EDEN Framework for Interactive Analysis of Ecosystems Models

Franck Pommereau[1], Colin Thomas[1,2] and Cédric Gaucherel[2]

[1]IBISC, Univ. Évry, Univ. Paris-Saclay, 91020 Évry-Courcouronne, France
[2]AMAP-INRA, CIRAD, CNRS, IRD, Univ. Montpellier, 34398 Montpellier, France

Abstract
Understanding ecosystems is crucial, in particular to take conservation actions. One way to do so is formal modelling and analysis. In this paper, we present the EDEN (*Ecological Discrete-Event Networks*) framework that provides ecologists with discrete modelling languages and dedicated analysis tools. These tools are based on well-known techniques in computer science, like symbolic state-spaces or model-checking, but used in quite a different way. Indeed, most formal analysis techniques provide yes/no answers to well-defined questions, possibly with a witness execution, which is good to assess whether a system exhibits or not a given property. However, most questions in ecology are not stated as "Does the system have such behaviour?" but rather as "Why does the system sometimes has such behaviour and how can we prevent it from happening?" Moreover, these questions are often hard to express formally. With EDEN, we propose an exploratory way to build progressively a representation of this behaviour that is suitable to answer such questions. The question itself being formalised on the way, together with the model exploration. This approach is based on a hybrid representation of the state-space that can be incrementally split into a graph of *components* (symbolic sets of states) linked by transitions. The goal is thus to provide the users with a human-readable representation of the state-space that can be fine-tuned with respect to the questions of interest, resulting in an object that constitutes by itself the expected explanation. While EDEN is rather specific to ecology, we advocate that its analysis method and tools could be beneficial for other domains.

Keywords
formal modelling, interactive analysis, ecosystems

1. Introduction

The Earth is experiencing its 6th mass extinction: experts estimate that 20% to 50% of living species may go extinct during the 21st century [1]. In this context, understanding ecosystems and their dynamics is crucial to be able to take conservation actions.

One way for this understanding, and in particular the discovery of the rules that govern ecosystems dynamics, is to resort to modelling and formal analysis. For this purpose, formal modelling approaches have been proposed [2, 3], yielding *discrete models* whose dynamics are represented as *labelled transition systems* (LTS) [4, Sec.2.1]. With such models, ecologists first want to extract the main dynamics of ecosystems and their drivers. When it comes to modelling and analysing ecosystems, some of their specific features have to be taken into account:

PNSE'22, International Workshop on Petri Nets and Software Engineering, Bergen, Norway, 2022
franck.pommereau@univ-evry.fr (F. Pommereau)
0000-0002-9959-3699 (F. Pommereau); 0000-0002-7650-784X (C. Thomas); 0000-0002-4521-8914 (C. Gaucherel)

- *lack of specification:* unlike a human-made system, an ecosystem has of course no specification, but even compared to a biological system of which many copies exist, most ecosystems are unique and nobody can tell how they are expected to behave;
- *lack of data:* compared to biological systems in particular, observations of ecosystems are much sparser, and at the same time, they are much more complex systems;
- *wrong behaviours are expected:* undesired behaviours are expected to possibly happen in ecosystems, for example, the extinction of some species is generally always a possibility and thus has to be included in a model.

Lack of specification and data may lead to consider abstract models; in this paper we use a Boolean setting allowing to embed the unavailable details into abstract descriptions. The presence of wrong behaviours impacts analysis more than modelling. In particular, using model-checking, one can automatically assess a property, usually formalised as a temporal logic formula [5]. That is, get a yes/no result, and possibly in the latter case, one execution of the system that violates the checked property. This kind of technique is well fitted to search for bugs in computer systems. But it is quite insufficient to analyse models of systems where the properties are expected to be violated by some executions and satisfied by others. In such cases the analyst is more interested in understanding the reasons leading to either behaviour, and to understand how and why it happens, and how it may be avoided. Moreover, the properties of interest may be very difficult to formalise as unique temporal logic formulas because we often do not know in advance how the system precisely behaves. In other words, we may not have specifications of the systems that could be used to assess their behaviour. Instead, we want to extract from the model an understanding of the system's main dynamics with respect to the studied problem. For instance, we may have observations of an ecosystem showing that the vegetation evolves towards forests that are too dense to allow pastoralism, but we do not know the steps that lead to this situation and even less their causes. Thus, a general question like "What can we do to avoid this situation?" cannot even be formalised. Instead, the need is rather to explore the model through simpler questions like "How does the system react to such event?", or "How does the system behave when such species are absent?", and so on until a global understanding of the system behaviour is captured.

These observations drove the development of specific analysis methods for ecosystems presented hereby. However, we believe they could be beneficial to other domains where systems share some features with ecosystems.

This paper proposes the first presentation of the *EDEN framework* that has been developed and used for years [3, 6, 7, 8, 9, 10, 11, 12]. EDEN provides ecologists with the approach presented above and consists of various features that will be presented throughout the paper:

- a pair of *modelling languages*, one textual and one graphical, both carrying the same information, and equipped with a *formal semantics* expressed in terms of Petri nets (Section 2);
- a *hybrid representation of the state-space* as graphs of components (*i.e.* symbolic set of states) linked by transitions (Section 3);
- a way to *split incrementally the components* with respect to various properties (in particular CTL [13] formulas) to obtain a refined view of the state-space (in Section 3 as well);

- an *implementation* that can be used interactively within Jupyter notebooks [14] (Section 4).

In addition, Section 5 shows an example of splitting a component graph to draw understanding from a simple running example.

This paper focuses on an intuitive presentation of EDEN, while being precise enough to capture the details. A detailed formalisation of the textual modelling language, its Petri nets semantics, and its properties, is the topic of another paper [15]. A presentation if CTL in the context of EDEN is available in [12].

2. Modelling Languages

At the heart of EDEN lie two complementary modelling languages: the language of *reaction rules* (RR for short), and that of *ecosystemic hypernetworks* (EH for short). The former is textual while the latter is graphical, and each can be translated into the other without any loss of information. While RR is used to actually input a model, EH may be more suitable to display it, in particular to better understand the relationships between its constitutive elements.

2.1. Reaction Rules (RR)

The RR modelling language involves entities, constraints, and rules [3, 16]. Entities are the biotic and abiotic elements of an ecosystem, modelled as Boolean variables (on/off, denoted as +/-). Rules and constraints (collectively referred to as actions) define how entities values may evolve applying an effect (assignment of entities values) and depending on a guard (condition on entities values). Constraints have a higher priority than rules and are used to model cascading events; for instance if a habitat disapears, all its inhabitants must disappear too.

Figure 1 shows a simple example of an ecosystem modelled this way: a termite colony. Entities are declared on the left-hand column as a name, an initial state (“`+`” for on, “`-`” for off, or “`*`” to allow both initial values), and a textual description. For instance, we declare entity `Rp` that is initially on and models the presence of the reproductives (queen and king) in the colony. Entity `Ac` has an undefined initial state, so both `Ac+` and `Ac-` are considered as initial states of the system. Entities declarations are organised into arbitrarily chosen categories (except for “`rules`” and “`constraints`” that are reserved keywords). For instance, entity `Rp` is declared within category `inhabitants`. These categories are for information purpose and have no consequences on the semantics. Rules and constraints are listed after entities and consist of two-sides separated by “`>>`”: left-hand side is the guard, that is, the condition for the execution of the action; right-hand side is the effect, that is, the entities assignment that takes place upon execution of the action. For instance, the unique constraint in this model specifies that, if fungal gardens are destroyed, then the fungi cannot live inside it as a cascading effect. Similarly, rule `R10` of the model specifies that if ant competitors are present (`Ac+`) but there is no soldiers (`Sd-`), then workers and reproductives may be killed (thus `Wk-` and `Rp-`).

The semantics of such systems has been originally defined both in terms of operational rules, and in terms of a translation to Petri nets (presented below), both semantics being proved equivalent [3, 15]. Intuitively, the operational semantics is as follows:

```
variables:                                 rules:
  Rp+: reproductives                         Rp+ >> Ec+                        # R1
  Wk-: workers                               Rp+, Ec+ >> Wk+                   # R2
  Sd-: soldiers                              Wk+ >> Wd+, Te+, Fg+, Ec+         # R3
  Te-: termitomyces (fungi)                  Wk+, Wd+ >> Sd+, Rp+              # R4
  Ec-: egg chambers                          Wk-, Te+ >> Wd-                   # R5
  Fg-: fungal gardens                        Wk+ >> Wd-                        # R6
  Wd-: wood                                  Wd- >> Te-                        # R7
  Ac*: ant competitors                       Wk- >> Fg-, Sd-                   # R8
constraints:                                 Wk-, Rp- >> Ec-                   # R9
  Fg- >> Te-                    # C1         Ac+, Sd- >> Wk-, Rp-              # R10
                                             Sd+ >> Ac-                        # R11
                                             Te- >> Rp-, Sd-                   # R12
```

Figure 1: A toy model of a termite colony inspired from [2, 3]. Actions are named for reference using a comment at the end of lines.

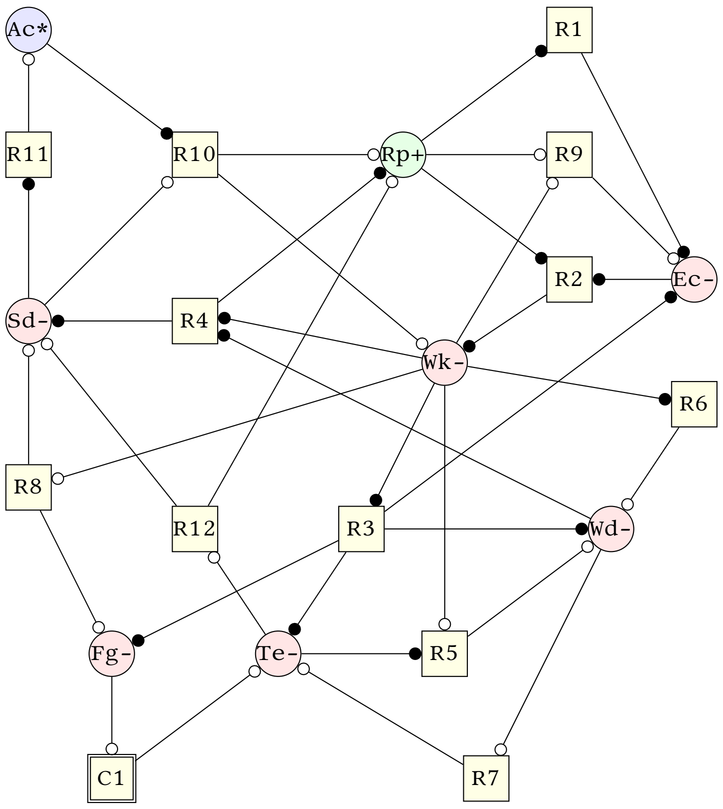


Figure 2: Translation of the RR model from Figure 1 into the equivalent EH. Action-nodes have been labelled with the name of the corresponding actions: C1 for the constraint, R1 to R12 for the rules, as used in the RR source code.

- the *initial states* are defined from the declaration of the entities, either they are initially on/off, or both values are considered (like `Ac*` in Figure 1);
- at a state, the *successors* are obtained by firing constraints, each yielding a successor state;
- if no constraint is enabled, then the successors are obtained by firing rules;
- if no rule nor constraint is enabled, then the state is a *deadlock*;
- an *action*, *i.e.* a rule or a constraint, is enabled when its guard is satisfied by the state and its effect is not already realised (*i.e.* we forbid self-loops: events must change states);
- firing an action a enabled at a state s is made by applying the effect of a onto s, yielding a new state $s' \neq s$, which is a *transition* denoted by $s \xrightarrow{a} s'$.

For instance, the model defined in Figure 1 has two initial states $\{\text{Rp}, \text{Ac}\}$ and $\{\text{Rp}\}$ (denoted by listing only the variables that are on). From both of these states, constraint `Fg- >> Te-` is not enabled because we already have `Te-`. But rule R1 is enabled because we have `Rp+` (condition) but not `Ec+` (effect). So it may fire and we have two transitions $\{\text{Rp}, \text{Ac}\} \xrightarrow{\text{R1}} \{\text{Rp}, \text{Ac}, \text{Ec}\}$ and $\{\text{Rp}\} \xrightarrow{\text{R1}} \{\text{Rp}, \text{Ec}\}$. Rule R9 is also enabled at initial state $\{\text{Rp}, \text{Ac}\}$ and its firing yields state $\{\text{Ac}\}$ that is a deadlock.

The ability to consider several initial states is useful to model control variables, whose values do not change but that control the execution of some actions. This is also useful to analyse together several scenarios, like the presence or absence of ants in the termite colony example.

A LTS is obtained from a RR model by firing constraints and rules as explained above. For example, our termite model has 42 reachable states, including 2 initial states and 3 deadlocks. However, computing such a LTS would require to implement the operational semantics. Instead, one can rely on the translation to Petri nets presented below and detailed in [15].

2.2. Ecosystemic Hypernetwork (EH)

A RR model can be equivalently represented has an *Ecosystemic Hypernetwork* (EH). This graphical notation is close to that of Petri nets and RR is tightly related to them as shown in the next section. In an EH:

- every entity is drawn as a round node labelled with the entity name and its initial state;
- every action is drawn as a square node, with a double-line border if it is a constraint;
- every action is linked to the entities it involves by edges whose ends are decorated to reflect how the action shall use each entity:
  - a black dot at the action-end of an edge means that the action expects the entity to be on, a white dot that the entity is expected to be off, no dot that it may be either on or off;
  - a black dot at the entity-end of an edge means that the action sets the entity to on, a white dot that it sets it to off, no dot that it is left unchanged;
  - an action may both read and write an entity, in which case there will be a dot at both ends of the edge.

Figure 2 shows the translation in EH of the RR model from Figure 1, and Figure 3 shows a table whose three first columns summarise this correspondence. Note that contrasting with the RR notation, EH does not convey the category nor the description of entities, but they play no role in the semantics and a graphical editor may take them into account.

Such a graphical representation is useful for ecologists as it allows to represent in a readable and non-ambiguous way relevant information about a model. In particular, EH directly displays the interactions between its entities. See *e.g.* in Figure 2 how Wk is connected to many rules and thus can be visually identified as a central component. On the other hand, Ac appears as an entity that is more external to the modelled ecosystem (which does not mean it has a marginal role). Finally, the EH allows to reason graphically on a model, which is a well-known strength of the graphical representation of Petri nets too.

2.3. Petri nets Semantics

As noted above, EH notation is inspired from Petri nets, and EH (like RR) can indeed be translated to Petri nets. For this translation, we consider two classes of Petri nets:

- *priority Petri nets* (PPN for short) which are regular place/transitions nets extended with transitions priorities: firing a transition with a higher priority is systematically preferred to firing a transition with a lower priority, the former being used to implement constraints while the latter implements rules;
- *extended Petri nets* (EPN for short) which are priority Petri nets with additional extensions:
  – *read arcs* test the presence of a token in a place without consuming it, depicted without any arrow tip;
  – *inhibitor arcs* test the absence of tokens in a place, depicted with a white dot instead of an arrow tip;
  – *reset arcs* empty a place whatever it contains, depicted with a diamond tip.

An RR or EH model can be translated to an EPN as shown by the fourth column of Figure 3. Note that, by doing so, we lose some information as distinct RR/EH actions may have the same translation. See for instance the two first rows in Figure 3: having A+ in the left-hand side of a rule and no A or A+ again in the right-hand side is semantically equivalent (A must be on before firing and remains on after firing). But the modeller may chose to explicitly add A+ at the right-hand side to carry some information (*e.g.* something is regenerated and not only preserved). Note also that the translation to EPN yields one place for each entity and one transition for each action. However, if an RR/EH model has several initial states, then we shall consider several initial markings for its EPN translation, putting one token in each place that corresponds to an entity initially on.

Then, an RR/EH model may also be translated into a PPN as show in the last column of Figure 3. Basically, the translation to PPN consists of three transformations added to the EPN semantics:

- entities are implemented as pairs of complementary places, which allows to remove inhibitor arcs;
- read arcs are implemented as side-loops (at the price of reduced concurrency if a concurrent semantics is to be considered);
- reset arcs are implemented by duplicating the transitions in order to take into account every possible marking of the places emptied by reset arcs.

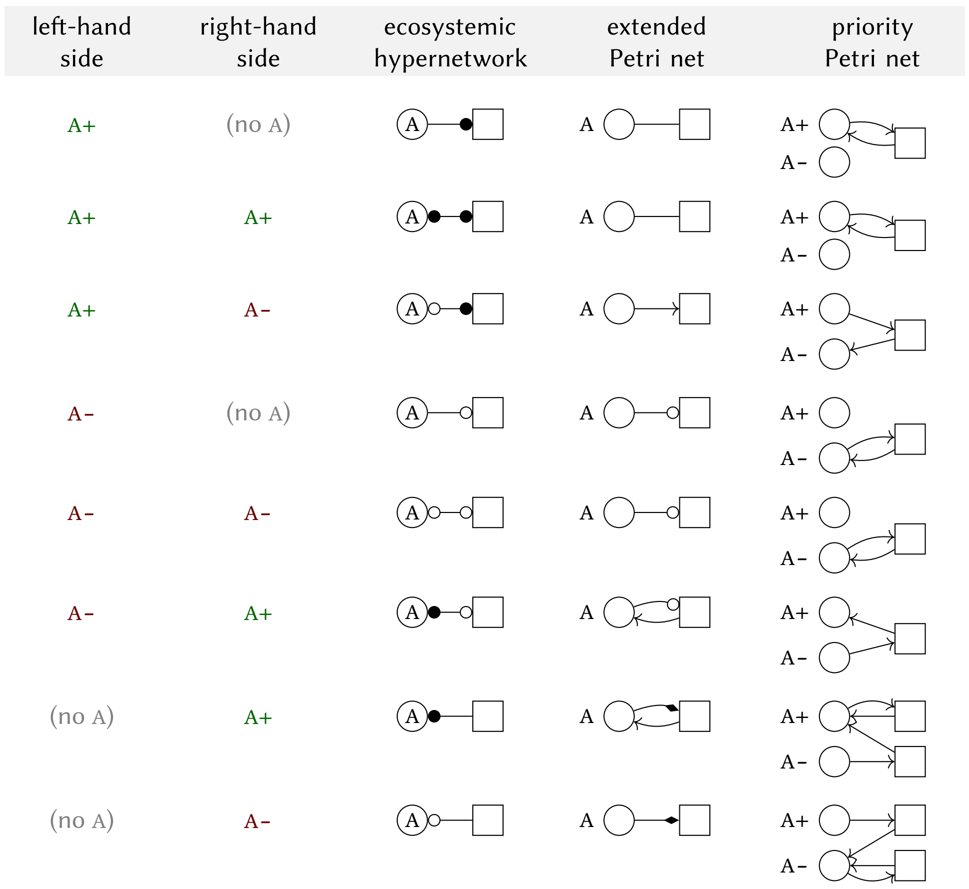


Figure 3: Columns 1—3: translation between reaction rules and ecosystemic hypernetworks, and vice-versa. Columns 4—5: translation to extended and priority Petri nets.

Then, the obtained PPN may be simplified to remove transitions that do not change the marking, which corresponds to the RR/EH semantics that forbid self-loops. Unfortunately, such a simplification is not possible on the EPN version because reset arcs allow to fire transitions with or without consuming tokens. In this case, the restriction has to be ensured at the semantics level, removing or avoiding self-loops in the marking graph for instance.

In [15], we provide a detailed definition of RR, its semantics, and its translation to PPN or EPN, and we prove that all the semantics are strongly equivalent as they generate isomorphic transition systems.

3. Component Graphs

A state-space in EDEN is represented as a *component graph* (CG for short) that is a graph whose nodes are sets of states (components) and whose edges are the transitions allowing to reach one component from another. We formalise this as a partition of the states of a LTS.

Given a RR or EH system, whose initial states form a set $\mathcal{I}$, we denote by $(\mathcal{R}, \mathcal{I}, \mathcal{A}, \rightarrow)$ its LTS semantics, where $\mathcal{R}$ is the set of reachable states, $\mathcal{I}$ is the set of initial states, $\mathcal{A}$ is the set of actions labelling transitions, and $\rightarrow$ is the reachability relation. $\mathcal{R}$, $\mathcal{A}$, and $\rightarrow$ are the smallest sets such that $\mathcal{I} \subseteq \mathcal{R}$, and, whenever $s \xrightarrow{a} s'$ for some $s \in \mathcal{R}$, we also have $s' \in \mathcal{R}$, $a \in \mathcal{A}$, and $(s, a, s') \in \rightarrow$. Note that, in contrast with the usual definition, we consider here multiple initial states. Such LTS could be obtained from the union, for all the initial markings corresponding to $\mathcal{I}$, of the marking graphs of the PPN or EPN.

Then, a component graph is defined as follows.

Definition 1. *Let $L \stackrel{\text{df}}{=} (\mathcal{R}, \mathcal{I}, \mathcal{A}, \rightarrow)$ be a labelled transition system. A* component decomposition *of L is a partition $\mathcal{C}$ of $\mathcal{R}$ and its elements are called the* components. *For $s \in \mathcal{R}$, We denote by $\langle s \rangle_{\mathcal{C}}$ the component in $\mathcal{C}$ such that $s \in \langle s \rangle_{\mathcal{C}}$, which may also be denoted by $\langle s \rangle$ when there is no ambiguity about $\mathcal{C}$. The* component graph *of L with respect to $\mathcal{C}$ is the* LTS *$L/\mathcal{C} \stackrel{\text{df}}{=} (\mathcal{C}, \mathcal{I}_{\mathcal{C}}, \mathcal{A}_{\mathcal{C}}, \rightarrow_{\mathcal{C}})$ such that:*

- $\mathcal{I}_{\mathcal{C}} \stackrel{\text{df}}{=} \{\langle i \rangle_{\mathcal{C}} \mid i \in \mathcal{I}\}$ *is the set of initial components;*
- $\rightarrow_{\mathcal{C}} \stackrel{\text{df}}{=} \{(\langle s \rangle_{\mathcal{C}}, a, \langle s' \rangle_{\mathcal{C}}) \mid s \xrightarrow{a} s' \wedge \langle s \rangle_{\mathcal{C}} \neq \langle s' \rangle_{\mathcal{C}}\}$ *is the labelled transition relation between components,* i.e. *a transition exists from one component to another iff a transition with the same label exists from one state of the first component to one state of the second component;*
- $\mathcal{A}_{\mathcal{C}} \stackrel{\text{df}}{=} \{a \in \mathcal{A} \mid \exists\, (C, a, C') \in \rightarrow_{\mathcal{C}}\}$ *is the set of visible actions,* i.e. *those actions that allow to reach one component from another.*

The main benefit of a CG is that its components may be represented symbolically. In particular, we discuss in the next Section how we use decision diagrams [17, Sec. 1] for this purpose. Doing so, a huge state-space may be represented efficiently by a small CG that can be seen as a hybrid object mixing symbolic components with explicit information about their relationship. For instance, we have been able to work with CGs encompassing billions of states dispatched over a few components.

The simplest, most symbolic, component decomposition of a LTS has a single component ($\mathcal{C} = \{\mathcal{R}\}$), and the most detailed, explicit, component decomposition has only singleton components ($\mathcal{C} = \{\{s\} \mid s \in \mathcal{R}\}$, which is equivalent to the LTS itself). Yet, we are mostly interested with something in between with sufficiently few components to remain human-readable, but with enough details to exhibit key aspects of the dynamics.

To obtain such a representation, we proceed through incremental decompositions by splitting components. In the rest of the section, we focus on several ways to split components.

3.1. Splitting with respect to topology

Considering a LTS as a graph, we can partition it into its topological components:

- the *deadlocks* are the states with no successors. They usually correspond to collapses of the ecosystem in which species have disappeared and nothing happens anymore;
- a *strongly connected component* (SCC) is a maximal set $S \subseteq \mathcal{R}$ of states such that $\forall s \neq s' \in S : s \rightarrow^{+} s'$ where $\rightarrow^{+}$ is the transitive closure of the transition relation $\rightarrow$. Note that, in contrast with the usual definition, a single state is not considered as an SCC because this would lead to large CGs with many singleton components, which contradicts our

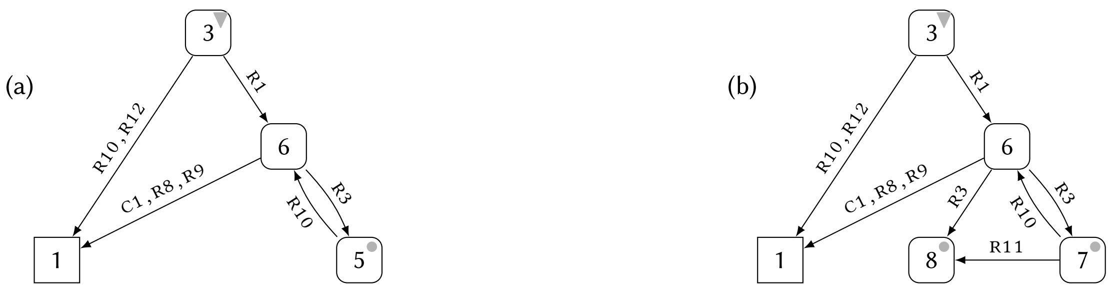


Figure 4: (a) A topological decomposition of the state-space of the termite model where component 3 is the set of initial states (hence its ▼ decoration), component 1 is the set of deadlocks (hence its square-shape), component 5 is the SCCs hull (thus it has at least a SCC, hence its ● decoration), and component 6 is the basin to components 1 and 5. **(b)** Decomposition of component 5 from (a) into components 7 and 8 with respect to either one-way rule R11, or CTL formula $\varphi_{\textit{may-die}} \overset{\text{df}}{=} \mathsf{EF}\,\neg(\mathtt{Rp} \vee \mathtt{Wk} \vee \mathtt{Sd})$.

readability goals. A SCC is an insightful structure because it represents a set of states within which the ecosystem may stay in the long term, which corresponds to stability (or multistability as systems biology calls it) of the ecosystem;

- the *convex hull of the SCCs* is the smallest set H that contains all the SCCs, and is such that for all $s \neq s' \in H$, if there is a state s'' such that $s \rightarrow^{+} s''$ and $s'' \rightarrow^{+} s$, then $s'' \in H$. Intuitively this is the union of the SCCs plus the states between them. Our experience shows that, at a first attempt, the SCC's hull makes a better component choice than the SCCs which are often too numerous to build a human-readable CG;
- the set of *initial states* may be set apart even if it is not defined with respect to the graph topology. Indeed, these states have been chosen by the modeller because it is interesting to examine the system's behaviour starting from them.

A typical topological decomposition would choose some components among SCCs or SCCs' hull, deadlocks, and initial states. Then, the remaining states would be split into the *basins* to these chosen components: given a set $\{C_1, \ldots, C_k\}$ of components, two states are in the same basin iff they allow to reach exactly the same C_i's [18]. For a cleaner presentation, the deadlocks component may be merged with the basin leading solely to it. Such a decomposition can be applied as well to an arbitrary component, as we shall use it below.

For instance, Figure 4(a) presents a topological decomposition of the state-space of the termite model. It shows in particular that rules R1 and R3 are crucial to build a living ecosystem, while R10, R12, C1, R8, and R9 on the other hand can lead it to collapse. All these latter rules correspond to one or another kind of depletion in the ecosystem, but R10 appears to play a more important role as it can take the system out of a potential long term stability (the SCC's hull 5).

3.2. Splitting with respect to one-way actions

By a static examination of a RR/EH system, it is possible to identify some variables that are always assigned the same way (*i.e.* they appear on the right-hand side of actions with always the same sign). For instance, in the termite model, entity Ac is assigned to off in rule R11 and never to on. Thus, rule R11 is one-way, *i.e.* it can never be fully undone, and components may be split into the states before the firing of R11 and those after. For example, we can obtain the CG of Figure 4(b) by splitting component 5 from Figure 4(a) with respect to rule R11.

Such one-way actions do not always exist but when they do, they are easy to discover statically and can be listed to the analyst who may chose to split accordingly.

3.3. Splitting with respect to states properties

Among temporal logics, state-based logics allow to compute sets of states that validate a property. For instance, given a LTS and a CTL formula φ, a typical symbolic model-checker computes the set of states that satisfy φ, and checks whether it includes an initial state [19]. By suppressing the last step, we obtain symbolically the set of states that satisfy φ. This allows to split the components of a CG into those states that validate φ and those that do not.

In practice, as ecologists may not be familiar with temporal logics, they can be provided with catalogues of query patterns mapping parameterised properties expressed in natural language to their translation into temporal logic formulas [12, Table 2]. For instance, pattern "It is possible to reach a state from which X states must persist forever" can be translated to $\mathsf{EF}(\mathsf{AG}\,X)$.

Splits with respect to properties are particularly interesting to study the relationship between several properties that are not necessarily temporally or logically linked. For instance, Figure 4(b) can be obtained from Figure 4(a) by splitting component 5 with respect to formula $\varphi_{\mathit{may\text{-}die}} \overset{\text{df}}{=} \mathsf{EF}\,\neg(\mathtt{Rp} \vee \mathtt{Wk} \vee \mathtt{Sd})$ that is "the system may eventually reach a state where all the termites have disappeared". This split yields two components: 7 where $\varphi_{\mathit{may\text{-}die}}$ holds, and 8 where $\varphi_{\mathit{may\text{-}die}}$ does not hold. Component 7 looks very similar to component 5: R3 is necessary to enter the component, and R10 can take the system out of it. Component 8 on the other hand includes at least one SCC and no deadlocks, and it is terminal (thus at least one of its SCCs is terminal too). It shows that rule R11 (termites kill ants), executed after R1 and R3 (termites complete the colony), allows the system to stay alive forever.

4. Implementation

We have implemented the EDEN framework as a Python [20] library intended to be used within Jupyter notebooks [14]. This implementation is called `ecco` to distinguish it from EDEN that is not only a software but a more general framework including the concepts and methods promoted in this paper. The library mainly consists of (1) a Cython [21] module that interfaces with `ITS-tools` and `libDDD` [22, 23] to provide the symbolic LTS class that is used by (2) a frontend that provides a CG class to compute and split CG objects from LTS objects.

A typical session is organised as follows:

- a RR model is loaded;
- a LTS object is built by translating a RR source file into its EPN semantics, which in turn is translated to a GAL source file [17, Sec. 5]. The GAL is then loaded by `ITS-tools` to provide the symbolic transition relations of the LTS object;
- a CG object is built on top of the LTS object by providing a set of initial states. This set is rebuilt from scratch since GAL only supports a single initial state. Then, the set of

reachable states is computed;
- new CGs are built by splitting, merging, or removing components.

Each CG object is as an immutable graph and the components are numbered consistently across all the CGs. This is just like in Figure 4 in which CG (a) and CG (b) both have components 1, 3, and 6, which guarantees they are exactly the same components (*i.e.* they hold exactly the same states from the same LTS). A CG object also comes with tabular representations of its nodes and edges stored as Pandas dataframes [24]. For example, Figure 5(left) shows the edges table of the CG from Figure 4(b), while its nodes table is visible in the screenshot from Figure 6.

The nodes table has the following columns:

- node: the components numbers;
- size: the number of states of each component;
- on: the variables that are always on within each component;
- off: the variables that are always off within each component;
- topo: the topological properties, has_init means that a component has initial states, is_init that it has all the initial states, etc.;
- hasno, has, contains, isin, equals: the relationships between each component and the formulas that have been tested against it, as illustrated in Figure 5(right). For instance here, we have split component 5 from the CG of Figure 4(a) using formula $\varphi_{\textit{may-die}}$, that thus appears in the lines of the resulting components 7 and 8.

The edges table is simpler and straightforward. Both tables may be augmented by the user with arbitrary columns. The columns content may be used to tune the graphical presentation of the CG, *e.g.* their colour is often chosen from column size and their label from column node.

One important aspect of the nodes table is that its "relationship" columns (hasno to equals) are updated each time a formula is checked against some components. In ecco, a state formula φ is considered as a subset of the reachable states of the LTS, *i.e.* the set $S_\varphi \subseteq \mathcal{R}$ of states that satisfy the formula. Thus, it is possible to qualify the relationship between components and S_φ as illustrated in Figure 5(right). This information is then inherited by any new CG obtained through further splits so that the nodes tables keep track of how a CG has been obtained through successive splits. All together, the shape of a CG and its nodes and edges tables exhibit the understanding built incrementally by the analyst.

Figure 6 shows a screenshot of ecco in action on our termite model, producing and displaying the CG of Figure 4(b).

ecco includes a symbolic CTL model-checker that implements the algorithms from [19] on top of ITS-tools and libDDD, including the capability to take fairness constraints into account (expressed themselves as formulas). Note that CTL formulas are normally evaluated onto infinite computation trees. The algorithm is thus adapted to consider a deadlock as an infinite path that sustains the same state properties. This model-checker is a Python library that is available separately [25]. We also have a home-grown state expression language to express state properties in a functional way (DEAD, INIT, and HULL used in Figure 6 are atoms of this language). These two languages allow to express the properties used to split components.

ecco is available as a free software, released under the GNU LGPL, and is hosted at http://github.com/fpom/ecco. It can be run or installed as a Docker image [26] to ease its use. The

src	dst	rules
6	1	C1,R8,R9
	8	R3
	7	R3
3	6	R1
	1	R10,R12
7	6	R10
	8	R11

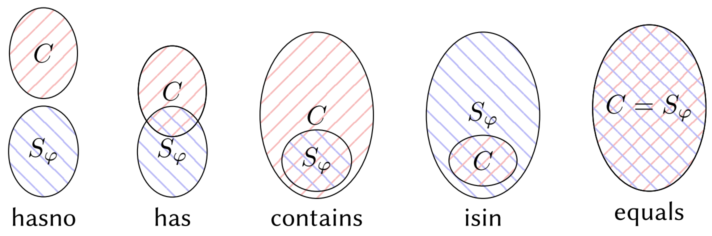
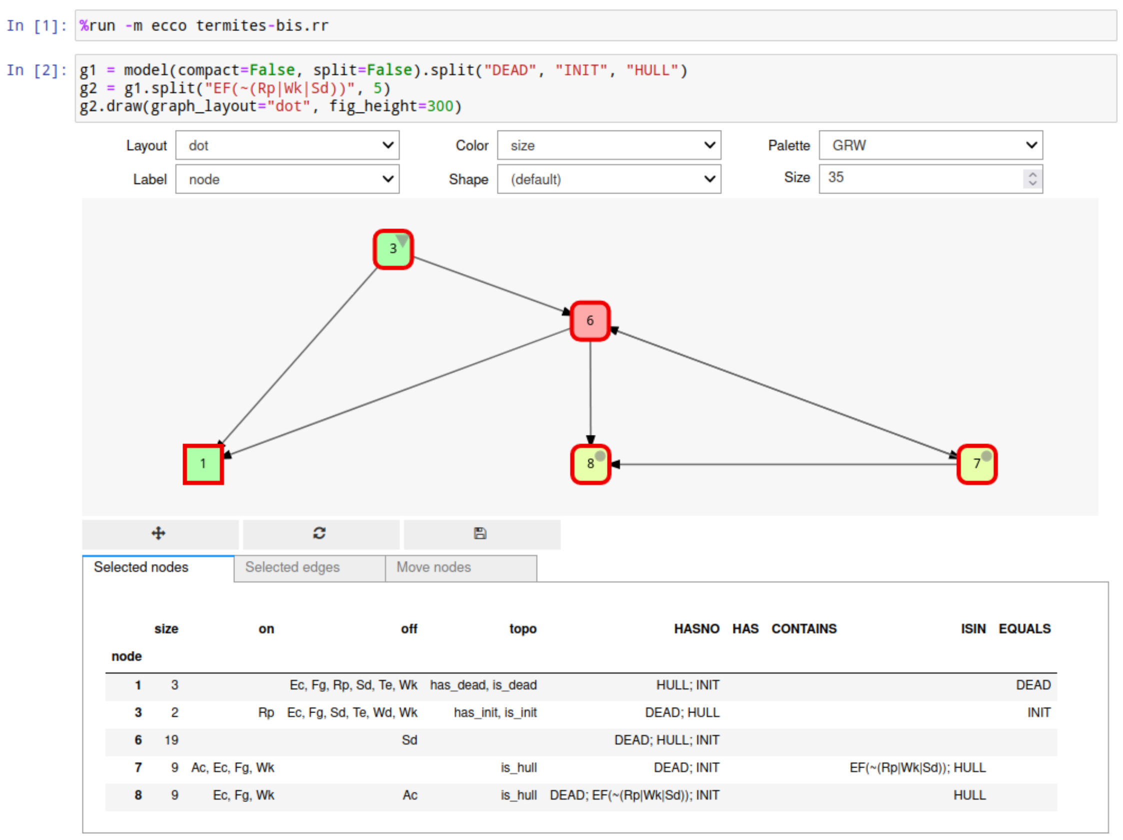


Figure 5: (Left.) The edges table for the CG from Figure 4(b). **(Right.)** The possible relationships between a component C and the set of states S_φ that satisfy a property φ.


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

In [2]: g1 = model(compact=False, split=False).split("DEAD", "INIT", "HULL")
        g2 = g1.split("EF(~(Rp|Wk|Sd))", 5)
        g2.draw(graph_layout="dot", fig_height=300)
```

Layout: dot
Color: size
Palette: GRW
Label: node
Shape: (default)
Size: 35

node	size	on	off	topo	HASNO	HAS	CONTAINS	ISIN	EQUALS
1	3		Ec, Fg, Rp, Sd, Te, Wk	has_dead, is_dead	HULL; INIT				DEAD
3	2	Rp	Ec, Fg, Sd, Te, Wd, Wk	has_init, is_init	DEAD; HULL				INIT
6	19		Sd		DEAD; HULL; INIT				
7	9	Ac, Ec, Fg, Wk		is_hull	DEAD; INIT			EF(~(Rp|Wk|Sd)); HULL	
8	9	Ec, Fg, Wk	Ac	is_hull	DEAD; EF(~(Rp|Wk|Sd)); INIT			HULL	

Figure 6: A simple ecco interactive session in a Jupyter Notebook.

distribution includes a script that can be the only installed element and that takes care of starting Docker with the appropriate options. For instance, with this script and Docker installed, starting ecco just requires to run “`ecco --mount=.`” from the command line. This retrieves online the Docker image for the latest version of ecco, starts it with an access to the files in the current directory, and opens Jupyter in the default web browser.

The Docker image features also JupyterHub [27] that is a multi-user server for Jupyter notebooks. Thus, ecco is readily configured to support multiple users with separated accounts.

5. Building Explanations

In this section, we illustrate the proposed analysis method by progressively splitting the state-space of the termite model of Figure 1. Note that this model and its analysis have a limited interest in ecology. However, they have been designed to provide a faithful witness of the EDEN methodology, while being simple enough for a short illustration. The Jupyter notebook corresponding to this section is available here: http://github.com/fpom/PNSE-22.

Can the termite colony collapse? Our central question is to understand how the colony may collapse. So, the first step is to split apart the states where no more termites are present, together with the states that inevitably lead to such a collapse. To do so, we use two CTL formulas: $\varphi_{dead} \overset{\text{df}}{=} \neg(\texttt{Rp} \vee \texttt{Wk} \vee \texttt{Sd})$ ("termites are dead"), and $\varphi_{must\text{-}die} \overset{\text{df}}{=} \mathsf{AF}(\mathsf{AG}\ \varphi_{dead})$ ("always in the future, it remains forever true that termites are dead"). This split yields CG G_1 depicted in Figure 7, which shows that termites can indeed disappear. Interestingly enough, G_1 shows that the states where φ_{dead} holds are exactly the same as those where $\varphi_{must\text{-}die}$ holds. It appears that the actions leading from component 2 to component 1 are R10 (`Ac+, Sd- >> Wk-, Rp-`) and R12 (`Te- >> Rp-, Sd-`). While it is expected that the former yields states where there are no termites, this is not trivial for the latter since it does not involve all the kinds of termites. This leads us to look how the rules may be chained and we can discover here a subtle causal sequence that starts with having `Wk-`, leads to `Wd-` through R5, then to `Te-` through R7, and results in firing R12 that removes the remaining termites.

```
dead = "~(Rp|Wk|Sd)"  # state has no termites
g0 = model(compact=False, split=False)  # G0 = full state space as a single component
g1 = g0.split(dead=dead, must_die=f"AF(AG{dead})")  # G1 = split G0 wrt Phi_dead and Phi_must_die
show(g1, 100)  # auxiliary function to display nodes/edges tables + graph of given height with tuned appearance
```

node	size	on	off	topo	HASNO	HAS	CONTAINS	ISIN	EQUALS
2	26			has_init	dead; must_die				
1	16		Rp, Sd, Wk	has_dead					dead; must_die

src	dst	rules
2	1	R10, R12

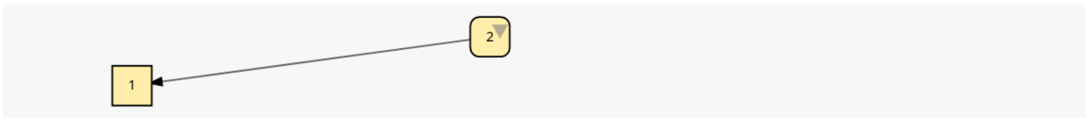


Figure 7: Component graph G_1 for the termite model where reachable states have been partitioned with respect to φ_{dead} and $\varphi_{must\text{-}die}$.

Can the termite colony avoid to collapse? Next question is to know if the termites can avoid to disappear. To answer it, we split the components where φ_{dead} does not hold with respect to $\varphi_{may\text{-}die} \overset{\text{df}}{=} \mathsf{EF}\ \varphi_{dead}$ ("possibly the termites can die"). This results in CG G_2 depicted in Figure 8. We can observe that the colony has a way to develop towards component 4: firing either R11 or R2. While the former is not surprising (termites kill ants), the latter just corresponds to a "normal" step in building the colony (reproductives give birth to workers) which does not appear as such a decisive step. However, it turns out to be a transition that prevents to have `Wk-`, which was identified in G_1 as a possible cause of collapse.

```
g2 = g1.split(*g1.hasno(dead), may_die=f"EF{dead}")  # g1.hasno(phi) => all the components where phi never holds
show(g2, 100)
```

node	size	on	off	topo	HASNO	HAS	CONTAINS	ISIN	EQUALS
1	16		Rp, Sd, Wk	has_dead					dead; must_die
4	11	Ec, Wk	Ac		dead; may_die; must_die				
3	15			has_init	dead; must_die			may_die	

src	dst	rules
3	4	R2, R11
	1	R10, R12

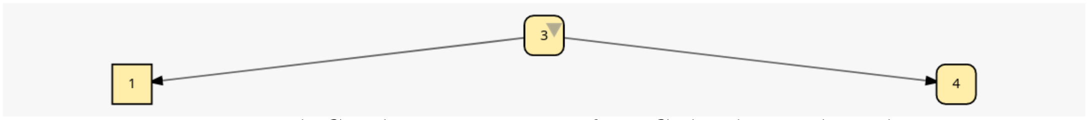


Figure 8: Component graph G_2 where component 2 from G_1 has been split with respect to $\varphi_{\textit{may-die}}$, yielding 3 and 4.

Can the branching between fates "always collapse" and "never collapse" be delayed forever? In G_2, from component 3, the colony may branch either to component 1 where it always collapses, or to component 4 where it never collapses. We want to know whether the colony may remain forever in such an uncertain state, or if the colony must always branch towards one or the other fate. To answer this, we can extract from component 3 its SCC's hull by splitting it against $\varphi_{delay} \overset{\text{df}}{=} \text{HULL}$ where HULL is the topological property (not a CTL formula) we have already used in Section 3.1 (see also Figure 6). The resulting CG G_3 is depicted in Figure 9. Note that, like in Figure 8, we have not directly used the component numbers to choose which one to split, but instead we have used a query expression provided by ecco to describe the components we are interested in (here: all those whose states do not satisfy φ_{dead} but satisfy $\varphi_{\textit{may-die}}$). From this split, we observe that the colony may remain forever in one or several SCCs from where it may branch to either fates. With respect to our previous observation on G_2 concerning R11 and R2, we observe here that R11 plays a crucial role only when the colony has already reached some stability in component 5. On the other hand, rule R2 is crucial at the very beginning of the behaviour.

What is the influence of ants on the colony's fates? By observing that R11 (termites kill ants) allows to reach component 4 from component 5, we may look more closely to G_3 and see that Ac is always off in component 4. So when R2 allows to reach component 4 from component 6, there are no ants already. Since the presence or absence of ants plays an important role in component 6, our last split aims at telling apart the initial states with ants from those without in order to better understand how they influence the ability of the system to remain forever uncertain about its fate. To do so, we split components with initial states with respect to $\varphi_{ants} \overset{\text{df}}{=} \text{Ac}$ that is only true in states having Ac+. The resulting CG G_4 is depicted in Figure 10. We observe that, without ants (component 8), branching is not avoidable, and once the colony has developed enough, it cannot collapse anymore (component 4). But, with ants (component 7), branching can be postponed forever (component 5) as long as there are ants, and developing enough is no longer a sufficient condition to avoid collapse (component 1). That is, the colony remains endangered until it can kill ants (R11 from component 5 to component 4).

```
g3 = g2.split(*(g2.hasno(dead) & g2.isin(f"EF{dead}")), delay="HULL")
show(g3, 250)
```

node	size	on	off	topo	HASNO	HAS	CONTAINS	ISIN	EQUALS
1	16		Rp, Sd, Wk	has_dead				may_die	dead; must_die
4	11	Ec, Wk	Ac		dead; may_die; must_die				
6	6		Fg, Sd, Te, Wd	has_init	dead; delay; must_die			may_die	
5	9	Ac, Ec, Fg, Wk		is_hull	dead; must_die			delay; may_die	

src	dst	rules
6	4	R2
	5	R3
	1	R10, R12
5	1	R10
	4	R11

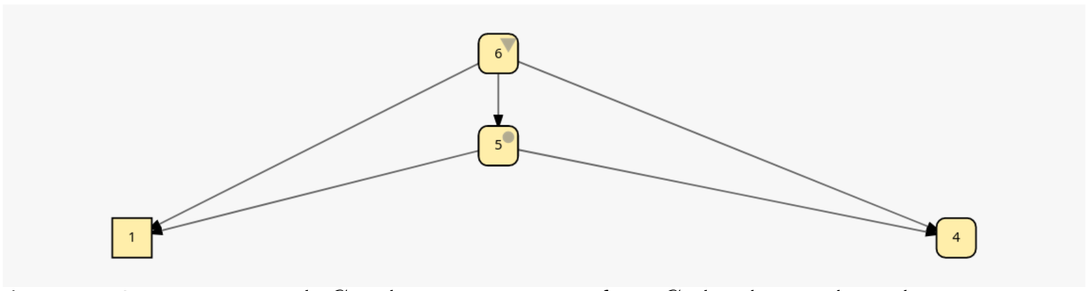


Figure 9: Component graph G_3 where component 3 from G_2 has been split with respect to φ_{delay}, yielding 5 and 6.

```
g4 = g3.split(*g3.has("INIT"), ants="Ac")
show(g4, 250)
```

node	size	on	off	topo	HASNO	HAS	CONTAINS	ISIN	EQUALS
1	16		Rp, Sd, Wk	has_dead	INIT			may_die	dead; must_die
4	11	Ec, Wk	Ac		INIT; dead; may_die; must_die				
5	9	Ac, Ec, Fg, Wk		is_hull	INIT; dead; must_die			delay; may_die	
7	4	Ac	Fg, Sd, Te, Wd	has_init	dead; delay; must_die	INIT		ants; may_die	
8	2	Rp	Ac, Fg, Sd, Te, Wd, Wk	has_init	ants; dead; delay; must_die	INIT		may_die	

src	dst	rules
5	1	R10
	4	R11
7	5	R3
	1	R10, R12
8	4	R2
	1	R12

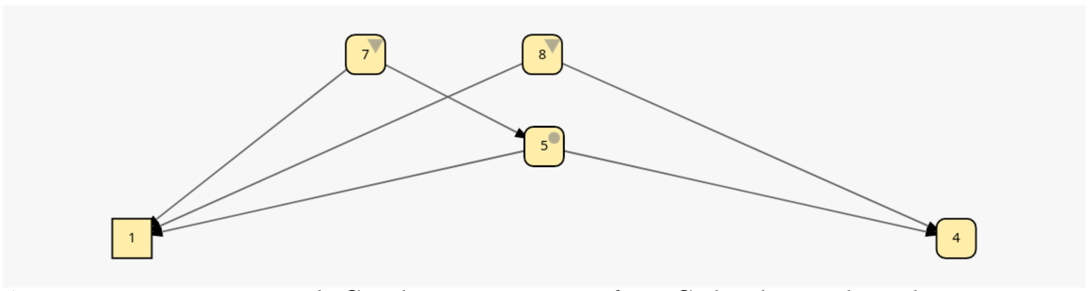


Figure 10: Component graph G_4 where component 6 from G_3 has been split with respect to φ_{ants}, yielding components 7 and 8.

6. Conclusion

We have presented the EDEN framework that is intended to support formal modelling and analysis of ecosystems. EDEN (and its implementation ecco) have been developed and used for more than five years and proved its relevance through a series of papers ranging from theoretical perspectives to applied studies [3, 6, 7, 8, 9, 10, 11], including the analysis of large models encompassing a few billions of states. Moreover, it has been used in the context of more than 10 master internships in ecology during which students have modelled ecosystems using RR/EH and have analysed their models using ecco. At the core of EDEN's analysis method is the notion of component graph that provides a synthetic view of potentially huge state-spaces, with the ability to explore them in an interactive way in order to incrementally build precise understanding about ecosystems dynamics.

An original aspect of this method is its use of well-known tools in computer science, like symbolic state-spaces or model-checking, by pushing them a bit further from their traditional use. Component graphs are not just symbolic state-spaces, but they are hybrid representations featuring symbolic components embedded within explicit graphs. Model-checking is not used to get yes/no answers, but to refine component graphs by discriminating precisely parts of a state-space. This incremental and interactive approach is not aimed at finding an answer to a question, but rather at progressively building an object that forms the answer to a question being itself progressively built.

This results in an original way of using symbolic model-checking, not as a push-button method as usually presented, but embedded into an interactive process, inspired from data science and theorem provers. This approach has been developed in the context of ecology, but we believe that it could be useful more generally. For example, in [28] a peer-to-peer storage system is modelled and its security is assessed in the presence of malicious peers. In this study, model-checking could not be used because an execution where malicious actions lead to file loss is always possible. Instead, [28] used simulation to quantify how the number of file loss was evolving with respect to the number of malicious peers in the system. With component graphs, it should be possible to go further, for instance by discriminating the states from which (1) files may possibly be lost, (2) files loss cannot be avoided anymore, and (3) no file can be lost. Then, by closely looking at the dynamics between these components, it may become possible to design a protocol that would force a malicious peer to behave correctly or to be detected as malicious. Such a protocol was a perspective in [28] but at this time no way was found to design such a solution. We believe today that a component graph analysis could be a helpful first step in this direction.

6.1. Related Works

Discrete formal modelling and analysis have been rarely used in the field of ecology where the dominant formalisms are rather based on systems of differential equations. Model-checking has been applied to study agrosystems modelled as timed-automata [29, 30], and Boolean networks (that are very popular in systems biology [31]) have been applied to study plant-pollinator associations and spruce budworm outbreaks [32, 33, 34].

It was shown in [35] that the RR language is strictly more expressive than Boolean networks. While RR allows to generate any LTS based on a set of Boolean variables, Boolean networks cannot generate a LTS where a state has two successors with inconsistent updates of the same variables (see, *e.g.* [15, Fig. 7]). Our experience showed that many ecosystem behaviours, when depicted as LTS, naturally exhibit such branching. More precisely, RR allows to express non-determinism at the level of rules (*e.g.* two rules with the same left-hand side but with incompatible right-hand sides), while with Boolean networks, non-determinism arises from the update scheme (asynchrony) only.

RR also shares similarities with Reaction Systems [36], both have a notion of a rule/reaction that is enabled by some positive and some negative conditions on Boolean variables, and whose firing yield a new state. But they diverge in two important aspects. First, Reaction Systems do not sustain the presence of reactants (in the absence of event to sustain a reactant, it disappears) while in RR, disappearance of an entity has to be explicit in a rule. Next, Reaction Systems do not have a notion of constraints or priorities between reactions (but extensions introduce time that can model kind of priorities).

Abstraction of an LTS through aggregation of its states is usually done by, first defining an adequate abstraction, which maps concrete states onto abstract states, and then reducing the state-space with respect to this abstraction. This reduction can be made on-the-fly while exploring the LTS, or afterwards. See [37, Sec. 13] for a quick survey of such approaches. Classically, a (bi)simulation relation is used to compute the so-called quotient LTS, which amounts to aggregate its (bi)similar states [4, Chap. 7]. Contrasting with existing works, our approach makes use of abstractions that are not defined *a priori*. Indeed, state-space partitioning, *i.e.* CG splits, is performed incrementally under the guidance of the analyst who aims at understanding the dynamics of the ecosystem, and the abstraction is a result of this process.

6.2. Perspectives

The EDEN framework is still under active development and stimulated by its numerous uses in a variety of contexts. When ecologists use EDEN to model and analyse an ecosystem, they often come with new questions, new needs, and even proposals of new analysis methods, that drive the evolution of the framework.

On the theoretical side, we intend to study the properties preserved or lost by splitting a component graph. So far, model-checking is always performed on the full symbolic state-space, and then its result is used for splitting components. But it would be good to known which properties a component graph itself has, when it is considered as an LTS. In particular it is not clear whether it validates or not the properties that have been used to split it.

Foreseen changes include new modelling languages, in particular with a representation of spatial information, allowing to model ecosystems as spatially-related patches, each with its own set of variables and actions linked to those of its neighbours. Introducing multivalued variables seems to be an obvious and not complicated next step, but we feel that there is still much to explore with Boolean variables, which proved useful in numerous ecological contexts [34, 38, 39]. This abstract approach is suitable in the field of ecology where ecosystems are usually not known in much details as very few observations are usually available considering the complexity of the ecosystems themselves.

Future analysis methods will take advantage of other temporal logics like in particular ARCTL [40], with the aim to build more compact and precise models. Indeed, using RR, one can model management policies (*i.e.* the choices of actions taken to control an ecosystem) as control variables whose initial state is the definition of a particular policy. This leads to increased combinatorial explosion, which is well fought by symbolic methods, but still may impair an interactive analysis. RR can be extended with labels on actions that could correspond to management policies, and ARCTL can then be used to quantify over these labels. Moreover, using ARCTL allows to naturally express changes in the management policies through time, which could not be obtained easily from control variables.

ecco will also evolve, integrating the foreseen works presented above. Moreover, as it has been the case so far, a lot of small new features or changes are likely to be included as users will report new needs or difficulties. From the general methodology presented in this paper, routine analyses are performed by ecologists. When such routines can be precisely characterised, it is generally easy to automate them in the implementation to streamline users' work. Beyond the engineering perspective, this dialogue between computer scientists and ecologists also leads the latter to better formalise their problems while helping the former to search how their well-known tools may be used in new ways to answer different questions.

If problems from other domains are to be explored using component graphs, it will be crucial to make ecco more generic with respect to the kind of models it can analyse. In principle, ecco is able to load the full range of formalisms that ITS-tools already handles (in particular: high-level or time Petri nets, timed automata, Promela, ...), but in practice, all the user-oriented programming and querying interface is crafted around the fact that models have Boolean variables only. As multivaluatedness is desired by some users in ecology, a generalisation would satisfy them and at the same time would open the possibility to analyse more diverse models.

Acknowledgments

We warmly thank Camille Carpentier, who has participated to the early development of the RR language, and Maximilien Cosme, for the numerous and fruitful discussions we had together about modelling and analysis, which helped to shape EDEN in its current form.

References

[1] Millennium Ecosystem Assessment, Ecosystems and human well-being, biodiversity synthesis, http://www.millenniumassessment.org/en/Synthesis.html, 2005.

[2] C. Gaucherel, H. Théro, A. Puiseux, V. Bonhomme, Understand ecosystem regime shifts by modelling ecosystem development using Boolean networks, Ecological Complexity 31 (2017).

[3] C. Gaucherel, F. Pommereau, Using discrete systems to exhaustively characterize the dynamics of an integrated ecosystem, Methods in Ecology and Evolution 10 (2019).

[4] C. Baier, J.-P. Katoen, Principles of Model Checking, The MIT Press, 2008.

[5] E. A. Emerson, Handbook of theoretical computer science, volume B, MIT press, 1991.

[6] C. Gaucherel, C. Carpentier, I. Geijzendorffer, F. Pommereau, Long term development of a realistic and integrated socio-ecological system, bioRxiv (2019). doi:10.1101/823294.

[7] 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:10.1155/2020/7163920.

[8] C. Di Giusto, C. Gaucherel, H. Klaudel, F. Pommereau, Analysis of discrete models for ecosystem ecology, in: Biomedical Engineering Systems and Technologies, Springer, 2020.

[9] C. Gaucherel, C. Carpentier, I. Geijzendorffer, C. Noûs, F. Pommereau, Discrete-event models for conservation assessment of integrated ecosystems, Ecological Informatics 61 (2021). doi:10.1016/j.ecoinf.2020.101205.

[10] 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.

[11] 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.

[12] C. Thomas, M. Cosme, C. Gaucherel, F. Pommereau, Model-checking ecological state-transition graphs, PLOS Computational Biology (accepted) (2022).

[13] E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic, in: Logics of Programs, volume 131 of *LNCS*, Springer, 1982.

[14] J. M. Perkel, Why Jupyter is data scientists' computational notebook of choice, Nature 563 (2018).

[15] F. Pommereau, C. Thomas, C. Gaucherel, Petri nets semantics of reaction rules (RR), in: PETRINETS'22, volume 13288 of *LNCS*, Springer, 2022.

[16] C. Di Giusto, C. Gaucherel, H. Klaudel, F. Pommereau, Pattern matching in discrete models for ecosystem ecology, in: Proc. of BIOINFORMATICS'19, SCITEPRESS, 2019.

[17] Y. Thierry-Mieg, From Symbolic Verification to Domain Specific Languages, Habilitation thesis, UPMC, 2016.

[18] D. Bérenguier, C. Chaouiya, P. T. Monteiro, A. Naldi, E. Remy, D. Thieffry, L. Tichit, Dynamical modeling and analysis of large cellular regulatory networks, Chaos: An Interdisciplinary Journal of Nonlinear Science 23 (2013). doi:10.1063/1.4809783.

[19] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang, Symbolic model checking: 10^{20} states and beyond, Information and Computation 98 (1992). doi:10.1016/0890-5401(92)90017-A.

[20] Python Software Foundation, The Python language, http://www.python.org, 2022.

[21] S. Behnel, al., Cython C-extensions for Python, http://cython.org, 2022.

[22] Y. Thierry-Mieg, Symbolic model-checking using ITS-tools, in: Proc. of TACAS'15, number 9035 in LNCS, Springer, 2015.

[23] Y. Thierry-Mieg, Homepage of ITS-tools, http://lip6.github.io/ITSTools-web, 2022.

[24] W. McKinney, et al., pandas: a foundational Python library for data analysis and statistics, Python for high performance and scientific computing 14 (2011).

[25] C. Thomas, Python CTL model checker, http://forge.ibisc.univ-evry.fr/cthomas/pyits_model_checker, 2022.

[26] C. Boettiger, An introduction to Docker for reproducible research, ACM SIGOPS Operating Systems Review 49 (2015).

[27] The Jupyter Project, JupyterHub, a multi-user version of the notebook designed for companies, classrooms and research labs, http://jupyter.org/hub, 2022.

[28] S. Chaou, F. Pommereau, Formal modelling and analysis of behaviour grading within a

peer-to-peer storage system, in: Proc. of TMS/DEVS'12, Society for Computer Simulation International, 2012, pp. 1–8.

[29] A. Hélias, F. Guerrin, J.-P. Steyer, Using timed automata and model-checking to simulate material flow in agricultural production systems—Application to animal waste management, Computers and Electronics in Agriculture 63 (2008). doi:10.1016/j.compag.2008.02.008.

[30] M.-O. Cordier, C. Largouët, Y. Zhao, Model-Checking an Ecosystem Model for Decision-Aid, in: 2014 IEEE 26th International Conference on Tools with Artificial Intelligence, 2014. doi:10.1109/ICTAI.2014.87.

[31] A. Naldi, P. T. Monteiro, C. Müssel, the Consortium for Logical Models, Tools, H. A. Kestler, D. Thieffry, I. Xenarios, J. Saez-Rodriguez, T. Helikar, C. Chaouiya, Cooperative development of logical modelling standards and tools with CoLoMoTo, Bioinformatics 31 (2015). doi:10.1093/bioinformatics/btv013.

[32] C. Campbell, S. Yang, R. Albert, K. Shea, A network model for plant–pollinator community assembly, Proceedings of the National Academy of Sciences 108 (2011). doi:10.1073/pnas.1008204108.

[33] T. LaBar, C. Campbell, S. Yang, R. Albert, K. Shea, Global versus local extinction in a network model of plant–pollinator communities, Theoretical Ecology 6 (2013). doi:10.1007/s12080-013-0182-8.

[34] R. Robeva, D. Murrugarra, The spruce budworm and forest: a qualitative comparison of ODE and Boolean models, Letters in Biomathematics 3 (2016). doi:10.1080/23737867.2016.1197804.

[35] C. Thomas, Model checking applied to discrete models of ecosystems, Master's thesis, ENS Paris-Saclay, 2019.

[36] A. Ehrenfeucht, G. Rozenberg, Basic notions of reaction systems, in: Developments in Language Theory, volume 3340 of *LNCS*, Springer, 2005. doi:10.1007/978-3-540-30550-7_3.

[37] E. M. Clarke, B. Schlingloff, Model checking, in: Handbook of Automated Reasoning, Elsevier, 2001.

[38] J. D. Yeakel, M. M. Pires, M. A. M. de Aguiar, J. L. O'Donnell, P. R. Guimarães, D. Gravel, T. Gross, Diverse interactions and ecosystem engineering can stabilize community assembly, Nature Communications 11 (2020). doi:10.1038/s41467-020-17164-x.

[39] C. Song, T. Fukami, S. Saavedra, Untangling the complexity of priority effects in multispecies communities, Ecology Letters 24 (2021). doi:10.1111/ele.13870.

[40] C. Pecheur, F. Raimondi, Symbolic Model Checking of Logics with Actions, in: Model Checking and Artificial Intelligence, Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-74128-2_8.