International Environmental Modelling and Software Society (iEMSs)
2010 International Congress on Environmental Modelling and Software
Modelling for Environment's Sake, Fifth Biennial Meeting, Ottawa, Canada
David A. Swayne, Wanhong Yang, A. A. Voinov, A. Rizzoli, T. Filatova (Eds.)
http://www.iemss.org/iemss2010/index.php?n=Main.Proceedings

Modelling the environment using graphs with behaviour: do you speak Ocelet?

P. Degenne[a], A. Ait Lahcen[b], O. Curé[c], R. Forax[c] D. Parigot[b] D. Lo Seen[a]

[a]*CIRAD - UMR TETIS, Montpellier France (pascal.degenne@cirad.fr),(danny.lo_seen@cirad.fr)*

[b]*INRIA, Sophia Antipolis France (ayoub.ait_lahcen@inria.fr),(didier.parigot@inria.fr)*

[c]*Institut Gaspard Monge - Université Paris Est, Marne la Vallée France (ocure@univ-mvl.fr),(forax@univ.mvl.fr)*

Abstract: Environmental modelling often implies defining elements that relate and interact with each other, in a system that evolves with time. Ocelet is a domain specific environmental modelling language that was designed around a limited set of key concepts chosen to help modellers focus on their model, while leaving the implementation to an automatic code generation phase. Here, we focus more specifically on a concept called Relation in Ocelet that allows to build graphs that describe which elements of the model interact, and how. It is designed to be used in combination with two other main concepts: Entities (elements of the model) and Scenarios (describing the temporal evolution). Every Ocelet Relation can express one specific point of view of a system and several Relations can be combined to integrate different points of view in the same model. By its diversity, points of view convey expressive power: with different expert views on a system, at different spatial scales, or an environment sensed by its different components. Moreover, in this versatile design, a Relation defined for one specific model can be reused in a different modelling context. Libraries of generic interaction behaviours can thus be developed for efficient and reliable modelling practices. An example is given to illustrate how interaction graphs can be built, manipulated, and reused using Ocelet. Finally, we give insight into the code generation phase that produces the simulator.

Keywords: Domain Specific Language, Landscape, Ocelet, Interaction graph, Relation

1 INTRODUCTION

Interactions are at the heart of most environmental studies and the way they are modelled is strongly characterised by the modelling formalism used. In System Dynamics (SD), interaction operations can be performed between any of the components of the system, but the latter are generally not spatially represented. In Cellular Automata, interaction operations are defined by a set of rules applied to the cells of a tessellation, and every cell can only interact with its immediate neighbours. Agent based models (ABM) are more versatile and agents are allowed to interact with each other according to a set of locally defined rules. In the case of Discrete Events and Object Oriented design, there is no constraint on how the elements of a model can interact with each other, but this comes at the expense of complex programming for the modeller. When building environmental simulation models, Domain Specific Languages (DSL) are an interesting compromise between expressive power and a steep programming language learning curve. However, to our knowledge only few attempts have been made in this domain so far. Fall and Fall [2001] proposed SELES in which a landscape modelling language is used to define landscape states in the form of grid layers and a set of landscape events to make the model evolve with time. Gaucherel et al. [2006] designed L1, a DSL based platform for simulating the evolution of patchy landscapes. CAOS (Grelck et al. [2007]) is a DSL specialized in parallel simulation of cellular

automata where space is represented in the form of a grid.
In these examples, a major constraint for the modeller is that space representation is imposed. This is not the case for Ocelet, a DSL developed for landscape and environmental modelling and simulation (Degenne et al. [2009]). Ocelet was built by carefully selecting the most elementary concepts needed for spatio-temporal modelling. It offers the possibility of creating new modelling primitives based on these concepts, and to incrementally assemble them for developing complex models. The result is a DSL with increased expressivity where the complexities of implementation are left to a code generator. In this article we emphasise on the concept of interaction graphs, as used in Ocelet under the name of *Relation*. In the next section we explain how and why interaction graphs can be used to express relations in a model. We then briefly present the Ocelet modelling language, the associated code generation and development framework, and also how Ocelet can be used with an ontology language. Finally, based on an example, we present how a relation can be built, manipulated and reused within Ocelet.

2 RELATIONS AS INTERACTION GRAPHS

An interaction graph not only defines *who* are in relation (graph structure) but also *how* the elements relate (behaviour). When modelling the environment, we consider that working directly on interaction graphs can be useful for at least two reasons. First, acting at the most elementary level of the underlying data structure (a set of dynamic graphs) allows manipulating in a similar way different kinds of relations (aggregations, spatial, functional, ...) Second, the state of the model at any given time can be analysed using graph analysis algorithms to extract topological characteristics that emerge during the simulation. These may reflect some specificities of the model that would hardly be visible otherwise. Such analysis algorithms have for example been developed by Batagelj and Mrvar [1998], Fuller and Sarkar [2006] or Saura and Torne [2009]. In this section, we describe the structure and the dynamic nature of relations or interaction graphs. We then explain how they are made to express behaviour, and how a relation developed in one case can be reused in another. Finally we outline how a relation holds the notion of a point of view.

2.1 Graph with dynamic structure and behaviour

Entities of a model can, at a given time, relate to each other in diverse ways. For example, neighbourhood (where two entities are considered neighbours if they are close enough for a given distance function), aggregation (where some entities are considered parts of a larger composite entity), connectivity (where entities can reach each other if a communication route exist between them), influence (where one entity can influence the behaviour of another one) are in fact relations. For each relation, one can build a graph where the nodes are entities and the relations between entities are the arcs.
In many environmental modelling cases the graphs needed are actually hypergraphs (each arc may connect more than two nodes). Such hypergraphs can be built explicitly. For example, if we have several groups of entities connected to each other in the form of simple graphs, one can establish another graph connecting those groups to each other at a broader scale. In that way, it is possible to consider the behaviour of entities within a group as well as between groups. But one can also build hypergraphs implicitly. For example, in the case of a spatial relation where an agricultural parcel is linked to each of its borders by one n-node arc, a graph is built using arcs linking more than two nodes. Such n-node arcs based graphs are de facto hypergraphs. Using n-node arcs can be a way to simplify the graph structure we have to manipulate in the model. Another aspect to take in consideration when modelling with interaction graphs is their dynamic nature. During a simulation, some entities can be added to the model, others can disappear, and individual relationships can be established or removed. This means that the interaction graphs are dynamic, with evolving numbers of nodes and arcs, and have changing graph topologies.
Attached to the graph are semantics that specify what happens between the linked entities when they do interact: the kind of information they exchange, the actions one performs on the other, the effects produced by the interaction on the entities and on the arcs involved. In many types of environmental models, attaching behaviour to an interaction graph is not straightforward. Sometimes the graph structure is implicit (e.g. cellular automata based on tessellations) and only the be-

haviour is specified. The programming work is then reduced but the specification of the behaviour is seriously constrained by the implicit graph structure. In other cases where the graph structure is more versatile, a greater power of expression is obtained but the lack of adapted tools makes the programming work difficult. In order to get the best of both solutions, it would be necessary to have ways to manipulate the graph structure and attach the behaviour semantics directly on that structure using one same appropriate modelling concept.

2.2 Roles and re-usability

It is rare when an environmental model is original in all its parts. The most common situation is to have some parts of the model that are similar to other already existing models. Re-usability has been a key concern in software development and modelling tools as well. In the case of behaviours attached to relation graphs, two situations can be considered:

Re-usability of a relation graph topology: It can be interesting to have ready made relation graph structures such as the 3-neighbours situations found in triangulated irregular network, the 4 or 8-neighbours situations found in grids, or also star and circular shaped relationships just to name a few. Based on the well known characteristics of such structures, one could imagine a modelling tool that provides optimized implementations for them to be used in different models.

Re-usability of an attached behaviour: In that case we wish to be able to reuse the definition of *how* entities interact with each other when they do, in different modelling situations. To make the behaviour definition adaptable to a different context, the interaction should not be specified using the *entities* relating with each other but using the *role* they play. It would then be possible to attach a behaviour definition to a different relationship graph where entities are able to play similar roles. It also means that a behaviour defined once can be instantiated several times, on different graphs, even in one same model.

Finally, it can be noted that by designing a modelling tool with re-usability concerns as described above, it becomes possible to build sub-model libraries (named *primitives* in Ocelet) and make them available for a modellers community.

2.3 Modelling your point of view

At least two cases can be identified where the notion of point of view can take the form of semantics attached to a graph. First, when specialists of several different fields work on the same environmental model, they may share the same entities but need to describe interactions between these entities differently according to their own expert view. The nodes of the graph could be shared, but the arcs and the behaviour attached to those arcs would reflect their different points of views on the model. Second, it happens that different entities of a model have different points of view on their environment and would then have to interact accordingly with that environment (see example section 4). Here again the nodes of a graph could be shared but the arcs and the behaviour attached to those arcs could be specific to every point of view.

3 THE OCELET LANGUAGE

The main concepts of the Ocelet domain specific language are presented here. The concept of Relation in Ocelet is then explained in more detail, showing how it addresses most of the concerns discussed in section 2. Models written in Ocelet are translated into a general purpose language through a code generation phase before a simulation can be run. The code generation aspect and the associated development platform, and the use of Ocelet with an ontology language are also briefly presented here.

3.1 Key concepts of the language

Three main concepts are at the core of the language. They are named *Entity*, *Relation* and *Scenario*:

- **Entity:** Entities are the basic elements that can be linked together to build a model. An entity may contain other entities, and is then called a composite entity. Entities that do not contain other entities are atomic entities. Entities have properties that can be used to reflect their state. Entities also provide *Services* which are published functions that can be called locally or remotely. The concept of entity is close to and inspired by the definition of Components in Service Oriented Architectures (Szyperski [1998]).
- **Relation:** A relation is a connection between two or more entities that provide and require compatible services. It defines the nature of interactions between these entities and provides services for the activation of those interactions. This concept in detailed in paragraph 3.2
- **Scenario:** A scenario is a sequence of actions composed of service calls or relation expressions within a model or composite entity. A scenario is activated for a period of time. Therefore the scenario expresses most of the temporal behaviour of a model or a composite entity.

In addition to these concepts, we have to mention a special category of atomic entities that is named *Datafacer*. A Datafacer is an atomic entity specialized in data access. The Datafacer provides different mechanisms for data persistence. Its implementation code can be written in a programming language other than Ocelet in order to optimize data access performances for every type of data sources that a model can integrate. Other concepts such as primary types (number, boolean, ...), tests and control instructions, are also available in Ocelet but they are not different from those of other programming languages such as C or Java, and therefore do not require specific descriptions here. It is also important to mention that even though Ocelet is not strictly an object oriented language, the elements (Entities and Relations) of a model have to be defined first and then instantiated within a Scenario allowing to create as many individual copies as necessary.

3.2 Relations are interaction graphs in Ocelet

The Relation concept as defined in Ocelet is an interaction graph very close to what was discussed in section 2: it contains the information of *who* is in interaction and also of *how* they interact. As Relations have semantics attached to the arcs of their graph, they are constrained by the type of entities that can be linked. The definition of a Relation has to specify the role played by the different entities involved, like for example:

```
relation RelationName[roleA, roleB] {...}
```

The statement above defines a Relation of the most common kind: every arc of the graph links two nodes. The nodes will be entities; one entity playing role A and the other role B. Once defined, the Relation must be instantiated, and which entities playing role A and role B must also be stated for that instance:

```
myInstance = RelationName[EntityA, EntityB];
```

The fact that Relations are defined using roles makes them reusable in different contexts. A Relation carefully designed with genericity in mind could then be used and adapted for several different models. To establish connections and actually build the graph, the predefined `connect()` and `disconnect()` services are available. For example, `myInstance.connect(lake,river)` implies that `lake` is an instance of `EntityA`, `river` is an instance of `EntityB` and an arc will be added to the Relation graph between them. Ocelet allows to define Relations holding hypergraphs directly by specifying more than two roles in the declaration statement, like for example: `relation RelationName[roleA, roleB, roleC, roleD] {...}`. The *how* part is defined in the form of services that the modeller can write to precisely describe what happens when the entities interact. The services are written in the declaration of the Relation, like in:

```
relation RelationName[roleA, roleB] {
 service foo() { roleA.doSmthg(); roleB.setVal(roleA.getVal()); }
}
```

The definition above implies that the entities playing `roleA` for that Relation must provide the two services `doSmthg()` and `getVal()`, while the entities playing `roleB` must provide the service `setVal()`. `getVal()` and `setVal()` must also return and accept compatible types. These are verified when the Relation is instantiated. One important point to note is that only one call to the `foo()` service is necessary to activate all the arcs of the relation graph.

3.3 Code generation and development platform

Models written in Ocelet are not compiled directly, but are translated into a general purpose programming language first. For the target language we use Java, and the Ocelet development environment is integrated into the Eclipse platform in the form of Eclipse plug-ins. The code generated is based on components such as defined by Szyperski [1998] to better separate the functional (the code related to what the model is about) and non-functional (the components discovery and communication mechanisms) aspects, as well as on the Service oriented computing (SOC) paradigm as described by Papazoglou and Georgakopoulos [2003]. SOC is a paradigm that uses services as fundamental elements for developing applications. The main purpose of this approach is to introduce the minimum dependencies between software bricks to promote their re-usability and their dynamic discovery and combination at run time.
For every Ocelet element, description files are generated that describe the services provided and required by that element. According to the description files, the component generator will produce non-functional code which will manage external communications based on sending or receiving messages synchronously or asynchronously. The assembly of components for a given application is not necessarily known at the start of a simulation and may change dynamically over time. Such a component framework provides an extensibility mechanism allowing the clear separation between the business logic and context-aware service interactions. The code generation allows modellers using Ocelet to take advantage of that dynamic execution environment without having to deal with implementation details.

3.4 Mapping Ocelet's concepts to an ontology language

The limited set of key concepts present in Ocelet have been selected to ease the work of the modellers but also to permit a mapping to an ontology language. We have chosen an ontology language, namely OWL2, which is based on Description Logics (Baader et al. [2003]) and recommended by the W3C's Semantic Web working group. The mapping is relatively straightforward and enables to transform automatically one language to the other. Hence modellers either have the possibility to specify their models directly in OWL2, using an editor like Protégé, or to specify the model directly in Ocelet, using an Eclipse plug-in.
Several advantages are provided by such an OWL2 serialisation of an Ocelet model. The main one consists in enjoying state of the art reasoning tools on standard inference services, i.e. detecting and repairing ontology inconsistencies and classifying the set of entities of a model. A second advantage corresponds to proposing an efficient storage and query solution for model instances. This aspect is particularly important considering that modellers will simulate temporal situations in any possible order. This feature will enable modellers to analyze the data stored in these data management systems directly with a query language or through applications using API for these query languages.

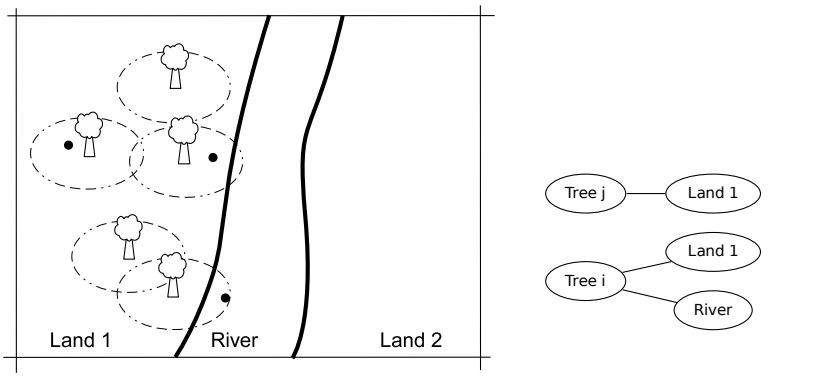
4 RELATIONS ILLUSTRATED

The development of the Ocelet DSL was based on the analysis of several very different modelling situations that are studied by our partners in different fields, such as the ecology of mangroves in French Guiana, the epidemiology of the Rift Valley Fever in Senegal, agricultural land dynamics

in France and West Africa, and forest landscape dynamics in South India. But for the purpose of this paper, in order to particularly illustrate the use of Relations in Ocelet, we have used a simple and didactic example of a modelling situation.

4.1 Neighbourhood from a tree point of view

In this example, the objective is to model the progressive colonization of trees on a given landscape. A first version of the model contains two pieces of land crossed by a river. The trees growing on one side (Land 1) spread their seeds around their close neighbourhood (Fig. 1(a)). In the model, we define a relation named `DropSeeds` to connect every tree to the land or river entities that are close enough to receive their seeds (Fig. 1(b)), and to specify what happens when seeds are dropped. The seeds falling into the river are considered lost, and in the present case, as the river is too large, the trees cannot spread their seeds on the other side of the river.



(a)

(b) Sample graph for the dropseed1 Relation

Figure 1: Trees dropping seeds in their close neighbourhood

The Tree entity is entirely defined using Ocelet. It has position coordinates and a service `getDroppedSeedLocations()` that returns the positions of the seeds it produces. The Land and River entities are also defined using Ocelet but their shape are defined by data sources. Those data sources are accessed by an appropriate Datafacer. In particular, the Datafacer has a service that can provide the distance between a shape and a given location. It does not matter if internally the shapes are stored in vector or grid format, or if they are in a file or a database. It is the purpose of the Datafacers to hide the underlying data implementations, and they are expected to be optimized for the kind of data source they are dealing with.
The DropSeeds Relation is defined as follows :

```
relation DropSeeds[seedEmitter, seedReceiver] {
  service drop() {
    group[Position]
      seedsPos = seedEmitter.getDroppedSeedLocations();
    for (pos in seedsPos) {
      seedReceiver.acceptSeedAt(Pos);
    }
  }
}
```

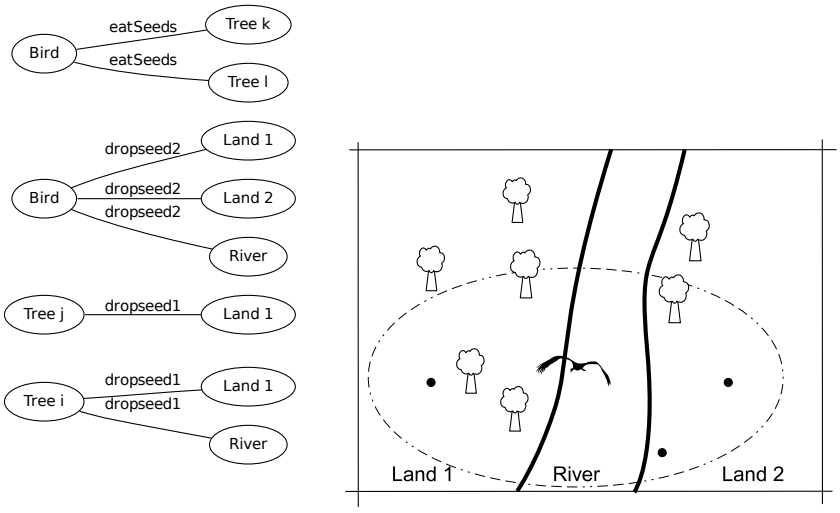
It can be noted that the DropSeeds Relation is defined not using Tree and Land entities, but using

the role they can play for that Relation: `seedEmitter` and `seedReceiver`. At initialization, the Relation is instantiated with a statement:

```
dropseed1 = DropSeeds[Tree,Land];
```

That statement specifies that for `dropseed1` the `seedEmitter` role will be played by Tree entities and the `seedReceiver` role will be played by Land entities. Then, tree - land connections are established using calls to `dropseed1.connect()`. The resulting graph held by the `dropseed1` instance of the Relation will be similar to the example shown in Fig.1(b). During the simulation, a scenario calls the `dropseed1.drop()` service when necessary. One such call is enough for the code of that service to be executed on every arc belonging to the `dropseed1` Relation graph. That service takes a list of seed positions from the `seedEmitter` and propose to the `seedReceiver` to accept those seeds if they are located within its area. The seed location tests are performed by the `seedReceiver`, through a Datafacer in the case of Land entities.

4.2 Neighbourhood from a bird point of view



(a) Sample graph for all Relation instances (b)

Figure 2: Adding a bird's neighbourhood point of view allows trees to spread across the river

Now, let us introduce in the model a species of birds that eat the seeds of the trees and drop them somewhere in their living area. A new Relation named `EatSeeds[seedProvider, seedEater]` can be defined to specify how the birds would choose the trees or seeds in a given area. Details of that new Relation are not given here as they are in principle similar to the DropSeeds Relation described above.
The same DropSeed Relation can thus be reused to express the bird's neighbourhood point of view. For that, a second instance of DropSeeds is created: `dropseed2 = DropSeeds[Bird,Land];` But, as a prerequisite for playing the `seedEmitter` role in the Relation, the Bird entity must provide the `getDroppedSeedLocations()` service. For the birds, both Land 1 and Land 2 are within reach, and the dropseed2 graph reflects that situation. Fig.2(a) shows both instances of the DropSeeds Relation. With this new point of view in the model, the trees are likely to have some of their seeds dropped on the other side of the river, thus allowing extension of the forest in that new area (Fig.2(b)).

5 CONCLUSION AND PERSPECTIVE

Interactions between landscape elements are essential in environmental modelling. For this reason, we have taken special care in designing a versatile way of modelling them when developing our DSL-based approach. Ocelet has been designed to take advantage of modelling all different categories of relationship, including spatial and functionnal using one same programming paradigm (graphs with behaviour) and to offer the tools necessary for considerably simplifying what would otherwise be a tedious programming work. In Ocelet, a relation can be defined very simply by an interaction graph that both describes *who* is interacting and *how*. The behaviour attached to a graph can be activated on all the arcs at once with only one service call. That behaviour can also easily be made reusable for application in a different modelling context. This relatively simple way of defining a relation has been found to allow modelling a large variety of situations, and reusable primitives are being built with Ocelet to ease the modelling process. We believe that the possibilities offered by integrating network analysis features to Ocelet's relations, the capacity of building reusable modelling bricks, and to relate them with an ontology language, are promising research subjects to be investigated.

ACKNOWLEDGMENTS

This work was supported (in part) by the *Agence Nationale de la Recherche* (ANR) under Project No. ANR-07-BLAN-0121 (STAMP: Modelling dynamic landscapes with Spatial, Temporal And Multi-scale Primitives).

REFERENCES

Baader, F., D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The description Logic Handbook : Theory, Implementation and Applications*. Cambridge University Press, 2003.

Batagelj, V. and A. Mrvar. Pajek - program for large network analysis. *Connections*, 21(2):47–57, 1998.

Degenne, P., D. Lo Seen, D. Parigot, R. Forax, A. Tran, A. A. Lahcen, O. Curé, and R. Jeansoulin. Design of a domain specific language for modelling processes in landscapes. *Ecological Modelling*, 220(24):3527 – 3535, 2009. Selected Papers on Spatially Explicit Landscape Modelling: Current practices and challenges.

Fall, A. and J. Fall. A domain-specific language for models of landscape dynamics. *Ecological Modelling*, 141(1-3):1 – 18, 2001.

Fuller, T. and S. Sarkar. Lqgraph: A software package for optimizing connectivity in conservation planning. *Environmental Modelling & Software*, 21(5):750 – 755, 2006.

Gaucherel, C., N. Giboire, V. Viaud, T. Houet, J. Baudry, and F. Burel. A domain-specific language for patchy landscape modelling: The brittany agricultural mosaic as a case study. *Ecological Modelling*, 194(1-3):233 – 243, 2006.

Grelck, C., F. Penczek, and K. Trojahner. Caos: A domain-specific language for the parallel simulation of cellular automata. *LNCS*, 4671:410–417, 2007. Parallel Computing Technologies (Pact'07), Pereslavl-Zalessky, Russia.

Papazoglou, M. P. and D. Georgakopoulos. Service oriented computing. *Communications of the ACM*, 46(10):24–28, 2003.

Saura, S. and J. Torné. Conefor sensinode 2.2: A software package for quantifying the importance of habitat patches for landscape connectivity. *Environmental Modelling & Software*, 24(1):135 – 139, 2009.

Szyperski, C. *Component software: Beyond object-oriented programming*. ACM Press and Addison-Wesley, 1998.