allowed executions.

--biased-matchrew Activates the biased rewriting of subterms described in Section 2.2. Notice that it is the opposite of the `full-matchrew` in umaudemc.

--opaque-strats ⟨*list*⟩ Indicates a comma-separated list of strategies to be considered opaque, as described in Section 2.3.

Moreover, if the Maude prelude is not available in the plugin directory, its directory path should be given with `MAUDE_LIB`.

B The Kleene-star semantics of the iteration

The usual meaning of the star operator in formal languages and regular expressions is the Kleene closure, in other words, all finite repetitions of the argument. On the contrary, the iteration operator α* of the strategy language also allows the infinite repetition of α, which may be denoted by α^ω. Otherwise, a system controlled by a strategy could not be represented in general by a plain transition system or Kripke structure, since preventing infinite iterations imposes fairness-like restrictions on the infinite behavior of the model that cannot be represented locally. However, these restrictions can be handled by using automata-based algorithms or by pushing them to the temporal formulae being checked. As a method for specifying fairness restrictions directly on the strategy, the umaudemc tool supports those approaches [25].

The `check` command can be passed a flag `--kleene-iteration` or simply `-k` to make the iteration be interpreted as the Kleene star when checking LTL, CTL, or CTL* properties (μ-calculus is not supported). If the Spot backend is available, LTL properties will be handled by an extension of the automata-theoretic model-checking approach where the model is a Streett automaton that captures the finiteness restrictions of the iteration. Otherwise, and for all other supported logics, the temporal formulae are extended with premises describing the aforementioned restrictions. Unless there are no iterations, CTL properties become proper CTL* formulae after the transformation, so a model-checking backend for this more general logic is required and performance could be substantially affected.

The Kleene-star semantics model checker does not directly use the transition system produced by Maude for the strategy-controlled system, but relies on a Python-based implementation of the strategy language included in umaudemc, where iterations can be effectively traced. At the moment, the `--full-matchrew` option of the command is always enabled, and the `--merge-states` choice is not properly respected by the LTSmin backend.

References

[1] G. A. Agha, J. Meseguer, and K. Sen. PMaude: rewrite-based specification language for probabilistic object systems. In A. Cerone and H. Wiklicky, editors, *Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages, QAPL 2005, Edinburgh, UK, April 2-3, 2005*, volume 153(2) of *Electronic Notes in Theoretical Computer Science*, pages 213–239. Elsevier, 2006.

[2] M. AlTurki and J. Meseguer. PVeStA: A parallel statistical model checking and quantitative analysis tool. In A. Corradini, B. Klin, and C. Cîrstea, editors, *Algebra and Coalgebra in Computer Science - 4th International Conference, CALCO 2011, Winchester, UK, August 30 - September 2, 2011. Proceedings*, volume 6859 of *Lecture Notes in Computer Science*, pages 386–392. Springer, 2011.

[3] O. Bournez and C. Kirchner. Probabilistic rewrite strategies. Applications to ELAN. In S. Tison, editor, *Rewriting Techniques and Applications, 13th International Conference, RTA 2002, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2378 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2002.

[4] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs, and T. A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In T. Vojnar and L. Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part II*, volume 11428 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2019.

[5] A. Casagrande. pyModelChecking. A simple Python model checking package, 2020.

[6] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: an opensource tool for symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002, Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[7] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott. *Maude Manual v3.4*. March 2024.

[8] F. Durán, S. Eker, S. Escobar, N. Martí-Oliet, J. Meseguer, R. Rubio, and C. Talcott. Programming and symbolic computation in Maude. *J. Log. Algebraic Methods Program.*, 110, 2020.

[9] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 - A framework for LTL and ω-automata manipulation. In C. Artho, A. Legay, and D. Peled, editors, *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129, 2016.

[10] S. Eker, N. Martí-Oliet, J. Meseguer, R. Rubio, and A. Verdejo. The Maude strategy language. *J. Log. Algebraic Methods Program.*, 134:100887, 2023.

[11] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In M. Archer, T. B. de la Tour, and C. Muñoz, editors, *Proceedings of the 6th International Workshop on Strategies in Automated Deduction, STRATEGIES 2006, Seattle, WA, USA, August 16, 2006*, volume 174(11) of *Electronic Notes in Theoretical Computer Science*, pages 3–25. Elsevier, 2007.

[12] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In F. Gadducci and U. Montanari, editors, *Proceedings of the Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19-21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 162–187. Elsevier, 2004.

[13] M. Fernández, H. Kirchner, and B. Pinaud. Strategic port graph rewriting: an interactive modelling framework. *Math. Struct. Comput. Sci.*, 29(5):615–662, 2019.

[14] C. Hensel, S. Junges, J.-P. Katoen, T. Quatmann, and M. Volk. The probabilistic model checker STORM. *Int. J. Softw. Tools Technol. Transf.*, 23(4):1–22, 2021.

[15] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: high-performance language-independent model checking. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 692–707. Springer, 2015.

[16] M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 585–591. Springer, 2011.

[17] A. Laarman, M. Weber, J. Meijer, S. Blom, et al. LTSmin. Model checking and minimization of labelled transition systems. URL: `https://ltsmin.utwente.nl`.

[18] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[19] R. Rubio. An overview of the Maude strategy language and its applications. In K. Bae, editor, *Rewriting Logic and Its Applications - 14th International Workshop, WRLA@ETAPS 2022, Munich, Germany, April 2-3, 2022, Revised Selected Papers*, volume 13252 of *Lecture Notes in Computer Science*, pages 65–84. Springer, 2022.

[20] R. Rubio. Maude as a library: an efficient all-purpose programming interface. In K. Bae, editor, *Rewriting Logic and Its Applications - 14th International Workshop, WRLA@ETAPS 2022, Munich, Germany, April 2-3, 2022, Revised Selected Papers*, volume 13252 of *Lecture Notes in Computer Science*, pages 274–294. Springer, 2022.

[21] R. Rubio. *Model checking of strategy-controlled systems in rewriting logic*. PhD thesis, Universidad Complutense de Madrid, 2022.

[22] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Model checking strategy-controlled rewriting systems. In H. Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, 34:1–34:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[23] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Model checking strategy-controlled systems in rewriting logic. *Automat. Softw. Eng.*, 29(1), 2022.

[24] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. QMaude: quantitative specification and verification in rewriting logic. In M. Chechik, J.-P. Katoen, and M. Leucker, editors, *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings*, volume 14000 of *Lecture Notes in Computer Science*, pages 240–259. Springer, 2023.

[25] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Specifying fairness constraints and model checking with non-intensional strategies. In K. Ogata and N. Martí-Oliet, editors, *Rewriting Logic and Its Applications - 15th International Workshop, WRLA 2024, Luxembourg City, Luxembourg, April 6-7, 2024, Revised Selected Papers*, volume 14953 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 2024.

[26] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Strategies, model checking and branching-time properties in Maude. In S. Escobar and N. Martí-Oliet, editors, *Rewriting Logic and Its Applications - 13th International Workshop, WRLA 2020, Virtual Event, October 20-22, 2020, Revised Selected Papers*, volume 12328 of *Lecture Notes in Computer Science*, pages 156–175. Springer, 2020.

[27] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. Strategies, model checking and branching-time properties in Maude. *J. Log. Algebr. Methods Program.*, 123, 2021.

[28] R. Rubio, N. Martí-Oliet, I. Pita, and A. Verdejo. The semantics of the Maude strategy language. Technical report 01/21, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2021.

[29] S. Sebastio and A. Vandin. MultiVeStA: statistical model checking for discrete event simulators. In A. Horváth, P. Buchholz, V. Cortellessa, L. Muscariello, and M. S. Squillante, editors, *7th International Conference on Performance Evaluation Methodologies and Tools, ValueTools '13, Torino, Italy, December 10-12, 2013*, pages 310–315. ICST/ACM, 2013.