



**HAL**  
open science

# A Multivalued, Spatialized, and Timed Modelling Language for Social-Ecological Systems

Franck Pommereau, Cédric Gaucherel

► **To cite this version:**

Franck Pommereau, Cédric Gaucherel. A Multivalued, Spatialized, and Timed Modelling Language for Social-Ecological Systems. International Workshop on Petri Nets and Software Engineering (PNSE 2024), Jun 2024, Genève, Switzerland. pp.13–32. hal-04667502

**HAL Id: hal-04667502**

**<https://univ-evry.hal.science/hal-04667502v1>**

Submitted on 5 Aug 2024

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

# A Multivalued, Spatialized, and Timed Modelling Language for Social-Ecological Systems

Franck Pommereau<sup>1</sup>, Cédric Gaucherel<sup>2</sup>

<sup>1</sup>IBISC, Univ. Évry, Univ. Paris-Saclay, 91020 Évry-Courcouronnes, France

<sup>2</sup>AMAP-INRAE, CIRAD, CNRS, IRD, Univ. Montpellier, 34398 Montpellier, France

## Abstract

The EDEN framework provides tools for the formal modelling and analysis of social-ecological systems and their dynamics. In particular, it features the RR modelling language (for *reaction rules*) that allows defining systems as Boolean variables and guarded actions to update them. This language is equipped with a compact semantics in terms of Petri nets extended with read-arcs, inhibitor arcs, reset arcs, and transition priorities. Apart from the last one, these extensions can be implemented in standard Petri nets at the price of building bigger nets. In this paper, we extend the RR language with multivalued variables, spatial information inspired by membrane computing, and timing. The resulting language, called MRR (for *multivalued RR*), is equipped with a compact colored Petri nets semantics extended with only transition priorities.

## Keywords

formal modelling and analysis, social-ecological systems, multivalued variables, spatialized systems, timed systems

## 1. Introduction

The EDEN framework has been developed and used in the field of ecology for years [1, 2, 3, 4, 5, 6]. It provides a formal language called RR that can be used to model social-ecological systems, and the tool ecco that allows for analyzing their properties [7, 8]. From the beginning, EDEN has promoted abstraction as a path to generalization [4, 7, 9]. Indeed, data about ecosystems are often very sparse and trying to model precisely partially known systems leaves gaps that are hard to fill without “forging” new data. By staying with abstract models, it is possible to render available knowledge with an adequate precision, leaving details blur where data is lacking. A typical example is the rendering of a population of animals through a single Boolean variable that can be interpreted as “enough” or “not enough” individuals, according to some threshold, for having an observable effect on the system [9].

However, having only Boolean variables sometimes requires awkward modelling tricks, which leads to the need for *multivalued variables*. For instance, a population of animals may have distinct effects on the system depending on distinct thresholds that are known to exist even if not precisely quantified. In such a case, we would like to move from “not enough”/“enough” to, for instance, “not enough”/“low”/“high” [9, 10]. Another example where multivalued variables is desired is the modelling of seasons. Having four Boolean variables for the four seasons requires strong modelling discipline in order to ensure that exactly one is on at every state, and at the same time, it makes states more convoluted as four variables have to be examined to read the season [1].


This paper extends the RR language [4, 7, 11] by adding multivalued variables. Doing so, we are able to introduce a representation of *time* borrowed from [12, 13]. It consists in modelling clocks as counters equipped with ticks that increment them, which has been called *causal time* because clock ticks may causally depend on other events. Having time is crucial to represent loosely coupled phenomena, for instance one may want to specify that a process always evolves faster than another one, or that two processes have speeds in the same order of magnitude, which is difficult if not impossible without a model of time.

---

PNSE’24, International Workshop on Petri Nets and Software Engineering, 2024

✉ [franck.pommereau@univ-evry.fr](mailto:franck.pommereau@univ-evry.fr) (F. Pommereau)

ORCID 0000-0002-9959-3699 (F. Pommereau); 0000-0002-4521-8914 (C. Gaucherel)

 © 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

Finally, we further extend RR with a representation of *spatial information* inspired from membrane systems [14, 15] (also called P-systems). This feature has been requested by many users over the years as ecosystems generally present strongly spatialized structures. While this can be rendered with tricks such as prefixing the variables names with the locations they belong to, it quickly leads to awkward and convoluted models. Having this information in the modelling language leads to simpler models that express more naturally and more accurately the ecological knowledge.

In this paper, we thus present the MRR language that extends RR with these three aspects. Both languages use discrete *variables*, with *actions* to model their changes. Actions are split into *rules* and *constraints*, the latter having a higher priority. Actions are composed off a guard on the variables, and a series of assignments to update them. An MRR system consists of as a set of nested *locations* in which the variables and actions are declared. Each action may refer only to the variables in the same location where it was defined, or those in the location that immediately contains it, or those in the locations immediately contained in it. Variables in MRR are allowed to have several initial states, which means that the system itself is analyzed starting from several initial states [16]. But at any state the system can reach, every variable has only a single value. Variables are either single-valuated or array-valuated and may range over Boolean or ranges of integers. *Clocks* may be defined in the top location, each being a time reference responsible for incrementing a set of variables as it ticks. The MRR language is equipped with an operational semantics as well as with a consistent Petri nets semantics. The tool *ecco* [8, 7], that implements RR analysis, has been extended to support MRR while preserving its approach and spirit.

The next section describes the syntax and semantics of MRR, with an example to illustrate the operational semantics. Since MRR is an extension of RR, references to the latter will point out where the languages coincide. The Petri nets semantics is defined in Section 3. Section 4 describes the implementation of MRR in *ecco*, and shows a second example that is quickly analyzed within *ecco*. Before the conclusion, related works are presented and discussed.

## 2. Syntax and operational semantics

We start with a presentation of the concrete MRR language, from a modeller’s perspective. The concrete syntax is given for reference in Figure 1, but it is presented more intuitively below. The end of the section will define the language from a more abstract perspective, which will allow giving its formal semantics.

The structure of an MRR file is as follows:

```

1  clocks:
2      # here: clocks declarations
3  variables:
4      # here: variables declarations
5  # here: locations declarations
6  constraints:
7      # here: constraints declarations
8  rules:
9      # here: rules declarations

```

Above, `#...` are comments that start with `#` and end at the end of the line. Every part above is optional and may be skipped, but if a section (e.g. `variables:`) is used, then it must not be empty, and its whole content should be indented. MRR is using indentation as blocks delimitation like in the Python programming language.

### 2.1. Clocks

Every clock is declared as `NAME: description` on a single line. `NAME` must start with a letter and may continue with letters or digits. The name given to each clock will be used later on to link variables to this clock, in order to indicate that the linked variables have to be incremented when the clock ticks.

```

system ::= [clocks] model
clocks ::= "clocks:" ↵ →| clockdecl+ ↵
clockdecl ::= NAME ":" DESC ↵
model ::= [variables] location* [constraints] [rules]
variables ::= "variables:" ↵ →| vardecl+ ↵
constraints ::= "constraints:" ↵ →| actdecl+ ↵
rules ::= "rules:" ↵ →| actdecl+ ↵
location ::= "location" NAME [{" NAT "}] ":" ↵ →| model ↵
vardecl ::= type NAME "=" init ":" DESC ↵
           | NAME /["+*-]/ ":" DESC ↵
type ::= scalar [{" NAT "}]
scalar ::= "bool"
           | "{" [NAME ":"] INT "." INT "}"
init ::= simpleinit ("|" simpleinit)*
simpleinit ::= INT [{" "."] INT]
            | "(" simpleinit ("," simpleinit)* ")"
            | "*"
actdecl ::= [{" /["^\\"]+/ "}] condition ("," condition)*
           ">>" assignment ("," assignment)* [quantifier] ↵
condition ::= (var | INT) CMP (var | INT)
            | var /["+ -]/
assignment ::= var /["+ -]?="/ (var | INT)
            | var /["+*-]|\\+\\+| - -/
var ::= NAME [{" index "}] [{"@@"} | {"@"} NAME [{" index "}]]
index ::= NAT
        | NAME [{"+" | "-"} NAT]
quantifier ::= "for" ("any" | "all") NAME ("," NAME)* ("," quantifier)*
NAME ::= /[a-z][a-z0-9]*/
NAT ::= /[0-9]+/
INT ::= /[+-]?[0-9]+/
CMP ::= /|=|<=|<|>=|>|!="/
DESC ::= /^[^\\n\\r]+/
    
```

**Figure 1:** MRR concrete syntax in BNF style, using the following notations: *non-terminals*, `TERMINALS`, [optional content], (grouped content), 0-or-more-repetitions\*, 1-or-more-repetitions+, “literal text”, /regular expression/. Moreover, | denotes alternation, ↵ denotes end-of-line, →| denotes the starting of an indented block, and ↵ its ending. This syntax does not include syntactic constraints that are described in the text.

MRR is using the so-called causal time model of time in which a time reference is just a tick event in the system that increments variables used to measure the passing of time by counting the ticks [12, 13]. In this perspective, one builds chronometers as simple counters that get automatically incremented as time passes, that is, when tick events occur. These chronometers then become the time reference used throughout the model. It has been proved that this approach is a valid representation of time, measured discretely [17]. In MRR, chronometers are built in two steps; first we declare a clock as presented above, then we link a counter to it in order to have it incremented upon clock ticks.

In MRR, clocked variables (*i.e.* integer variables linked to clocks) are necessarily bounded as MRR only allows finite ranges for every variable. This ensures that every state-space (often called state-transition graph [18, Sec. 1.1] in ecology) is finite, even if it may be so huge that it cannot be computed. This has a consequence that is well known in causal time approaches: when a clocked variable has reached its maximal value, the corresponding tick cannot occur anymore as it would overcome the variable range. So time progression is blocked, but only in appearance. Indeed, when using causal time, two situations become logically equivalent:

- an action will always be taken before the next tick;
- no tick can occur until the action has been taken.

This is also called the *deadline paradox* that disappears when time is treated in a strictly causal way [19]. So, giving a bound to variables is a way to specify structural deadlines, or hard deadlines, which are deadlines that the system cannot violate in any situation. This differs from soft deadlines, or alarms, that are deadlines the system can violate, in which case it will start special behavior in reaction to the violation, like raising an alarm. MRR has provisions to avoid having to manage the structural deadlines that every clocked variable creates. It is possible in MRR to reset a clock variable to a special value `*` that lets time pass and ticks occur without changing the clocked variable. So, when a clocked variable is not used to measure anything, it can be set to `*`, but when one needs to measure the time between two actions, it can be assigned to `0` in one action and tested in a later action. Structural deadline are directly implemented by the maximum value of clocked variables, and alarms can be implemented as constraints whose priorities guarantee that every violated deadline is detected immediately.

## 2.2. Variables

The simplest form of a variable declaration in MRR is, like in RR, one of:

- `NAME+ : description` to declare a Boolean variable initialized on;
- `NAME- : description` to declare a Boolean variable initialized off;
- `NAME* : description` to declare a Boolean variable with two initial states on and off.

More generally, a variable is declared as `TYPE NAME = INIT : description` where:

- `TYPE` is the type of the variable, *i.e.* the range of values it can hold, which can be:
  - `bool` for a Boolean variable,
  - `{MIN..MAX}` for an integer variable ranging from `MIN` to `MAX`,
  - `{CLOCK: MIN..MAX}` for an integer variable that is incremented when `CLOCK` ticks;
- `NAME` is the name of the variable that is used to refer to it within actions, this name must be unique at the level of the location where it is declared;
- `INIT` is the set of initial values for the variables:
  - `INT` specifies a single value,
  - `INT..INT` specifies a full range of values,
  - `INIT|INIT` specifies the union of two sets of values,
  - `*` specifies the full range of possible values for the variable.

For instance, declaration `{0..1} var = 0 : my variable` is equivalent to the shorter declaration `bool var = 0 : my variable` (because Boolean are implemented as 0/1-valuated integers), or the even shorter `var- : my variable`. Declaration `{0..4} level = 2 : a description` is a declaration of a variable called `level` that ranges between 0 and 4 and is initially set to 2.

Note that, when a variable is clocked, initial value `*` has the special meaning that time can pass without changing the variable.

Finally, variables may be declared as arrays by adding `[SIZE]` after their type, where `SIZE` is a positive integer value. In such a case, `INIT` may be a single initialization as above to set all the array to the same initial value, or a sequence of initialization to set different initial values for each cell. For instance `bool[4] vec = (0, 1, 0, 0) : my array` is a declaration of a Boolean array of size 4 whose cells are all initialized to 0 except for the second one that is initialized to 1.

## 2.3. Nested locations

A nested location is declared as follows:

```

1 location NAME:
2   variables:
3     # ...
4     # here: further nested locations
5   constraints:
6     # ...
7   rules:
8     # ...

```

`location NAME`: introduces the new location, and its content is indented and corresponds exactly to the structure described above for the top location, except that clocks are not allowed here. Contrasting with the top location that has no name, a nested location is always named, and its name must be unique at the level of the location where it is declared. It is also possible to declare an array of locations by adding `[SIZE]` after its name, where `SIZE` is a positive integer value:

```

1 location NAME[SIZE]:
2   # as previously

```

Arrays of locations are convenient for declaring several identical locations that will evolve independently during the execution.

## 2.4. Actions: constraints and rules

Actions are all declared the same way, but in two separate sections to distinguish between constraints and rules. The only difference is that constraints execute with a higher priority. Every action is declared following the same structure `[TAG, ...] COND, ... >> ASSIGN, ... QUANT` where:

- `[TAG, ...]` is an optional comma-separated sequence of arbitrary tags that decorate the action, at the level of MRR they have no special meaning, but they may be used during the analysis;
- `COND, ...` is a mandatory comma-separated sequence of conditions, all of which must be validated to allow the execution of the action;
- `ASSIGN, ...` is a mandatory comma-separated sequence of assignments, all of which is performed upon the execution of the action;
- `QUANT` is a quantification of the free variables used in each `COND` or `ASSIGN` (see below).

A `COND` must be one of:

- `VAR OP VAL` or `VAL OP VAR` to compare a variable to a value;
- `VAR OP VAR` to compare two variables;
- `VAR+` like in RR, that is here a shorthand for `VAR==1`;
- `VAR-` like in RR, that is here a shorthand for `VAR==0`;

where `OP` is one of `==`, `!=`, `<`, `>`, `<=`, `>=`. Every `VAR` must be either:

- `NAME` to refer to a variable declared in the same location as the action;
- `NAME[IDX]` to refer to a cell of an array declared in the same location;
- `NAME@LOC` to refer to a variable declared in a location named `LOC` nested immediately inside that where the action is defined;
- `NAME[IDX]@LOC` to refer to an array in nested location `LOC`;
- `NAME@LOC[IDX]` to refer to a variable declared in a nested array of locations;
- `NAME[IDX]@LOC[IDX]` to refer to an array declared in a nested array of locations;
- `NAME@@` to refer to a variable declared in the enclosing location;
- `NAME[IDX]@@` to refer to an array declared in the enclosing location.

Then, every `IDX` must be either an integer literal, a free variable (a name that does not refer to a known variable, clock, or location), or a simple expression as `i+1` or `i-2` involving a free variable, plus or minus operator, and a constant. We will come back to free variables below.

On the right-hand side of an action, following `>>`, every `ASSIGN` must be one of:

- `VAR = EXPR` to assign a new value expressed by `EXPR` to `TARGET`;
- `VAR += EXPR` or `VAR -= EXPR` to perform so-called augmented assignment;
- `VAR++`, `VAR--` as shorthands for the longer forms above;
- `VAR+` like in `RR`, that is here a shorthand for `VAR=1`;
- `VAR-` like in `RR`, that is here a shorthand for `VAR=0`;
- `VAR*` that is a special form reserved to clocked variables that lets time pass (*i.e.* ticks occur) without changing the variable.

Here `VAR` is as described for conditions, and `EXPR` must be either another `VAR` or an integer.

As presented above, an `IDX` may refer to free variables, *e.g.* we could have an action such as `array[i]<2 >> array[i]++`. Such an action needs to be completed by a quantifier for `i` in order to specify its semantics.

First, `array[i]<2 >> array[i]++ for all i` means that the condition and assignment will be evaluated for all the possible values of `i`, which depends on the declaration of the array and on its size in particular. Assuming, for instance, that `array` was declared with size 2, the action above is equivalent to `array[0]<2, array[1]<2 >> array[0]++, array[1]++` which indeed checks the condition and perform the assignment for all possible `i`.

Then, `array[i]<2 >> array[i]++ for any i` means that the condition and assignment will be evaluated using any but only one of the possible values of `i`. Assuming the same size, this is equivalent to having actually two actions: `array[0]<2 >> array[0]++` and `array[1]<2 >> array[1]++`.

In general, quantifiers may be combined arbitrarily when there is more than one free variable. For instance, `arr[i]>vec[j] >> tab[k]++ for all i, j, for any k` is a valid action assuming `arr`, `vec`, and `tab` are arrays.

To conclude on arrays within actions, one special free variable `self` needs no quantification and can be used only from within an array of locations, in which case it refers to the index within the array of the location that executes the action. This is illustrated just below.

## 2.5. Example

Figure 2 shows a system formed of `NUM` nests inhabited by at most `POP` animals. Every animal can leave its nest at any time, and go back to its nest (and not another one) after that. Moreover, the nest cannot be left unoccupied for more than `MAX` time units. Before it is parsed, a `.mrr` file is preprocessed using the C preprocessor, which allows parametrizing the model by `NUM`, `POP`, and `MAX`. As `ecco` allows defining these parameters when loading the model, we wrap default definitions within `#ifdef / #endif` to avoid unexpected overrides (lines 1 to 9).

Line 12, we declare a clock to measure time throughout the system. Line 15, we declare an array `out` that holds `NUM` values within `0..POP` all initialized to 0, to count the animals outside each nest. Line 17, we declare an array `nest` of `NUM` locations, each has one variable `in` within `0..POP` that counts how many animals are present inside the nest, which is initialized to `POP`. Location `nest` also declares a clocked variable `time` to measure how long the nest has been left empty, it is initialized with `*` which is interpreted as “no time is measured” for clocked variables.

Line 23, a rule implements one animal leaving the nest in the case it is not the last one to leave. Note how we use the variable `self` to increment the cell of `out` in the top location that corresponds to the nest the animal is leaving. Line 25 is a similar rule but when the last animal leaves the nest, in which case `time` is set to 0, so that time starts to be counted. Finally, line 29, a rule implements one animal going back to its nest. We use here a free variable `i` quantified `any` to choose a nest such that `out[i]>0`. Note how `time` is reset to `*` and `in` is incremented in the same nest.

```

1  #ifndef NUM
2  #define NUM 2
3  #endif
4  #ifndef POP
5  #define POP 2
6  #endif
7  #ifndef MAX
8  #define MAX 2
9  #endif
10
11 clocks:
12     tick: unique clock
13
14 variables:
15     {0..POP}[NUM] out = 0: how many animals from each nest are outside
16
17 location nest[NUM]:
18     variables:
19         {0..POP} in = POP: how many animals are currently in this nest
20         {tick: 0..MAX} time = *: how long the nest has been empty
21     rules:
22         # one animal, that is not the last in the nest, leaves it
23         in>1 >> in--, out[self]@++
24         # last animal leaves the nest
25         in==1 >> in=0, out[self]@++ , time=0
26
27 rules:
28     # an animal that is out can go back to the nest it came from
29     out[i]>0 >> out[i]--, in@nest[i]++, time@nest[i]* for any i

```

**Figure 2:** Example model with nests hosting animals.

This model has a static number of nests and animals in each nest, but this can be changed easily thanks to the `#define` directives at the beginning. An MRR model is always passed to the C preprocessor before it is parsed, so MRR supports the full range of C preprocessor directives even if not all are meaningful in this context.

## 2.6. Abstract syntax and flattening

An MRR system and its constituting elements will be formally defined in terms of structures with typed fields, like in programming languages, and we will use the dot notation to access these fields. For instance, if  $m$  is a structure  $\langle \text{clocks} | \text{top} \rangle$  then  $m.\text{clocks}$  and  $m.\text{top}$  are its two fields. We implement value `*` used to assign clocked variables as  $-1$ , thus, we assume that the domain of every clocked variable, is augmented with value  $-1$  that does not need to be declared in the concrete syntax but will be present in the abstract syntax so that variables are valued in  $\mathbb{Z}$  instead of in  $\mathbb{N}$ . Then, we consider a set  $\mathbb{I}$  of identifiers, *i.e.* arbitrary names including forms like `nest[0]` that is considered as an identifier. Names can be combined with dots, to form new names, for instance, if `nest[0]` and `in` are identifiers, then `nest[0].in` is also an identifier.

**Definition 1 (MRR abstract syntax).** *A variable is a structure  $\langle \text{name} | \text{dom} | \text{init} | \text{clock} \rangle$  where:*

- $\text{name} \in \mathbb{I}$ ;
- $\text{dom} \subset \mathbb{Z}$  is its domain;
- $\text{init} \subseteq \text{dom}$  is its set of initial values;
- $\text{clock} \in \mathbb{I} \cup \{\perp\}$  is its clock, if any.

*A condition is a structure  $\langle \text{left} | \text{op} | \text{right} \rangle$  where:*



- left  $\in \mathbb{I}$ ;
- op  $\in \{ ==, !=, <, >, <=, >= \}$ ;
- right  $\in \mathbb{I} \cup \mathbb{Z}$ .

An expression is a structure  $\langle \text{left}|\text{op}|\text{right} \rangle$  where:

- left  $\in \mathbb{I}$ ;
- op  $\in \{ +, -, *, / \}$ ;
- right  $\in \mathbb{I} \cup \mathbb{Z}$ .

An assignment is a structure  $\langle \text{target}|\text{value} \rangle$  where:

- target  $\in \mathbb{I}$ ;
- value is an expression or a value in  $\mathbb{Z}$ , the augmented assignments allowed by the concrete syntax are thus here translated to regular assignments.

An action is a structure  $\langle \text{left}|\text{right}|\text{tag} \rangle$  where:

- left is a set of conditions;
- right is a set of assignments;
- tag is arbitrary text.

A location is a structure  $\langle \text{name}|\text{vars}|\text{locs}|\text{cons}|\text{rules} \rangle$  where:

- name  $\in \mathbb{I}$ ;
- vars is a set of variables;
- locs is a set of locations;
- cons is a set of actions;
- rules is a set of actions.

An MRR system is a structure  $\langle \text{clocks}|\text{top} \rangle$  where:

- clocks  $\subset \mathbb{I}$ ;
- top is a location such that  $\text{top.name} = \varepsilon$ . ■

An MRR system  $m$  so defined must validate varied constraints, in particular no two items declared at the same level can have the same name. For instance, for every location  $l$ , we must have  $\{l.\text{name}\}$ ,  $\{v.\text{name} \mid v \in l.\text{vars}\}$ ,  $\{s.\text{name} \mid s \in l.\text{locs}\}$ , and  $m.\text{clocks}$  be pairwise disjoint. Moreover, a variable cannot be assigned twice in the right-hand-side of an action. Finally, in an MRR obtained from the concrete syntax, the identifiers involved in actions must be either:

- names of variables defined in the same location as the action;
- names of the form  $..\text{name}$  to refer to variables defined in the enclosing location;
- names of the form  $s.\text{name}$  to refer to variables defined in a sub-location  $s$ .

With respect to the concrete syntax defined in the beginning of the section, the abstract syntax does not provide arrays that are thus unrolled into their components. Thus, a variable `bool[2] b` is considered as two variables `b[0]` and `b[1]` ranging on  $\{0, 1\}$ . And similarly with arrays of locations. In this latter case, `self` is statically replaced with the index of the location when it is unrolled. Moreover, `for all` and `for any` are also unrolled by copying the terms in the former case and copying the whole action in the latter case.

Finally, an MRR system can be flattened by defining all its variable in its top-level locations with a name referring to their paths within the nested structures. For instance, the variables in the model from Figure 2 can be flattened as `out[0]`, `out[1]`, ..., `nest[0].in`, `nest[1].in`, ..., `nest[0].time`, `nest[1].time`, ..., depending on `NUM`. And then actions can also be all defined at the top-level by replacing the variables they involve by their flattened names. Such a flattened system has only a top-location with no nested locations and is much handier to define the operational and Petri nets semantics. Figure 3 shows how the model from Figure 2 is flattened.

- variables:
  - $\langle \text{name: out}[0] \mid \text{dom: } \{0, \dots, 5\} \mid \text{init: } \{0\} \mid \text{clock: } \perp \rangle$
  - $\langle \text{name: out}[1] \mid \text{dom: } \{0, \dots, 5\} \mid \text{init: } \{0\} \mid \text{clock: } \perp \rangle$
  - $\langle \text{name: nest}[0].\text{in} \mid \text{dom: } \{0, \dots, 5\} \mid \text{init: } \{5\} \mid \text{clock: } \perp \rangle$
  - $\langle \text{name: nest}[0].\text{time} \mid \text{dom: } \{-1, 0, \dots, 3\} \mid \text{init: } \{-1\} \mid \text{clock: tick} \rangle$
  - $\langle \text{name: nest}[1].\text{in} \mid \text{dom: } \{0, \dots, 5\} \mid \text{init: } \{5\} \mid \text{clock: } \perp \rangle$
  - $\langle \text{name: nest}[1].\text{time} \mid \text{dom: } \{-1, 0, \dots, 3\} \mid \text{init: } \{-1\} \mid \text{clock: tick} \rangle$
- rules:
  - $\langle \text{left: } \{ \text{out}[0] > 0 \} \mid \text{right: } \{ \text{out}[0] - = 1, \text{nest}[0].\text{in} + = 1, \text{nest}[0].\text{time} = - 1 \} \rangle$
  - $\langle \text{left: } \{ \text{out}[1] > 0 \} \mid \text{right: } \{ \text{out}[1] - = 1, \text{nest}[1].\text{in} + = 1, \text{nest}[1].\text{time} = - 1 \} \rangle$
  - $\langle \text{left: } \{ \text{nest}[0].\text{in} > 1 \} \mid \text{right: } \{ \text{nest}[0].\text{in} = - 1, \text{out}[0] + = 1 \} \rangle$
  - $\langle \text{left: } \{ \text{nest}[0].\text{in} = = 1 \} \mid \text{right: } \{ \text{nest}[0].\text{in} = 0, \text{out}[0] + = 1, \text{nest}[0].\text{time} = 0 \} \rangle$
  - $\langle \text{left: } \{ \text{nest}[1].\text{in} > 1 \} \mid \text{right: } \{ \text{nest}[1].\text{in} = - 1, \text{out}[1] + = 1 \} \rangle$
  - $\langle \text{left: } \{ \text{nest}[1].\text{in} = = 1 \} \mid \text{right: } \{ \text{nest}[1].\text{in} = 0, \text{out}[1] + = 1, \text{nest}[1].\text{time} = 0 \} \rangle$

**Figure 3:** The variables and rules of the model from Figure 2 once flattened. For readability, conditions and assignments have been kept simplified instead of being fully expanded as structures. Action tags are all empty and thus completely omitted.

## 2.7. Operational semantics

The state of a flattened MRR system is a vector of integers indexed by its variables. For a state  $s$  and a variable  $v$ , we denote by  $s[v]$  the value of  $v$  at state  $s$ . From the declarations of these variables, several initial states may be defined. Our example, however, has a unique initial state.

**Definition 2 (Operational semantics).** *Let  $m$  be a flattened MRR. From a state  $p$ , an action  $a \in m.\text{top.cons} \cup m.\text{top.rules}$  is enabled if:*

- all the conditions in  $a.\text{left}$  evaluate to true at  $p$ ;
- calling  $q$  the state defined by applying onto  $p$  all the assignments in  $a.\text{right}$ , then we must have  $p \neq q$ .

Firing  $a$  in this situation is denoted by  $p \xrightarrow{a} q$ , and we say that  $p$  enables  $a$ . Moreover, a system may perform time transitions from a state  $p$  if: for every clock  $t \in m.\text{clocks}$  and every variable  $v \in m.\text{top.vars}$  such that  $v.\text{clock} = t$  and  $p[v] \geq 0$ , then we must have  $p[v] < \max(v.\text{dom})$ ; in which case a new state  $q$  is obtained by incrementing all the considered  $p[v] \geq 0$ . This is denoted by  $p \xrightarrow{t} q$  just like actions firing.

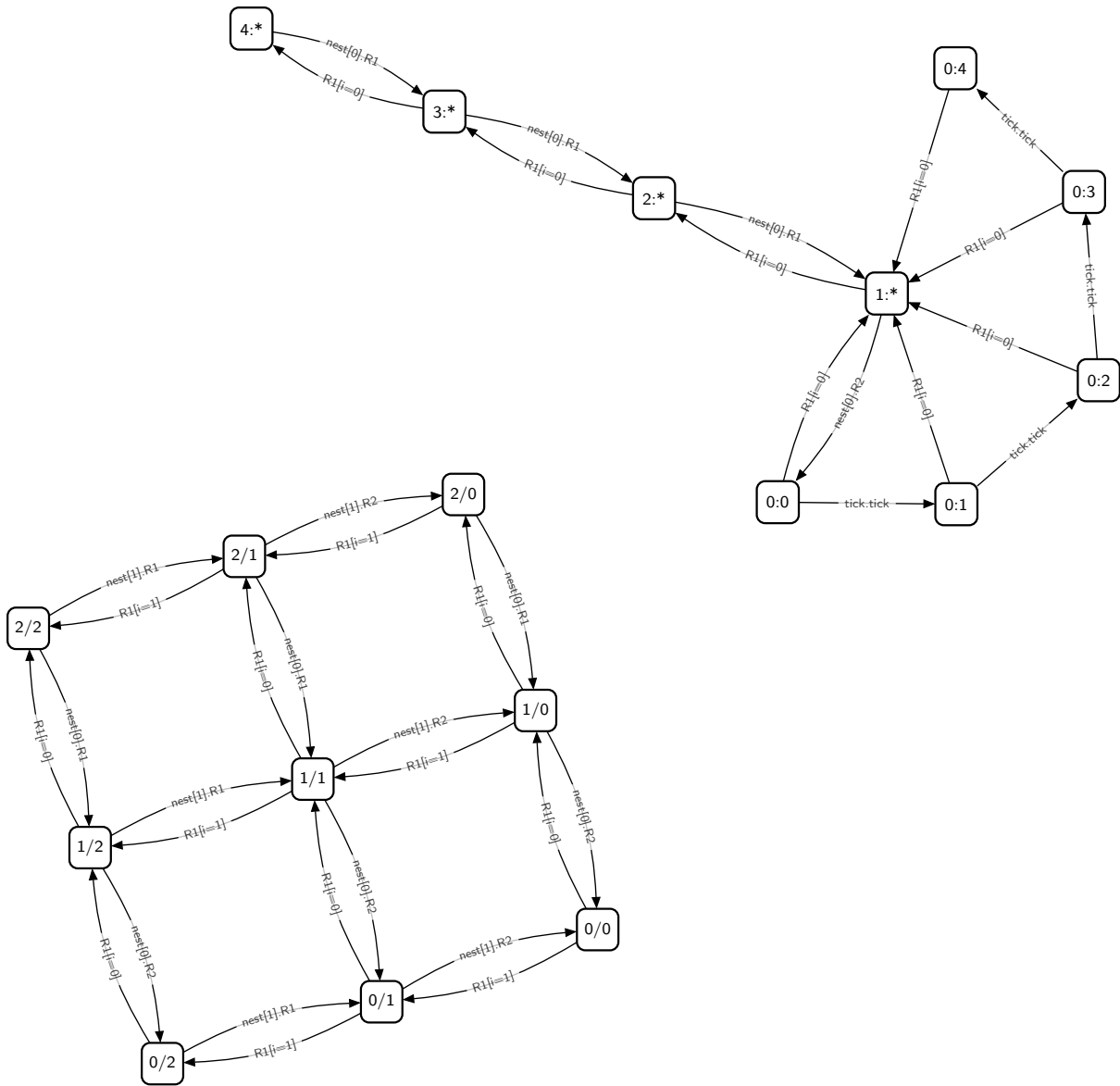
An MRR system  $m$  can generate a state-space that is defined as a graph  $(R, A)$  where  $R$  is the set of reachable states and  $A$  is the set of transitions, defined inductively by:

- $R$  includes the initial states defined in the model;
- for  $p \in R$  and a constraint  $c$  such that  $p \xrightarrow{c} q$ , then we have  $q \in R$  and  $(p, c, q) \in A$ ;
- for  $p \in R$  that enables no constraints and a rule  $r$ , if  $p \xrightarrow{r} q$ , then we have  $q \in R$  and  $(p, r, q) \in A$ ;
- for  $p \in R$  that enables no constraints and a clock  $t$ , if  $p \xrightarrow{t} q$  then we have  $q \in R$  and  $(p, t, q) \in A$ ;
- if  $p \in R$  enables no actions and no clock ticks, it is called a deadlock. ■

Figure 4 shows the state-spaces for two simpler variants of the model from Figure 2. These two examples allow seeing how the state-space grows with each parameter, but even small parameters such as `NUM=2`, `POP=2`, `MAX=2` yield a state-space with 25 reachable states that is hardly readable due to its strong connectivity and three-dimensional structure.

## 3. Petri nets semantics

In this paper, we define (colored) Petri nets as in [20, Sec. 2.2], with respect to a set of variables  $\mathbb{V}$  ranging on  $\mathbb{Z}$  and a set of expressions  $\mathbb{E} \supset \mathbb{V}$  that includes all the arithmetic and Boolean expressions



**Figure 4:** Two state-spaces of the model from Figure 2 with, distinct values of `NUM`, `POP`, and `MAX`. Top (`NUM=1`, `POP=4`, `MAX=4`): states are labelled with  $n:t$  where  $n$  is the number of animals in the nest and  $t$  is the value of `nest[0].time`. Bottom (`NUM=2`, `POP=2`, `MAX=0`): states are labelled with two integers  $n_0/n_1$  where  $n_i$  is the number of animals inside nest  $i$ . In both cases, transitions labels are rules short names with information on their origin, as used by the implementation (e.g. `nest[0].R1` is the first rule in `nest[0]`, or `R1[i=1]` is the first rule on the top-location for  $i = 1$ , and `tick.tick` is a tick of clock `tick`). The notebooks that produce this figures can be played online at <https://ecco.ibisc.univ-evry.fr/voila/voila/render/pnse-24/nests-144.ipynb?> and <https://ecco.ibisc.univ-evry.fr/voila/voila/render/pnse-24/nests-220.ipynb?>.

that MRR can express. Given  $e \in \mathbb{E}$ , we denote by  $\text{vars}(e)$  the subset of variables involved in  $e$ . A *binding* of  $e$  is a function  $\text{vars}(e) \rightarrow \mathbb{Z}$  that maps variables to values. Then, we denote by  $\beta(e)$  the evaluation of  $e$  under the binding  $\beta$ , which may be a value in  $\mathbb{Z}$  if the evaluation is possible, or an error value  $\perp$  if the evaluation is not possible for one reason or another.

**Definition 3 (Petri nets).** A Petri net is a tuple  $(S, T, \ell)$  where:

- $S$  is the finite set of places;
- $T$ , disjoint from  $S$ , is the finite set of transitions;
- $\ell$  is a labelling function such that:
  - for all  $s \in S$ ,  $\ell(s) \subset \mathbb{Z}$  is the type of  $s$ ,

- for all  $t \in T$ ,  $\ell(t) \in \mathbb{E}$  is the guard of  $t$ ;
- for all  $(x, y) \in (S \times T) \cup (T \times S)$ ,  $\ell(x, y)$  is a multiset over  $\mathbb{E}$  defining the arc from  $x$  to  $y$ , the empty multiset implementing to the absence of arc.

A marking of a Petri net  $(S, T, \ell)$  is a function  $M$  mapping each place  $s$  to a multiset over  $\ell(s)$ . A transition  $t$  is enabled at a marking  $M$  with a binding  $\beta$ , which is denoted by  $M[t, \beta]$  if:

- $M$  has enough tokens: for all  $s \in S$ ,  $\beta(\ell(s, t)) \leq M(s)$ ;
- the guard is satisfied:  $\beta(\ell(t))$  is true;
- place types are respected: for all  $s \in S$ ,  $\beta(\ell(t, s))$  is a multiset over  $\ell(s)$ .

If  $t \in T$  is enabled at  $M$  with binding  $\beta$  then it may fire yielding a new marking  $M'$  such that for all  $s \in S$ ,  $M'(s) = M(s) - \beta(\ell(s, t)) + \beta(\ell(t, s))$ . This is denoted by  $M[t, \beta]M'$ . This allows defining the reachability graph (or marking graph) of a Petri net as usual, similarly to the state-space of an MRR system, by firing transitions from the initial marking until saturation. ■

This definition can be extended in two ways:

- adding transitions priorities with two levels to implement constraints and rules:  $T$  is partitioned into  $T_c \cup T_r$  and the firing semantics is modified so that whenever  $M[c, \beta]$  for  $c \in T_c$  then every  $r \in T_r$  is inhibited and cannot be enabled at  $M$ ;
- forbidding side-loops, *i.e.* no firing  $M[t, \beta]M$  being allowed; this can be enforced at the semantics level or side-loops can be removed from the marking graph afterward.

To translate a flattened MRR system  $m$  into a Petri net, we first consider that it declares only variables  $v$  such that  $v.\text{init}$  is a singleton because Petri nets do not support having multiple initial marking. From this, we can have one reachability graph for each initial state of  $m$ , and we can take their union to get a graph that matches the state-space of  $m$ . Moreover, we assume a function  $\lfloor x \rfloor$  that translate elements of an MRR structure into Petri nets variables or expressions:

- for a variable structure  $v$ ,  $\lfloor v \rfloor \in \mathbb{V}$  is its translation into a Petri net variable;
- for an MRR expression  $e$  (**COND** or **EXPR**),  $\lfloor e \rfloor \in \mathbb{E}$  is a Petri net expression that implements  $e$ ;
- for a clock  $k \in \mathbb{I}$ ,  $\lfloor k \rfloor \in \mathbb{V}$  is a Petri net variable that cannot be obtained as  $\lfloor v \rfloor$  for any variable  $v$ .

For  $i, t, e \in \mathbb{E}$ , we shall use ternary if-then-else expressions as  $\llbracket i ? t : e \rrbracket$  that evaluates to  $t$  if  $i$  is true and to  $e$  otherwise.

**Definition 4 (Petri nets semantics of MRR).** Let  $m$  be a flattened MRR system with a unique initial state. Its Petri net semantics is a Petri net  $(S, T, \ell)$ , with  $T$  partitioned into  $T_c \cup T_r$ , and marked with  $M$  such that:

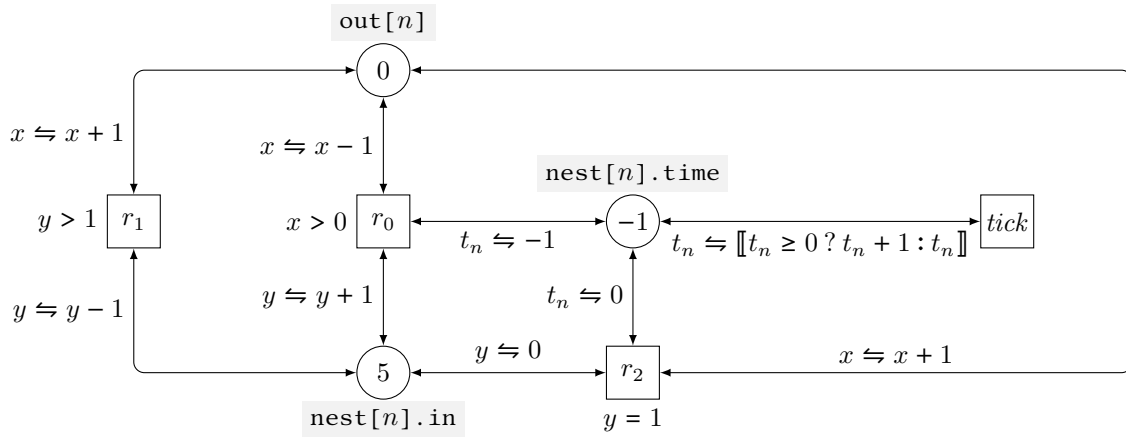
- $S = \{s_v \mid v \in m.\text{top.vars}\}$  with
  - $\ell(s_v) = v.\text{dom}$ , and
  - $M(s_v) = v.\text{init}$ ;
- $T_c = \{t_c \mid c \in m.\text{top.cons}\}$  and  $T_r = \{t_r \mid r \in m.\text{top.rules}\} \cup \{t_k \mid k \in m.\text{clocks}\}$  with
  - for all  $a \in m.\text{top.cons} \cup m.\text{top.rules}$ ,  $\ell(t_a) = \bigwedge_{e \in a.\text{left}} \lfloor e \rfloor$ ,
  - for all  $k \in m.\text{clocks}$ ,  $\ell(t_k)$  is true;
- for all  $t_a \in T$ , for all variable  $v$  involved in  $a$ ,  $\ell(s_v, t_a) = \{\lfloor v \rfloor\}$  and
  - $\ell(t_a, s_v) = \{\lfloor g.\text{value} \rfloor\}$  if  $v.\text{name} = g.\text{target}$  for some  $g \in a.\text{right}$ ,
  - $\ell(t_a, s_v) = \{\lfloor v \rfloor\}$ , otherwise;
- for every  $k \in m.\text{clock}$  and every variable  $v$  such that  $v.\text{clock} = k$ :
  - $\ell(s_v, t_k) = \{\lfloor k \rfloor\}$ , and

- $\ell(t_k, s_v) = \{\llbracket [k] \geq 0 ? [k] + 1 : [k] \rrbracket\}$ ;
- every  $\ell(x, y)$  not listed above is the empty multiset (no arc). ■

As an example, Figure 5 shows the semantics of the MRR model from Figure 2.

Given how direct is the translation, it is easy to check that an MRR system and its Petri net semantics are strongly equivalent.

**Claim 1.** *Let  $m$  be a flattened MRR system with one initial state, and  $N$  be its Petri nets semantics. Then, the state-space of  $m$  is isomorphic to the marking graph on  $N$ .* □



**Figure 5:** The semantics for one nest  $n$  in the model from Figure 2. The places are labelled by the name of the variables and their types have been omitted for readability. Transition  $r_0$  corresponds to the rule from line 29 for  $\mathbf{i} = n$ ; transition  $r_1$  corresponds to the rule from line 23; transition  $r_2$  corresponds to the rule from line 25; transition  $tick$  implements clock `tick`. All the arcs are bidirectional and labelled with pairs  $c \rightleftharpoons p$  where  $c$  is the token consumed and  $p$  the token produced, which corresponds to one arc labelled by  $\{c\}$  from the place to the transition, and one opposite arc labelled by  $\{p\}$ . More nests are obtained by duplicating this net, except for transition  $tick$  that is shared throughout the whole semantics.

## 4. Implementation

EDEN is equipped with the tool `ecco` [8, 7] that consists of a Python library intended to be used within Jupyter notebooks [21]. This environment provides the interactivity needed for the incremental analysis of models [7]. Working on a model with `ecco` supposes to load it and to explore its state-space by building and refining *component graphs*. These are graphs whose nodes (components) are sets of states linked by the transitions allowing to reach one component from another. Components can be checked against properties (CTL or `ecco` specific languages), and even split into the states that validate a property and those that do not. Doing so, users keep at hand a graph that provides a representation of the main dynamics of the system with respect to properties of interest [7, 1, 18]. This graph can be kept always readable and can be precisely tuned to progressively answer complex questions that could not necessarily be formulated from the beginning.

The MRR language has been implemented within `ecco` [8, 7], which yielded surprisingly few changes to its interface. Essentially, as MRR is a generalization of RR, the main changes are in the displays: Boolean variables are still displayed as `A+` or `A-`, but other variables can be displayed as `B=0`, `B=1`, `B=2`, ... More generally, the interface has been streamlined and simplified, with more consistency across the various methods provided to users. Under the hood, the symbolic engine based on `libddd` and `libits` [22, 23] has been rewritten to support multivalued variables. This resulted in a tool that is both very familiar and renewed, which led to reimplement the support of RR in `ecco` on the top of MRR: if loaded with a `.rr` file, `ecco` now treats it as a `.mrr` file after a light preprocessing in order to

accommodate the syntactic differences (in `RR`, variables may be distributed among arbitrarily chosen sections, while in `MRR` they are all defined within section `variables:`). Legacy `RR` support is still available by naming files with a `.lrr` extension.

#### 4.1. Example

To illustrate how `ecco` can be used with an `MRR` model, we present a new version of the termites colony model originally presented in [4], extended by considering that the termites may build several nests (we also removed two variables that were never assigned). The new model is expressed with `MRR` in Figure 6. It is organized as follows. The top location hosts the interactions between termites and their ants competitors. At the beginning termites reproductives are present, as well as ants. Reproductives may start building the first nest, which is modelled as rule `R1` that let them “enter” into location `nest[0]`, where the rules are exactly those from [4] apart from the interactions with the ants. The other rules in the top location are:

- `R2` : one nest is fully populated (*i.e.* it has reproductives, workers, and soldiers) and the colony spawns into another `nest` location that is currently empty;
- `R3` : ants can kill the termites from all the nests when there are no soldiers;
- `R4` : ants can be killed if there are soldiers in all the nests.

The two last rules are directly adapted from [4], only the first one is really new. Note that `R3` could be more specific using `for any i` which would allow ants to kill the termites in only one nest that would have no soldiers. Here, we have chosen a rule that is closer to the original model and corresponds to considering that the presence of soldiers in some nests is enough to refrain the ants from invading the region where nests are, so that they cannot kill the termites.

A Jupyter session for this illustration is displayed in Figure 7. To analyze this model, we first load it into `ecco` (`In[1]`). Note that we use extension `.mrr` so that `ecco` knows how to interpret the model. After loading our model, we can build a component graph by calling object `model()` as a function. Its arguments specify successive splits to perform onto the components:

- starting from a unique component that holds all the reachable states, the first split separates the initial states from the rest, which yields a graph with two components (this first property is called in `ecco` a *topological property*);
- this graph is then split w.r.t. the basins of attraction of the deadlocks, *i.e.* the states that necessarily lead to deadlocks, which is specified with a formula passed as a string (this syntax is `ecco` specific *functional properties*);
- then, a split is requested w.r.t. the presence/absence of ant competitors (this is actually a very simple CTL property);
- and finally, a last split extract the convex hulls of strongly-connected components (which is also a topological property).

The resulting component graph is saved into variable `g` and then displayed.

It has 6 components, within `ecco`, we can click them and explore their characteristics, which shows that #1 has a unique state that is the initial state (which is visible from tag  $\blacktriangleright$  that tells us this component is exactly the set of initial states). From #1, firing rule `R1` lead to #10, from which the termites may be killed either by the ants (rule `R3`) or by failing to build a nest (rule `nest[0].R11`). In both cases, this leads to #7 that is tagged by  $\square$  which means it contains deadlocks. Another path from #10 is to fire `nest[0].R5` leading to #9 where the system may stay forever. Indeed, tag  $\circ$  tells us that this component is the convex hull of some SCCs. From #9, termites may die (rules `nest[].R11`) or be killed by ants (rule `R3`). From #7 and #9, termites may kill ants (rule `R4`) which is surprising when starting from #7 because it is reached when termites die. But this can be explained looking at rule `nest[].R11` where only reproductives and workers die, so that there may still be soldiers in the nests. They will die later on because of rule `nest[].R9` but before they have a chance to kill the ants. Component #6 is tagged with  $\bullet$ , which tells us that it is a SCC.

```

1  #ifndef NUM
2  #define NUM 2
3  #endif
4
5  variables:
6      Rp+: reproductives in the wild
7      Ac+: ant competitors
8
9  location nest[NUM]:
10     variables:
11         Rp-: reproductives in the nest
12         Wk-: workers
13         Sd-: soldiers
14         Te-: termitomyces
15         Ec-: egg chambers
16         Fg-: fungal gardens
17         Md-: mound
18         Wd-: wood
19         Ai-: air of the nest
20
21     rules:
22         Wk+, Te+ >> Wd-, Ai-           # nest[].R1
23         Fg- >> Te-                       # nest[].R2
24         Wk+ >> Wd+, Te+, Fg+, Ec+, Md+  # nest[].R3
25         Wd- >> Wk-, Te-                 # nest[].R4
26         Rp+ >> Ec+                       # nest[].R5
27         Rp+, Ec+ >> Wk+                 # nest[].R6
28         Wk+, Wd+ >> Sd+, Rp+           # nest[].R7
29         Md+ >> Ai+                       # nest[].R8
30         Wk- >> Fg-, Sd-                 # nest[].R9
31         Wk-, Rp- >> Ec-                 # nest[].R10
32         Ai- >> Rp-, Wk-, Te-           # nest[].R11
33
34     rules:
35         # R1: reproductives enter the first nest
36         Rp+ >> Rp-, Rp@nest[0]+
37         # R2: if a nest i is fully populated, reproductives may spawn to another one j
38         Rp@nest[i]+, Wk@nest[i]+, Sd@nest[i]+, Rp@nest[j]- >> Rp@nest[j]+ for any i, j
39         # R3: if there are no soldiers in all nests, ants may kill all the termites
40         Ac+, Sd@nest[i]- >> Wk@nest[i]-, Rp@nest[i]- for all i
41         # R4: if there are soldiers in all nests, ants may be killed
42         Sd@nest[i]+ >> Ac- for all i

```

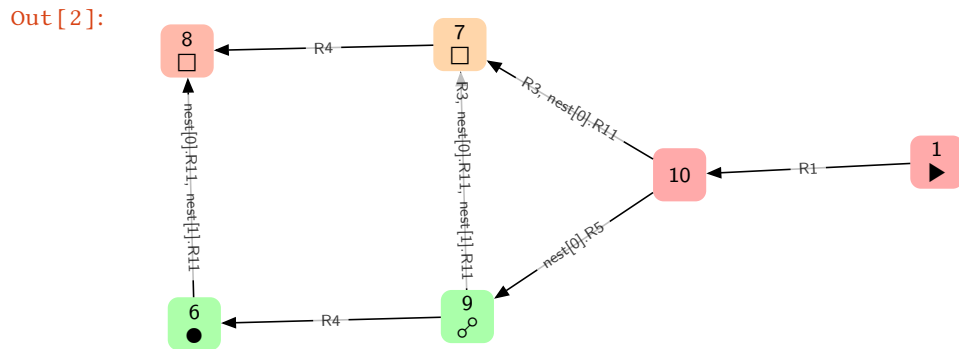
**Figure 6:** MRR modelling of the new termites model.

Cell `In[3]` allows printing the size of the state space by summing the sizes of the components in `g`, which is 11,842 states.

Finally, to clarify the dynamics related to the colonizing of new nests, cell `In[4]` splits the components with respect to the presence or absence of reproductive in each nest. Python syntax `g.split(*(...))` calls `g.split` by passing it the items resulting from comprehension `...` as individual parameters. In the comprehension, we build only the two formulas `"'nest[0].Rp'"` and `"'nest[1].Rp'"`, and using a comprehension with string substitution is only useful to demonstrate how this approach scales for more nests. These two formulas are strings (hence `"..."` quotes) with inside `'...'`-quoted atoms as the CTL syntax used in `ecco` requires complex atoms to be quoted. By exploring the components, we can see that #16 and #18 have `nest[0].Rp` on, while #12 and #14 have `nest[1].Rp` on, and #15 and #17 have both variables on. From #15 it is possible to fire `R3` (ants kill termites), but this is no more possible from #17 where ants have been killed. Otherwise, the dynamics between #12/#15/#16 on the one hand, and #14/#17/#18 on the other hand are similar, so we focus on the former group. The system enters #16 from #10 when the first nest starts to be constructed. Then it may oscillate between

```
In[1]: %run -m ecco termites.mrr
```

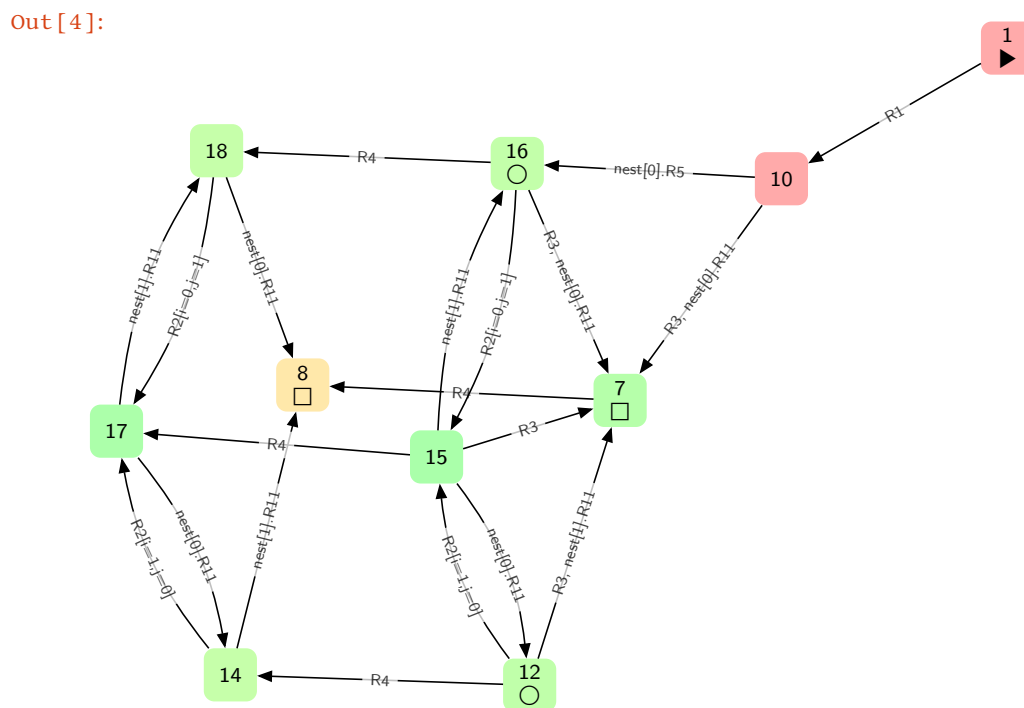
```
In[2]: g = model(topo.INIT, "basin(DEAD)", "Ac", topo.HULL)
g
```



```
In[3]: sum(len(c) for c in g)
```

Out[3]: 11842

```
In[4]: g.split(*(f"nest[{i}].Rp" for i in range(2)))
```



**Figure 7:** Jupyter session for the termites example. Python inputs are numbered with prompts like `In[...]`, and their output, if any, are numbered with corresponding prompts `Out[...]`. For the sake of quality, the inputs and outputs are not screenshots but exports from Jupyter, with no editing. This notebook can be played online at <https://ecco.ibisc.univ-evry.fr/voila/voila/render/pnse-24/termites.ipynb?>



#12/#15/#16 depending on how termites in one nest die from the lack of fresh air ( `nest[ ].R11` ) or termites in one nest spawn to the other one ( `R2` instantiated with `i=0, j=1` or `i=1, j=0` ). Tag `o` on components #12 and #16 tells us that they contain some SCCs but they are not SCCs themselves.

We shall not push further this analysis because it is already sufficient to see that the spirit and principles of `ecco` as presented in [7] are preserved: analysis is still based on component graphs that can be refined through incremental splits, which allows building readable representations of large state-spaces, targeted towards specific questions, that could not be easily obtained without this interactive environment.

## 5. Related works and discussion

The `RR` modelling language has been developed on the basis of a theory of social-ecological systems requiring to grasp their cores and inner structures. The `RR` syntax is summarizing the interaction network of each system which we need to compute its qualitative and coarse-grain dynamics. Such a long term system dynamics is conveniently modeled by a qualitative and abstract representation of the system [24, 4, 6]. `MRR` is thus not proposed as a replacement of `RR`, but really as an extension, and the language has been indeed designed to this aim. Extending a Boolean formalism in the three directions we proposed (multivalued variables, time, and space) is, to the best of our knowledge, very original. It is more usual to proceed one direction at a time, and we consider them separately in the following.

Extending a Boolean formalism to support multivalued variables is classical, for instance this was made in [25] for Boolean regulatory networks. Equipping such extended formalisms with a colored Petri semantics is also classical, for instance, the semantics proposed in [26] for regulatory networks is similar to ours: it defines one place for each variable and one transition for each function. The main difference comes from the underlying formalisms: while regulatory networks have only one function to compute the next state of one variable, we may have multiple actions changing multiple variable at a time. Similar translations may be found in many works, *e.g.* [27, 28]. Using this translation scheme allows then relying on a translation from colored Petri nets to standard Petri nets (often called unfolding, which is ambiguous w.r.t. McMillan’s unfolding [29]). Another classical translation scheme is to use standard Petri nets and pairs of complementary places to implement the variables, see *e.g.* [30]. Both techniques solve the problem to exactly check the value of a variable, which reduces to zero-test. In [7], we solved this problem using inhibitor- and reset-arcs in a compact semantics, or using pairs of complementary places in an alternative equivalent semantics [11]. From this perspective, the Petri nets semantics of `MRR` is not conservative w.r.t. to the Petri nets semantics of `RR`.

Extending an untimed formalism to support time is also classical. In this work we have chosen causal time and our extension is very similar to what was made in [13] to extend the Petri Box Calculus [31] with causal time. The main difference, apart from the formalisms that are very dissimilar, is that `MRR` allows the use of several clocks. There are of course lots of formalisms extended with continuous time, including Petri nets. We felt more relevant to propose a discrete timing model that is consistent with the abstract perspective promoted in `EDEN` [7, 9]. Considering we have introduced multivalued variables, the causal time approach then almost comes “for free”, and has a very light impact on the Petri nets semantics (we just need to add one transition per clock). However, we know that introducing causal time yields extra combinatorial explosion in the state-spaces. Indeed, for each clocked variable in a range  $[0, d]$ , every intermediate count  $0 \leq c \leq d$  may be enumerated in the state-space. If several such variables are considered, these enumerations will combine and yield even more combinatorics. `ecco` makes use of a symbolic representation of the components based on data decision diagrams [32], which should fight efficiently this combinatorial explosion. Moreover, we already observed on `RR` modelling that combinatorial explosion is often a strong incentive towards abstraction when the state-space of Boolean models with too many variables cannot be computed. Having to deal with a limit of what can be computed leads to think more thoroughly to what should be included or not in the model, and how available knowledge should be represented or abstracted away. So, we believe that this will be also the case for timed models, and that the extra combinatorial explosion they bear will lead to a qualitative

approach of time representation, where durations are more used as orders of magnitude for the relative speeds of processes, rather than actual measures of time passing.

Extending a formalism with a representation of space is probably much less classical than with multivalued variables or time, and spatial information is usually built-in from the beginning as a prominent feature of a formalism. (See [33] for a survey of space-enabled formalisms in systems biology.) However, this was made in [34] to extend Boolean networks with an abstract notion of space (modelling the topology rather than the geometry). This was pushed further in [35] to allow the dynamicity of the topology, using algebraic representations of it. Inspired from this approach, a first attempt to introduce space in RR systems was performed, leading to a language featuring a complex representation of space, with even more complex queries in order to access the variables distributed into distinct locations [36]. This language was transpiled to RR, allowing to analyze the modelled systems in ecco. This implementation has not been publicly released because we felt it is far too complex to be adopted by ecologists. Such a complexity however, does not exist with wide-spread formalisms as membrane systems [14, 15]. The main difference is that, when a complex representation of spaces allows to directly access information distributed in distant locations, simpler representations lead to explicitly model the routing of information through rules added to the model. This result in models that are less compact, but more explicit and easier to understand. Moreover, this lets modelers express more precisely the information flow, in ways that may not have been anticipated in the hard-wired routing provided at language-level. For instance, rule R2 in the termites model from Figure 6 (termites spawn to another nest) could have been implemented differently, e.g. with a rule inside location `nest` that would send new reproductives outside, and a more generic version of rule R1 allowing reproductives to enter any empty nest. Distinct implementations would express slightly different ways of modelling the same observed behavior, leading to similar or distinct trajectories, with similar or distinct conclusions on the studied system, which is exactly where the expertise of modelers gets expressed.

## 6. Conclusion and future works

We presented the MRR language that is an extension of the existing RR language for the formal modelling of social-ecological systems. MRR extends RR in several ways: first, it adds multivalued variables, including statically-sized arrays; then, it adds time as clock ticks incrementing counters; finally, it adds spatial organization of the model as nested locations. The syntax of MRR has been presented and illustrated with examples, and its formal semantics has been defined in terms of rules operating on a flattened version of a system (i.e. with a unique location). Then, a colored Petri nets semantics of MRR has been provided, which is consistent with the operational semantics and opens the way for Petri net based analysis. Finally, we presented the implementation of MRR in ecco tool, and illustrated its usage on a concrete example adapted from our early works on RR.

As discussed above, extending RR to MRR leads to several sources of increased combinatorial explosion. While this is interesting as an incentive to keep models abstract, it is sometimes desirable to introduce more details when more data is available about the system being modelled. Multivaluated variables are perhaps the least of our concerns because ecco is built on the top of data decision diagrams that already deal efficiently with them. Introducing spaces is not specifically a source of combinatorial explosion, but having arrays of locations is a way to quickly grow the size of a model. See for instance our termites example with two nests that has 11,842 states when the same model with a unique nest only has 110 states (and note that  $110^2 = 12,100 \approx 11,842$  which is consistent with the fact that two nests evolve largely independently of one another). One future direction will be to consider hierarchical data decision diagrams, called set decision diagrams, which are known to be an efficient way of factoring the states of similar sub-systems into shared sub-diagrams [37].

Clocked variables introduced by the causal time approach are however more specific than multivalued variables. For instance, in our first example from Figure 2, the intermediate counts between 0 and `MAX-1` do not constrain the system. Only when one `nest[].time` reaches `MAX` the system is constrained and animals are forced to go back to the nest before time can pass again. This is well known in timed systems

where only the constants that are compared to time measures are meaningful, which led to symbolic representations of time-states such as zones [38], bounded difference matrices and variants [39], among other approaches. A specific approach was developed in the case of Petri nets with causal time, using dual states representation with the marking on the one hand, and clock counters on the other hand, which allowed to produce aggregated state-spaces [40]. Adapting one or another of these methods to MRR will be an interesting challenge as it may lead to combine two symbolic representation of states: decision diagrams for data, and potentially another representation for time.

Analyzing RR models is also possible using Petri nets unfoldings *à la* McMillan as in [41, 42], which allows for causal analysis that is hard to perform using state-oriented component graphs. This was actually a strong motivation for proposing a Petri net semantics for MRR, and future work will be dedicated to explore how the semantics we proposed, based on colored Petri nets, will be usable to perform unfolding-based analysis.

Last but not least, MRR has been created with the aim it will be used by ecologists and other environmental scientists (*e.g.* sociologists or economists), to conduct studies on social-ecological systems, just like RR has been already, either from a theoretical or from a practical perspective. Next steps in this direction will be to work with ecologists in order to help them become familiar with this new language and the changes it led to make in *ecco*. In this perspective, having MRR be an extension of RR will certainly help and allow a progressive transition from one language to another.

## Acknowledgments

We warmly thank Sergiu Ivanov for our fruitful discussions about membrane systems formalisms, which helped a lot to design a suitable representation of space in MRR.

## References

- [1] M. Cosme, C. Hély, F. Pommereau, P. Pasquariello, C. Tiberi, A. Treydte, C. Gaucherel, Qualitative modeling for bridging expert-knowledge and social-ecological dynamics of an east African savanna, *Land* 11 (2022). doi:10.3390/land11010042.
- [2] C. Di Giusto, C. Gaucherel, H. Klaudel, F. Pommereau, Analysis of discrete models for ecosystem ecology, in: *Biomedical Engineering Systems and Technologies*, Springer, 2020. doi:10.1007/978-3-030-46970-2\_12.
- [3] C. Gaucherel, F. Pommereau, C. Hély, Understanding ecosystem complexity via application of a process-based state space rather than a potential surface, *Complexity* 2020 (2020). doi:doi.org/10.1155/2020/7163920.
- [4] C. Gaucherel, F. Pommereau, Using discrete systems to exhaustively characterize the dynamics of an integrated ecosystem, *Methods in Ecology and Evolution* 10 (2019). doi:10.1111/2041-210X.13242.
- [5] Z. Mao, J. Centanni, F. Pommereau, A. Stokes, C. Gaucherel, Maintaining biodiversity promotes the multifunctionality of social-ecological systems: holistic modelling of a mountain system, *Ecosystem Services* 47 (2021). doi:10.1016/j.ecoser.2020.101220.
- [6] C. Gaucherel, M. Cosme, C. Noûs, F. Pommereau, A single changing hypernetwork to represent (social-)ecological dynamics, *bioRxiv* (2024). doi:10.1101/2023.10.30.564699, peer-reviewed and recommended by [43].
- [7] F. Pommereau, C. Thomas, C. Gaucherel, EDEN framework for interactive analysis of ecosystems models, in: *Proc. of PNSE'22*, volume 3170, *CEUR Workshop Proceedings*, 2022. URL: <https://ceur-ws.org/Vol-3170/#paper7>.
- [8] F. Pommereau, *ecco*, tools for the formal modelling and analysis of ecosystems, 2024. URL: <http://ecco.ibisc.univ-evry.fr>.
- [9] M. Cosme, C. Thomas, C. Gaucherel, On the history of ecosystem dynamical modeling: The rise and promises of qualitative models, *Entropy* 25 (2023). doi:10.3390/e25111526.

- [10] P. Salles, B. Bredeweg, Modelling population and community dynamics with qualitative reasoning, *Ecological Modelling* 195 (2006). doi:10.1016/j.ecolmodel.2005.11.014.
- [11] F. Pommereau, C. Thomas, C. Gaucherel, Petri nets semantics of reaction rules (RR), in: Proc. of PETRINETs'22, Springer, 2022. doi:10.1007/978-3-031-06653-5\_10.
- [12] C. Bui Thanh, H. Klaudel, F. Pommereau, Petri nets with causal time for system verification, *Electronic Notes in Theoretical Computer Science* 68 (2003). doi:10.1016/S1571-0661(04)80521-8, proc of MTCS'02.
- [13] F. Pommereau, Causal time calculus, in: Proc. of FORMATS'04, Springer, 2004. doi:10.1007/978-3-540-40903-8\_21.
- [14] G. Păun, G. Rozenberg, A guide to membrane computing, *Theoretical Computer Science* 287 (2002). doi:10.1016/S0304-3975(02)00136-6.
- [15] A. Alhazov, R. Freund, S. Ivanov, Polymorphic P systems: a survey, *Bulletin of the International Membrane Computing Society* (2016). URL: <https://idus.us.es/bitstream/handle/11441/127673/BullDec2016.pdf?sequence=1#page=79>.
- [16] M. Cosme, F. Pommereau, C. Gaucherel, Farm trajectories in the South Sudanian zone of Burkina Faso: How to achieve a persistent crop-livestock integration?, SSRN, in review (2024). doi:10.2139/ssrn.4635820.
- [17] F. Pommereau, Petri nets as executable specifications of high-level timed parallel systems, in: Proc. of ICCS'04, Springer, 2004. doi:10.1007/978-3-540-24688-6\_44.
- [18] C. Thomas, M. Cosme, C. Gaucherel, F. Pommereau, Model-checking ecological state-transition graphs, *PLOS Computational Biology* 18 (2022). doi:10.1371/journal.pcbi.1009657.
- [19] R. Durchholz, Causality, time, and deadlines, *Data & Knowledge Engineering* 6 (1991). doi:10.1016/0169-023X(91)90024-R.
- [20] F. Pommereau, Algebras of coloured Petri nets, Habilitation thesis, UPEC, 2009. doi:10.5281/zenodo.3478451.
- [21] J. M. Perkel, Why Jupyter is data scientists' computational notebook of choice, *Nature* 563 (2018). doi:10.1038/d41586-018-07196-1.
- [22] Y. Thierry-Mieg, From symbolic verification to domain specific languages, Habilitation thesis, UPMC, 2016. URL: <http://pages.lip6.fr/Yann.Thierry-Mieg/hdr-ytm.pdf>.
- [23] Y. Thierry-Mieg, Homepage of ITS-tools, <http://lip6.github.io/ITSTools-web>, 2023.
- [24] C. Gaucherel, *The Languages of Nature: When nature writes to itself*, Lulu éditions, Paris, 2019. ISBN: 9780244214982.
- [25] G. Didier, E. Remy, C. Chaouiya, Mapping multivalued onto Boolean dynamics, *Journal of Theoretical Biology* 270 (2011). doi:10.1016/j.jtbi.2010.09.017.
- [26] R. Banks, L. J. Steggle, A high-level Petri net framework for genetic regulatory networks, *Journal of Integrative Bioinformatics* 4 (2007). doi:10.1515/jib-2007-60.
- [27] J.-P. Comet, H. Klaudel, S. Liauzu, Modeling multi-valued genetic regulatory networks using high-level Petri nets, in: Proc. of PETRINETs'05, Springer, 2005. doi:10.1007/11494744\_13.
- [28] C. Chaouiya, E. Remy, D. Thieffry, Qualitative Petri net modelling of genetic networks, in: *Transactions on Computational Systems Biology VI*, Springer, 2006. doi:10.1007/11880646\_5.
- [29] K. L. McMillan, D. K. Probst, A technique of state space search based on unfolding, *Formal methods in system design* 6 (1995). doi:10.1007/BF01384314.
- [30] C. Chaouiya, E. Remy, D. Thieffry, Petri net modelling of biological regulatory networks, *Journal of Discrete Algorithms* 6 (2008). doi:10.1016/j.jda.2007.06.003.
- [31] E. Best, R. Devillers, M. Koutny, *Petri net algebra*, Springer, 2001. doi:10.1007/978-3-662-04457-5.
- [32] J.-M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, P.-A. Wacrenier, Data decision diagrams for Petri net analysis, in: Proc. of PETRINETs'02, Springer, 2002. doi:10.1007/3-540-48068-4\_8.
- [33] K. Takahashi, S. N. V. Arjunan, M. Tomita, Space in systems biology of signaling pathways – towards intracellular molecular crowding in silico, *FEBS Letters* 579 (2005). doi:10.1016/j.febslet.2005.01.072.

- [34] C. Chaouiya, H. Klaudel, F. Pommereau, A modular, qualitative modeling of regulatory networks using Petri nets, Springer, 2011. doi:10.1007/978-1-84996-474-6\_12.
- [35] J.-L. Giavitto, H. Klaudel, F. Pommereau, Integrated regulatory networks (irns): Spatially organized biochemical modules, *Theoretical Computer Science* 431 (2012). doi:10.1016/j.tcs.2011.12.054.
- [36] M. Leloup, M. Cosme, F. Pommereau, C. Gaucherel, A spatially explicit scenario analysis of food security in a West-African socio-ecological system under climate change, in review (2024).
- [37] Y. Thierry-Mieg, D. Poitrenaud, A. Hamez, F. Kordon, Hierarchical set decision diagrams and regular models, in: *Proc. of TACAS'09*, Springer, 2009. doi:10.1007/978-3-642-00768-2\_1.
- [38] P. Bouyer, P. Gastin, F. Herbreteau, O. Sankur, B. Srivathsan, Zone-based verification of timed automata: extrapolations, simulations and what next?, in: *Proc. of FORMATS'22*, Springer, 2022. doi:10.1007/978-3-031-15839-1\_2.
- [39] K. Klai, N. Aber, L. Petrucci, A new approach to abstract reachability state space of time Petri nets, in: *Proc. of TIME'13*, IEEE, 2013. doi:10.1109/TIME.2013.22.
- [40] F. Pommereau, R. Devillers, H. Klaudel, Efficient reachability graph representation of Petri nets with unbounded counters, *Electronic Notes in Theoretical Computer Science* 239 (2009). doi:10.1016/j.entcs.2009.05.034, *proc. of INFINITY'2009*.
- [41] G. K. Aguirre-Samboní, C. Gaucherel, S. Haar, F. Pommereau, Reset Petri net unfolding semantics for ecosystem hypergraphs, in: *Proc. of PNSE'22*, volume 3170, *CEUR Workshop Proceedings*, 2022. URL: <https://ceur-ws.org/Vol-3170/#poster2>.
- [42] G. K. Aguirre-Samboní, *Ecosystem Causal Analysis using Petri Net Unfoldings*, PhD thesis, Univ. Paris-Saclay, 2023.
- [43] C. Sueur, The dawn of dynamic hypergraph modelling in ecology, 2024. doi:10.24072/pci.networksci.100148.