FormalMiner: A Formal Framework for Refinement Mining

Antonio Cerone(✉)

Department of Computer Science, Nazarbayev University, Astana, Kazakhstan
antonio.cerone@nu.edu.kz

Abstract. Refinement mining has been inspired by process mining techniques and aims to refine an abstract non-deterministic model by sifting it using event logs as a sieve until a reasonably concise model is achieved. FormalMiner is a formal framework that implements model mining using Maude, a modelling language based on rewriting logic. Once the final formal model is attained, it can be used, within the same rewriting-logic framework, to predict the future evolution of the behaviour through simulation, to carry out further validation or to analyse properties through model checking. In this paper we focus on the refinement mining capability of FormalMiner and we illustrate it using a case study from ecology.

Keywords: Formal methods · Model-driven approaches
Rewriting logic · Maude · Process mining
Application to ecosystem modelling

1 Introduction

The use of large repositories to collect data in various domains of social sciences, physical sciences and life sciences offers great opportunities for systematic analysis. *Data mining* aims to extract meaningful information from data and exploit it to describe and understand the processes that have generated such data.

More recently, the scope of data mining enlarged from the description of properties of the data organisation, such as clustering and classification, to the description of the actual process that led to the creation and organisation of the data. *Process mining*, which emerged in the field of business process management (BPM), has been used to extract information from event logs consisting of activities and then produce a graphical representation of the process control flow, detect relations between components involved in the process and infer data dependencies between process activities [20]. This is achieved through either the *discovery* of an *a posteriori* process model or the *extension* of a pre-existing *a priori* model, or the comparison of the *a priori* model with the event logs using a technique called *conformance analysis* [15]. However, these three approaches cannot be automatically integrated, but require the analyst to compare them manually [13]. Therefore, process mining is used for *descriptive* purposes, aiming at the discovery of some aspects of the *past behaviour*, that is, the dynamics that produced the event log.

M. Mazzara et al. (Eds.): STAF 2018 Workshops, LNCS 11176, pp. 185–200, 2018.
https://doi.org/10.1007/978-3-030-04771-9_15

Although there are a number of works in the areas of synthesis of programs [10,17,18] and synthesis of biological and probabilistic systems from data [7,11,14], to our knowledge, the only attempt to integrate process mining and formal verification is a work by van der Aalst, de Beer and van Dongen's, which aims at verifying whether an event log satisfies a property expressed in linear temporal logic (LTL) [19]. However, such an approach still has a descriptive purpose: the formal characterisation of properties of the event logs from *past behaviour*. The construction of a formal model within a framework equipped with automatic verification tools, instead, enable prediction of *future behaviour*.

In our previous work [5] we have taken a step forward and exploited real data in a *constructive* rather than descriptive way by integrating techniques from the realm of process mining with modelling approaches. The technique we developed, which we called *model mining*, supports the synthesis of a formal model from a dataset and enables the formal analysis of such a model in order to predict the *future behaviour* and characterise its properties. The synthesised model is not a mere representation of the unfolded behaviour, but comprises, instead, a set of formal transition rules for generating the system behaviour, thus supporting powerful predictive capabilities. The set of transition rules can be either inferred directly from the events logs (*constructive mining*) [5] or refined by sifting a plausible *a priori* model using the event logs as a sieve until a reasonably concise model is achieved (*refinement mining*) [4,5]. To this purpose, events are partitioned into two classes, *environmental events*, which allow us to update the system state, and *target events*, which are used in refinement mining to sift a non-deterministic model by possibly invalidating one of its deterministic instances.

We use *equational logic* [9] to define the Model Mining Formal Framework (MMFF) [4,5], which provides a formal description of the events, the system state and the transition rules that change the system state accoding to the event occurrences. *Rewriting logic* [12] is then used to manipulate the data structures defined in MMFF in order to implement the *constructive mining algorithm*, which exploits a list of events to build a set of transition rules, and the *refinement mining algorithm*, which exploits a list of events to refine an *a priori* model consisting of sets of alternative transition rules by reducing the possible alternatives (thus reducing non-determinism) [5].

In this tool paper we introduce *FormalMiner*, which implements MMFF and the two model mining algorithms using the *Maude rewrite system* [8]. Maude is a high-performance modelling and analysis system based on a reflexive language that supports both equational and rewriting logic.

The focus of this paper is the use of FormalMiner in performing refinement mining. Section 2 introduces rewriting logic and Maude and briefly overviews the Maude syntax to enable the reader to understand the examples in this paper and refer them to the Maude code that implements FormalMiner. FormalMiner 1.1, on which this paper is based, and the case study can be downloaded at

https://cs-sst.github.io/faculty/cerone/modellingfromdata.

Sect. 3 presents the architecture of FormalMiner and illustrates the data structures of MMFF and the general functions for manipulating them. Section 4 describes how refinement mining is carried out and introduces the *sifting metric*, which is used to evaluate the validity of alternative instances of the model. Finally, Sect. 5 discusses strategies for carrying out model mining as well as future work.

2 Rewriting Logic and the Maude System

Rewriting logic [12] is based on *rewite rules* of the form $t \Longrightarrow t'$, with t and t' expressions in a given language. A rewrite rule can be interpreted both *computationally*, as a *local transition* in a concurrent system, and *logically*, as an *inference rule*. Therefore, rewriting logic can be considered both a *computational theory* and a *logical theory*.

Maude [8] is a modelling language based on rewriting logic. It also provides model-checking capabilities, thus supporting the analysis of the modelled system. In this work we only use *Core Maude*, whose syntax we briefly overview in this section. There are two types of modules in Core Maude, *functional modules*, which are restricted to *equational logic* and support declaration of sorts (with keyword `sort` for one sort, or `sorts` for many), operations (with keyword `op` or `ops`) on them and the definition of such operation using equations (with keyword `eq`, or `ceq` in case of conditional equations), and *system modules*, which also support *rewriting logic*, by additionally including the definition of rewrite rules (with keyword `rl`, or `crl` in case of conditional rewrite rules). A number of *flags* can be used while defining an operation. For example flag `ctor` designates the operation as a constructor.

A sort `A` is specified as a subsort of a sort `B` by '`subsort A < B`'. Keyword `subsorts` is used in case of multiple subsorts. Variables denote indefinite values for sorts to be used within equations or rewrite rules. They are declared with keyword `var` or `vars`. Constants are basically operations with no arguments and are defined through equations. Maude has several predefined sorts for basic values, including the obvious sorts `Bool`, `Nat`, `Int`, `Rat`, and a sort `Qid` for *quoted identifiers*, which are sequences of characters starting with the character '`'`'. All constructs of the Maude language, apart from the toplevel module construct, end with a space followed by a dot ('.').

3 FormalMiner Core: Events, States and Evolution

In order to be successfully processed with the purpose of extracting process control flow information, event logs have first to be semantically interpreted according to the purpose of the model we aim to devise, so that such interpretation can drive their structuring and clustering. Structural and semantical organisation can be attained by applying text mining techniques, in particular semantic indexing, in combination with an appropriate ontology from the given application domain. This approach is commonly used in process mining, which

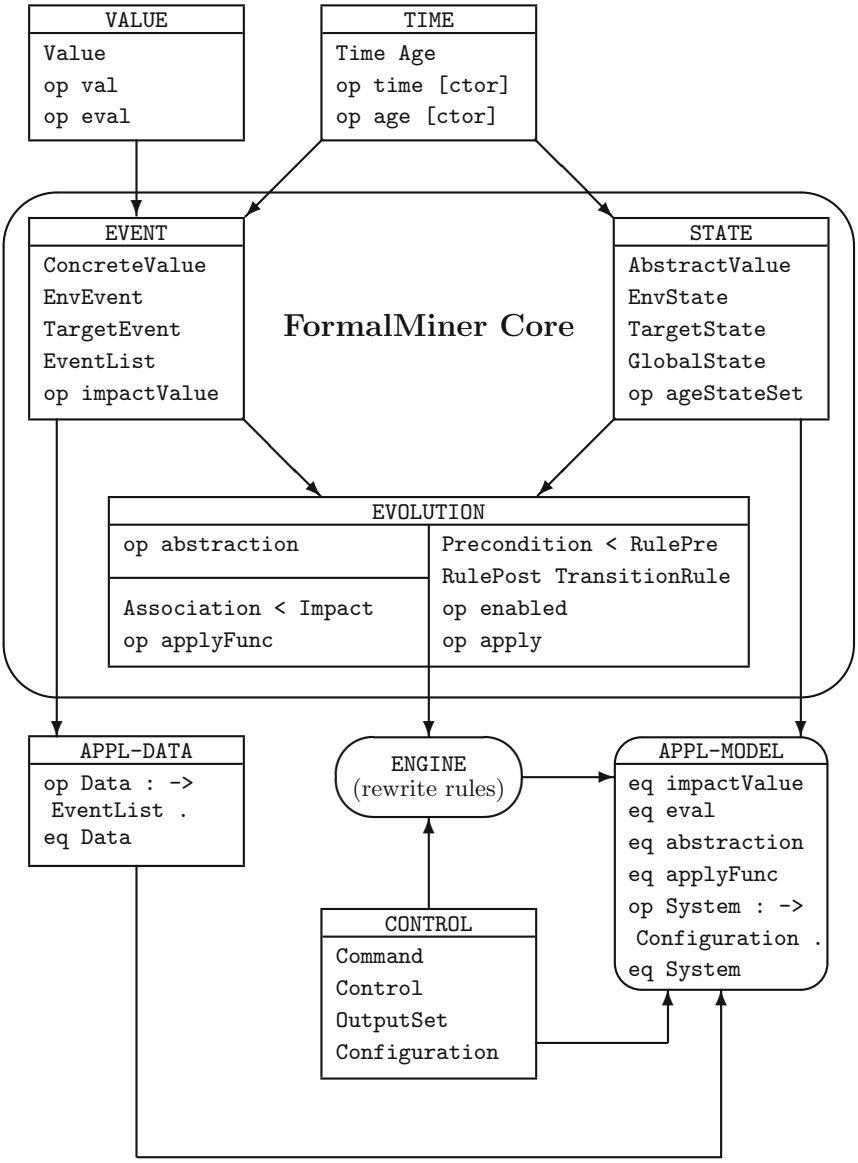

Fig. 1. Architecture of the FormalMiner tool in terms of Maude modules.

is thus applied to a set of *pre-processed* event logs. For the purpose of model mining we assume to have already pre-processed events organised as a sequence of structured entities, which are ordered based on their occurrence times.

In this section we show the basic data structures that make up MMFF and how they are implemented in FormalMiner. For this purpose we refer to the

FormalMiner architecture described in Fig. 1, where squared boxes represent functional modules and oval boxes represent the system modules. An arrow between module `M1` and module `M2` denotes that `M2` imports all declarations and definitions in `M1`. The FormalMiner Core implements MMFF data structures and the general functions to manipulate them. It consists of three functional modules: `EVENT`, `STATE` and `EVOLUTION`. System module `ENGINE`, which comprises the rewrite rules that implement the model mining algorithms, and functional module `CONTROL`, which control FormalMiner output, are also part of the MMFF implementation.

Outside the FormalMiner Core, modules `VALUE` and `TIME` contain declarations of generic sorts, whereas constructors and operation declaration and definition are depending on the value and time domains considered for the specific application. Thus, although the structure of these two modules is standard and includes generic sorts `Value`, `Time` and `Age`, such sorts need to be instantiated for each specific application.

Also outside the FormalMiner core, functional module `APPL-DATA` and system module `APPL-MODEL` are application specific. Module `APPL-DATA` contains the dataset, in the form of a defined timed list of events. Module `APPL-MODEL` contains the definition of the initial configuration, which includes the initial state of the system and the plausible model to refine. The names for these two modules provided here and in Fig. 1 are just placeholders and can be changed to best describe the application. This is consistent with the fact that these two modules are not imported by any other module.

Our goal is to model how environmental events affect a target event. Environmental events act on the system by changing its state and such changes are normally visible through target events. Since the system consists of several components, which we call *domains*, we need to define a local notion of state for each of such domains. We denote such a notion of local state as *domain state*.

3.1 Environmental and Target Events

Environmental and target events are modelled in the `EVENT` module. An *event* is defined as a triple consisting of

- the *time* at which the event occurs;
- the *event name*;
- a *concrete value* represented as a set $\{t_1(v_1^{V_1}), \ldots, t_n(v_n^{V_n})\}$ of typed basic values, with t_i being the type name and $v_i^{V_i}$ belonging to value domain V_i, for $i = 1. \ldots, n$.

Sorts `BasicEvent`, `EnvEvent` (environmental events) and `TargetEvent` (target events), as well as `Event`, which comprises the last two as subsorts, and `EventList` (sequences of events), are declared in Maude together with the definitions of their operations as follows:

```
sorts BasicEvent Event EnvEvent TargetEvent EventList .
subsorts EnvEvent TargetEvent < Event < EventList .
```

```
op [|_,_:_|] : Time Qid ConcreteValue -> BasicEvent [ctor prec 30] .
op env_ : BasicEvent -> EnvEvent [ctor prec 31] .
op target_ : BasicEvent -> TargetEvent [ctor prec 31] .

op noEvents : -> EventList [ctor] .
op _&>_ :  EventList EventList
              -> EventList [ctor assoc prec 32 id: noEvents] .
```

Constructors `env` and `target` denote environmental and target events respectively. Constructor `&>` sequentialises events.

In previous work [1] we modelled the dynamic of a population of *Aedes albopictus*, a mosquito species known as "tiger mosquito", which is endemic in Asian regions, where it is a carrier of dengue fever, and now widespread also in Europe. The model developed in that work is based on biological aspects of the mosquito and considers the impact of changes in the environmental conditions on such biological aspects to simulate the population dynamics. Among relevant environmental conditions are average temperature and rain amount. The simulation made use of data on the size of the mosquito population collected during May–November 2009 in the province of Massa-Carrara (Tuscany, Italy) using CO2 mosquito traps. We will use this case study throughout the paper to illustrate FormalMiner.

```
eq May = env[| time(1) , 'Temp : 'avg(val(18)) |]         ---   8 May
      &>  target[| time(1) , 'Aedes : 'adult(val(4)) |]
      &>  env[| time(2) , 'Temp : 'avg(val(19)) |]        ---   9 May
...
      &>  env[| time(22) , 'Temp : 'avg(val(20)) |]       ---   29 May
      &>  env[| time(23) , 'Temp : 'avg(val(20)) |]       ---   30 May
      &>  env[| time(24) , 'Temp : 'avg(val(22)) |] .     ---   31 May
...
eq August = env[|  time(96) , 'Temp : 'avg(val(28)) |]    ---   1 August
     &>  env[| time(97) , 'Temp : 'avg(val(31)) |]        ---   2 August
     &>  env[| time(98) , 'Temp : 'avg(val(31)) |]        ---   3 August
     &>  env[| time(99) , 'Temp : 'avg(val(34)) |]        ---   4 August
     &>  env[| time(100) , 'Temp : 'avg(val(33)) |] .     ---   5 August

eq Data = May &> June &> July &> August .
```

Fig. 2. Sequence of events describing changes of temperature and sampling of the mosquito population.

Figure 2 shows fragments of a sequence of events over a period of 100 days from 8 May to 5 August 2009. There are only one kind of environmental event, named `Temp` and describing daily temperature changes, and one kind of target event, named `Aedes` and describing the sampling of the mosquito population on a specific day. In both kinds of events the concrete value is a singleton describing the average value for temperature changes and the values for the sampling of the mosquito adult population. A richer concrete value for temperature would have also values for minimum and maximum temperature, for example:

```
env[| time(1), 'Temp: 'min(val(13)) 'avg(val(18)) 'max(val(20)) |]
```

3.2 Domain State and Global State

Differently from events, which are concrete entities that accurately represent the reality, states, modelled in module `STATE`, refer to the model rather than the reality. They are thus abstract entities, whose values are abstract values.

A *domain state* is defined as a quadruple consisting of

- the *domain name*;
- the *set of specifiers* for the domain;
- the *abstract value* that refers to the domain and its specifiers;
- the *state age*, i.e. the amount of time that the state has persisted with unchanged abstract value.

Similarly to events, states may be environmental and target. Environmental states are changed by the occurrence of environmental events (i.e. by the data as event occurrences), whereas target states are changed according to the model (i.e. by the model execution) and will be then validated against the target events. The *global state* of the system we are modelling consists of the *current time* and a set of environmental and target states. The Maude syntax for sorts `DomainState`, `StateSet` and `GlobalState` is

```
op <|_,_,_,_|> : Qid Specifiers AbstractValue Age -> DomainState [...] .
op __ :  StateSet StateSet -> StateSet [...] .
op _at_ :  StateSet Time -> GlobalState [...] .
```

For example, we may consider four possible abstract values for the size of the adult *Aedes* population: `low` for low, `med` for medium, `high` for high and `extr` for extreme and the first three also for average temperatures. Then

```
env<| 'Temp,'avg,'med,age(2) |>  target<| 'Aedes,'adult,'med,age(5) |>
at time(0)
```

is the global state describing the initial day (`time(0)`) in which the average temperature has a medium value (`med`) that persisted for 2 days (`age(2)`) and the adult mosquito population has a medium value (`med`) that persisted for 5 days (`age(5)`).

3.3 Environment-Driven Evolution

In this section we describe how concrete values of environmental events are mapped to abstract values, which then determine the evolution of environmental states.

Although in our example we have used the same name for corresponding events and states, in general events may be defined with no explicit reference to the affected domains. The evolution of environmental states is described by an *impact relation*, which is implemented by sort `Impact`, whose elements are sets of impact associations of sort `Association`. These sorts are declared in module `EVOLUTION`, where `Association` < `Impact` denotes that `Association` is a subsort of `Impact`, and defined by the equations in application-specific system module `APPL-MODEL`.

An impact association defines, under a given condition on the concrete value of the environmental event, which environmental state is affected by the occurrence of the event and how it is affected. The syntax of an impact association

$$\texttt{\{}\ \langle EventName\rangle\ \texttt{,}\ \langle CondLabel\rangle\ \texttt{|}\ \langle FuncLabel\rangle\ \texttt{|}\ \langle StateName\rangle\ \texttt{,}\ \langle Specifiers\rangle\ \texttt{\}}$$

defines that an event named $\langle EventName\rangle$

- is associated with a state named $\langle StateName\rangle$ and having $\langle Specifiers\rangle$ as set of specifiers, and
- determines a state transition defined by label $\langle FuncLabel\rangle$, which represents an *evolution function* (not a Maude function!),

when the condition labelled as $\langle CondLabel\rangle$ is true.

The impact relation for our case study consists of the following three impact associations

```
{ 'Temp , 'lowTempCond | 'lowTemp | 'Temp , 'avg }
{ 'Temp , 'medTempCond | 'medTemp | 'Temp , 'avg }
{ 'Temp , 'highTempCond | 'highTemp | 'Temp , 'avg }
```

Actual conditions and model functions are defined by Maude functions `eval` and `applyFunc` in module `APPL-MODEL`. For example, in our case study

```
eq eval('lowTempCond , V) = 0 <= V and V < 20 .
```

defines that condition labelled as `lowTempCond` is true when average temperature is less than 20°, and

```
eq applyFunc('lowTemp , AV ) = 'low .
```

defines that the function represented by label `lowTemp` is the constant function `low`, independently of the previous abstract value `AV` of the average temperature. Actual conditions and evolution functions of the impact relation for our case study are given in Table 1.

Table 1. Conditions and evolution functions of the impact relation for the mosquito case study

State Name	Condition Label CL	Condition Value eval(CL,V)	Function Label CL	Evolution for any AV applyFunc(CL, AV)	Event Name	Concrete Value
Temp	lowTempCond	$0 \leq$ V < 20	lowCond	constant low	Temp	avg(V)
	medTempCond	$20 \leq$ V < 25	medCond	constant med		
	highTempCond	$25 \leq$ V < 36	highCond	constant high		

Although this approach may seem cumbersome for the simple case study we consider in this paper, it supports a general form of modelling. For example,

```
{ 'Rainfall , 'low2medCond | 'low2med | 'Water , 'moisture }
eq eval('low2medCond , amount(A) intensity(I)) = A*K/I>=2 and A*K/I<10 .
eq applyFunc('low2med , 'low ) = 'med .
```

describes the impact of the rain on a soil with permeability characterised by parameter `K`: a rainfall with amount `A` in millimetres and intensity `I` determines an increase of soil moisture from low to medium if $2 \leq$ `A*K/I` < 10. The fact that the higher the intensity of the rainfall the faster water tends to flow and the less it is absorbed, thus resulting in a lower level of soil moisture, is described by `A*K/I`. Obviously, an environmental event for this example needs two types of basic value, `amount` and `intensity` to make a concrete value, for example

```
env[| time(5), 'Rainfall: 'amount(val(13)) 'intensity(val(2)) |]
```

3.4 Model-Driven Evolution

In this section we describe how the plausible model determines the transition of target states. The mapping from concrete values of target events to abstract values of target states is defined by function `abstraction` declared in functional module `EVOLUTION` and defined by the equations given in functional module `APPL-DATA`. For example, the abstraction for our case study, given in Table 2 is defined by

Table 2. Abstract relation for the mosquito case study

Event Name	Typed Basic Value	Condition on Concrete Value `V`	Abstract Value
`Aedes`	`adult(V)`	$0 \leq$ `V` < 100	`low`
		$100 \leq$ `V` < 250	`med`
		$250 \leq$ `V` < 500	`high`
		`V` > 500	`extr`

```
eq abstraction('Aedes , 'adult , V) =
   if 0 <= V and V < 100 then 'low else
     if 100 <= V and V < 250 then 'med else
       if 250 <= V and V < 500 then 'high else 'extr fi fi fi .
```

The transition of target states is modelled by *transition rules* whose components are declared in module `EVOLUTION`. The rule precondition is defined by sort `RulePre`, which is a set of preconditions of sort `Precondition`, where each precondition states the existence in the global state of an environmental state of given domain and set of specifier such that the age of the abstract value has a given relation with a given threshold. For example

```
( 'Temp , 'avg , 'med | >= age(10) >>)
```

denotes that a medium abstract value of the average temperature has persisted (relation `>=` between age and threshold) for at least 10 days (threshold expressed by `age(10)`). The rule postcondition is defined by sort `RulePost`. For example

```
(<< 'Aedes , 'adult | 'low -- 'increase -> 'med >>)
```

denotes an increase of the mosquito adult population from low to medium. Given an appropriate abstraction of concrete values, it may be obvious that the persistence of a medium temperature for a certain number of days results in an increase of the adult mosquito population from low to medium. However, it is not clear how many days are needed for such an increase. Therefore, we normally need to have, within the same model, distinct transition rules with the same postcondition but preconditions which are not mutually exclusive. A good strategy is to initially include two rules corresponding to what we expect to be reasonable lower and upper bound limits. In our specific case, we may expect the population increase to occur on the same day as the temperature increase or within up to 10 days:

```
( 'Temp , 'avg , 'med | >= age(0) >>)
                => (<< 'Aedes , 'adult | 'low -- 'increase -> 'med >>)
( 'Temp , 'avg , 'med | >= age(10) >>)
                => (<< 'Aedes , 'adult | 'low -- 'increase -> 'med >>)
```

We might even not be sure that the population size can change within these 10 days. Then we also add transition rules

```
( 'Temp , 'avg , 'med | >= age(0) >>)
                => (<< 'Aedes , 'adult | 'low -- 'stable -> 'low >>)
( 'Temp , 'avg , 'med | >= age(10) >>)
                => (<< 'Aedes , 'adult | 'low -- 'stable -> 'low >>)
```

These four rules are grouped together in a set called *option set*, which describes a form of non-determinism, called *option-related non-determinism*, which we expect to be reduced by sifting out those transition rules that are invalidated by the data. Therefore, refinement mining exploits data, in the form of event logs consisting of target events, to reduce option-related non-determinism.

However, not all determinism may be reduced using refinement mining. For example, in ecology, there might be alternative forms of behaviour of individuals which are dictated by their free will. Such alternatives must be modelled by transition rules of distinct option sets. This form of non-determinism, which we call *model-related non-determinism* is intrinsic to the model and cannot be reduced using refinement mining. Finally, the impact of environmental events on environmental states may depend on unknown or only partially known factors. Or it may depend on known factors, which, however, we may have chosen not to model or we cannot model because they depend, in turn, on factors which are either unknown or too complex to include in the model. For example, soil moisture may also depend on desiccation, which, however, we may have chosen not to model or we may not be sure how to model. We call this third form of non-determinism *impact-related non-determinism* and we model it by impact associations whose conditions are not mutually exclusive.

We can thus define a *plausible model* as a set of option sets. The plausible model for our case study is given in Fig. 3. It consists of 10 option sets, numbered 1 to 10. Option set 4 consists of the 4 alternative transition rules

```
[1] < { 1 | ( 'Temp , 'avg , 'low | >= age(0) >>)
            => (<< 'Aedes , 'adult | 'extr -- 'decrease -> 'med >>) }  >
[2] < { 1 | ( 'Temp , 'avg , 'low | >= age(0) >>)
            => (<< 'Aedes , 'adult | 'high -- 'decrease -> 'low >>) }  >
[3] < { 1 | ( 'Temp , 'avg , 'low | >= age(0) >>)
            => (<< 'Aedes , 'adult | 'med -- 'decrease -> 'low >>) }  >
[4] < { 1 | ( 'Temp , 'avg , 'med | >= age(0) >>)
            => (<< 'Aedes , 'adult | 'low -- 'increase -> 'med >>) }
      { 2 | ( 'Temp , 'avg , 'med | >= age(10) >>)
            => (<< 'Aedes , 'adult | 'low -- 'increase -> 'med >>) }
      { 3 | ( 'Temp , 'avg , 'med | >= age(0) >>)
            => (<< 'Aedes , 'adult | 'low -- 'stable -> 'low >>) }
      { 4 | ( 'Temp , 'avg , 'med | >= age(10) >>)
            => (<< 'Aedes , 'adult | 'low -- 'stable -> 'low >>) } >
[5] < { 1 | ( 'Temp , 'avg , 'med | >= age(0) >>)
            => (<< 'Aedes , 'adult | 'high -- 'decrease -> 'med >>) }
      { 2 | ( 'Temp , 'avg , 'med | >= age(4) >>)
            => (<< 'Aedes , 'adult | 'high -- 'decrease -> 'med >>) } >
[6] < { 1 | ( 'Temp , 'avg , 'med | >= age(0) >>)
            => (<< 'Aedes , 'adult | 'extr -- 'decrease -> 'high >>) } >
[7] < { 1 | ( 'Temp , 'avg , 'high | >= age(0) >>)
            => (<< 'Aedes , 'adult | 'low -- 'increase -> 'med >>) }
      { 2 | ( 'Temp , 'avg , 'high | >= age(4) >>)
            => (<< 'Aedes , 'adult | 'low -- 'increase -> 'med >>) } >
[8] < { 1 | ( 'Temp , 'avg , 'high | >= age(0) >>)
            => (<< 'Aedes , 'adult | 'med -- 'increase -> 'high >>) }
      { 2 | ( 'Temp , 'avg , 'high | >= age(4) >>) $\!\!\!\!\!\!\!$
            => (<< 'Aedes , 'adult | 'med -- 'increase -> 'high >>) } >
[9] < { 1 | ( 'Temp , 'avg , 'high | >= age(0) >>)
            => (<< 'Aedes , 'adult | 'med -- 'increase -> 'extr >>) }
        { 1 | ( 'Temp , 'avg , 'high | >= age(16) >>)
            => (<< 'Aedes , 'adult | 'med -- 'increase -> 'extr >>) } >
[10] < { 1 | ( 'Temp , 'avg , 'high | >= age(0) >>)
            => (<< 'Aedes , 'adult | 'high -- 'increase -> 'extr >>) }
       { 2 | ( 'Temp , 'avg , 'high | >= age(2) >>)
            => (<< 'Aedes , 'adult | 'high -- 'increase -> 'extr >>) } >
```

Fig. 3. Plausible model for the mosquito case study.

illustrated above. Some option sets consist of just one transition rules. This is the case for option sets 1–3, due to the belief that a low temperature will always cause a decrease of the adult mosquito population to low on the same day, independently of the size of the initial population. Similar beliefs are that a medium temperature will always cause a decrease of the adult mosquito population from extreme to high on the same day (option set 6). The other option sets (5 and 7–10) consist of two rules corresponding to the same day as the lower bound and what we expect to be a reasonable upper bound.

4 Plausible Model Refinement

In order to check which transition rules are invalidated by the data, each transition rule of an option set has to be consistently checked against the entire dataset. To this purpose, the plausible model is decomposed in all possible *option-free* models, that is, models without option-related non-determinism; each model is then used to perform a simulation, during which the results of simulation steps are compared with the target events. For each option-free model, the model

refinement engine records the number of times the model is invalidated by a target event and, for each transition rule used within that model, it also records the number of times the rule is applied through simulation. Finally, the recorded information on the number of times the model is invalidated and its rules are applied is combined into a *sifting metric* that is associated with that specific model in order to provide a measure of model validation.

For each option-free model considered, the refinement mining returns a set Ψ of *option references* representing the selected transition rule for each option set and providing the number of times such a rule is applied. These references are formally defined as follows.

Definition 1. *An* ***option reference*** *is defined as a Maude term* `[`k `:` i`(`j`)]` *where*

- k *is a reference to the option set whose identification number is* k*;*
- i *is a reference to the transition rule whose identification number is* i *within the option set identified by* k*;*
- j *is the number of times the referred transition rule has been applied during the current simulation.*

For example, in the following list of option references represented as a Maude term

```
[1 : 1(0)] [&> [2 : 1(0)] [&> [3 : 1(1)] [&>
[4 : 2(1)] [&> [5 : 1(0)] [&> [6 : 1(2)] [&>
[7 : 1(0)] [&> [8 : 1(1)] [&> [9 : 2(0)] [&> [10 : 1(3)]        (1)
```

rule 1 of option set 3, rule 2 of option set 4 and rule 1 of option set 8 are applied once, rule 1 of option set 6 is applied twice and rule 1 of option set 10 is applied three times, whereas all other selected rules are never applied.

4.1 Sifting Metric

Definition 2. *A* ***model reference*** *is defined as a Maude term* `{` μ`,` ω `|` Ψ `}`*, where*

- $\Psi =$ `[1 :` i_1`(`j_1`)] [&>` ... `[&> [`n `:` i_n`(`j_n`)]` *is a list of* n *option references, one for each option set;*
- ω *is the number of times the model is invalidated by a target event;*
- μ *is the measure of model validation, which is calculated using the following* sifting metric

$$\mu = \frac{(1 - c \cdot \frac{\omega}{\eta}) \cdot \sum_{k=1}^{n} j_k}{\sigma}$$

in which

- η *is the number of target events against which the model is checked during simulation;*
- $c \leq \eta$ *is a positive constant that assigns a fixed weight to all invalidations;*

- σ *is the number of environmental state changes due to the occurrences of environmental events.*

For example, the model reference corresponding to the list of option references represented as Maude term (1) above is

```
{ 8/17 , 0 | [1 : 1( 0)][&> [2 : 1(0)][&> [3 : 1(1)] [&>
             [4 : 2(1)][&> [5 : 1(0)][&> [6 : 1(2)] [&>
             [7 : 1(0)][&> [8 : 1(1)][&> [9 : 2(0)][&> [10 : 1(3)] }
```

where $\mu = 8/17 \simeq 0.47$ is the value provided by the sifting metric and $\omega = 0$ denotes that the model was never invalidated.

Definition 3. *A* ***refinement*** *is a set of model references* `{ μ, ω | Ψ }`*, such that* $\mu > 0$.

The probability that the model is invalidated by a target event is given by $\frac{\omega}{\eta}$. Thus, for $c = 1$, term $1 - c \cdot \frac{\omega}{\eta}$ gives the probability that the model is not invalidated by a target event. The higher such a value, the more accurate the model. The role of constant c is to take *noise* into account. We choose $c < \eta$ in situations in which noise may invalidate correct models. In this case the higher the noise, the smaller should c be chosen. Unfortunately, the sifting metric cannot help in situations in which noise may validate incorrect models. In the absence of noise we could choose $c = \eta$, which includes in the refinement only the models that are never invalidated, but would exclude possible models invalidated by noise. In our previous work [4] we sifted out a model at the first invalidation. This can be reproduced in our more general framework by choosing $c = \eta$.

Term $(\sum_{k=1}^{n} j_k)$ gives the total number of transition rules applications. Since transition rules are only applied after an environmental state changes due to the occurrence of an environmental event, term $(\sum_{k=1}^{n} j_k)/\sigma$ gives the probability of application of transition rules after an environmental change. This provides a measure of how frequently the model evolves in response to environmental changes. Therefore metric μ combines the fact that the model is validated by the target events (term $1 - c \cdot \frac{\omega}{\eta}$) with the responsiveness of the model to environmental events (term $(\sum_{k=1}^{n} j_k)/\sigma$).

Table 3 shows the results of applying refinement mining to our case study. In absence of noise, for $c = \eta = 10$, the refinement comprises 32 out of 128 option-free models, i.e. all option-free models that are not invalidated ($\omega = 0$). However, for $c = 9 < \eta$, the number of option-free models in the refinement increases to 96 and, for $c = 1 < \eta$, the refinement actually comprises all 128 option-free models, with some models with 1 invalidation ($\omega = 1$) being better ($\mu = 9/17 \simeq 0.529$) than all models with 0 invalidations ($\omega = 0$, $\mu = 8/17 \simeq 0.471$). In fact, the negative effect of ω on metric μ is directly proportional to constant c and inversely proportional to the number η of target events. Thus, in our case study, the small number of target events $\eta = 10$ makes the effect of ω decrease quickly when c decreases: a small η cannot effectively validate the model against noise.

Table 3. Results from applying refinement mining to the plausible model in Fig. 3 (128 option-free models, $\eta = 10$ and $\sigma = 17$).

refinement	no. models	μ	ω	$\sum_{k=1}^{n} j_k$
32 models for $c = 10 = \eta$	**32**	$8/17 \simeq 0.471$	0	8
	64	0	1	8 or 10
	32	$-10/17 \simeq -0.588$	2	10
96 models for $c = 9 < \eta$	**32**	$8/17 \simeq 0.471$	0	8
	32	$1/17 \simeq 0.059$	1	10
	32	$4/85 \simeq 0.047$	1	8
	32	$-8/17 \simeq -0.471$	2	10
128 models for $c = 1 < \eta$	**32**	$9/17 \simeq 0.529$	1	10
	32	$8/17 \simeq 0.471$	0	8
	32	$8/17 \simeq 0.471$	2	10
	32	$36/85 \simeq 0.424$	1	8

5 Conclusion and Future Work

In this tool paper we have presented the architecture of FormalMiner and described how refinement mining is carried out on a real case study from ecology. With respect to our previous work [4,5], we emphasised on practical aspects such as the *sifting metric*, which is used to evaluate the validity of alternative option-free instances of the model. In particular, the sifting metric was simplified by removing the dependency from the number of option sets. The definition of refinement was also simplified by removing the notion of a threshold. These simplifications led to a more informative output of FormalMiner, which allows us to improve the refinement mining strategy as discussed below.

In our previous work [5] we discussed the complexity of model mining and presented the results of timing testing. In term of complexity, we observed that the number of transition rule applications grows linearly with respect to the size of the dataset. Moreover, the complexity of the refinement mining algorithm is subquadratic with respect to both the size of the dataset and the number of option-free models.

The main problem is that the total number of option-free models grows exponentially with the number of options per option set. In this sense the complexity is exponential with respect to the level of option-related non-determinism. It is therefore a good strategy to apply refinement mining repeatedly on plausible models which contain only a small number of options per option set. In this way, we may get some ideas on which combinations of transition rules are incompatible and then test them in separate runs of the refinement mining algorithm. In particular, in Sect. 3.4, we suggested to initially include two rules corresponding to what we expect to be reasonable lower and upper bound limits for the persistence of a domain state. In this way, after running refinement mining once, we can select the models that are part of the refinement (those for which $\mu > 0$) and then combine them in a new plausible model such that

1. when only one of the two rules of a pair with lower and upper bound limits was excluded from the refinement, while the other rule was applied at least once, a modified version of the former rule with limit closer to the bound of the latter rule may be introduced;
2. when both rules of a pair with lower and upper bound limits were included in the refinement, the more stringent rule may be eliminated;
3. when both rules of a pair with lower and upper bound limits were excluded from the refinement, either their bound limits may be relaxed or both rules may be eliminated, depending on whether or not we find these two rules consistent with the rest of the plausible model.

Obviously pairs of rules that are included in the refinement but are never applied require further data in order to be validated.

For example, let us consider the option-free model defined by the list of option references reppresented by Maude term (1) in Sect. 4, which is part of the refinement for any choice of constant c. Since rule 2 was included in and rule 1 was excluded from option set 4 in the refinement, following item 1 above, we may make the lower bound closer to the upper bound and run the refinement mining again, thus finding out that for lower bounds between 6 and 9 also rule 1 is included in the refinement. This means that we can replace the two rules of the pairs with just one rule with a persistence of 6 days as a condition. Since both rules 1 and 2 of option set 8 were included in the refinement and applied once, following item 2 above, we may eliminate rule 2, which requires a persistence of 4 days and is therefore more stringent than rule 1. Rules 3 and 4 of option set 4 were both excluded from the refinement. Following item 3 above, they may be eliminated due to their inconsistency with rules 1 and 2 of option set 4. Finally, rules in option sets 5, 7 and 9, which are never applied, require further target events in order to be validated.

In our future work, we intend to automate the combination of option-free models that are comprised by the refinement into a new plausible model. Moreover, we are planning to apply refinement mining to other case studies from ecology and from other areas [2], such as human-computer interaction [3,6] and emergency management [16].

References

1. Basuki, T.A., Cerone, A., Barbuti, R., Maggiolo-Schettini, A., Milazzo, P., Rossi, E.: Modelling the dynamics of an aedes albopictus population. In: Proceedings of the AMCA-POP 2010, volume 227 of Electronic Proceedings in Theoretical Computer Science, pp. 37–58 (2010)
2. Cerone, A.: Process mining as a modelling tool: beyond the domain of business process management. In: Bianculli, D., Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9509, pp. 139–144. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-49224-6_12
3. Cerone, A.: A cognitive framework based on rewriting logic for the analysis of interactive systems. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 287–303. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41591-8_20

4. Cerone, A.: Refinement mining: using data to sift plausible models. In: Milazzo, P., Varró, D., Wimmer, M. (eds.) STAF 2016. LNCS, vol. 9946, pp. 26–41. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50230-4_3
5. Cerone, A.: Model mining – integrating data analytics, modelling and verification. J. Intell. Inf. Syst. (2017). https://doi.org/10.1007/s10844-017-0474-3
6. Cerone, A.: Towards a cognitive architecture for the formal analysis of human behaviour and learning. In: Mazzara, M., et al. (eds.) STAF 2018 Workshops. LNCS, vol. 11176, pp. 1–17. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-04771-9_17
7. Češka, M., Dannenberg, F., Kwiatkowska, M., Paoletti, N.: Precise parameter synthesis for stochastic biochemical systems. In: Mendes, P., Dada, J.O., Smallbone, K. (eds.) CMSB 2014. LNCS, vol. 8859, pp. 86–98. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12982-2_7
8. Clavel, M., et al.: The maude 2.0 system. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44881-0_7
9. Gries, D., Scheneider, F.B.: A Logical Approach to Discrete Math. Springer, Heidelberg (1993). https://doi.org/10.1007/978-1-4757-3837-7
10. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of the POPL 2011 ACM SIGPLAN Notices, vol. 46, pp. 317–330. ACM (2011)
11. Koksal, A.S., Pu, Y., Srivastava, S., Bodik, R., Fisher, J., Piterman, N.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of the POPL 2013 ACM SIGPLAN Notices, vol. 48, pp. 469–482. ACM (2013)
12. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. Theor. Comput. Sci. **285**(2), 121–154 (2002)
13. Mukala, P.: Process Models for Learning Patterns in FLOSS Repositories. Ph.D. thesis, Department of Computer Science, University of Pisa (2015)
14. Paoletti, N., Yordanov, B., Hamadi, Y., Wintersteiger, C.M., Kugler, H.: Analyzing and synthesizing genomic logic functions. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 343–357. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_23
15. Rozinat, A., van der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst. **33**(1), 64–95 (2008)
16. Shams, F., Cerone, A., De Nicola, R.: On integrating social and sensor networks for emergency management. In: Bianculli, D., Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9509, pp. 145–160. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-49224-6_13
17. Solar-Lezama, A., Rabbah, R.M., Bodik, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: Proceedings of the PLDI 2005 ACM SIGPLAN Notices, vol. 40, pp. 281–294. ACM (2005)
18. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: Proceedings of the POPL 2010 ACM SIGPLAN Notices, vol. 45, pp. 313–326. ACM (2010)
19. van der Aalst, W.M.P., de Beer, H.T., van Dongen, B.F.: Process mining and verification of properties: an approach based on temporal logic. In: Meersman, R., Tari, Z. (eds.) OTM 2005. LNCS, vol. 3760, pp. 130–147. Springer, Heidelberg (2005). https://doi.org/10.1007/11575771_11
20. van der Aalst, W.M.P., Stahl, C.: Modeling Business Processes: A Petri Net-Oriented Approach. The MIT Press, Cambridge (2011)