DEVELOPING A SERVICE-ORIENTED COMPONENT FRAMEWORK FOR A LANDSCAPE MODELING LANGUAGE

Ayoub Ait Lahcen
INRIA, 06902 Sophia Antipolis, France
Mohammed V University-Agdal, LRIT
Laboratory, 10090, Morocco
ayoub.ait_lahcen@inria.fr

Pascal Degenne, Danny Lo Seen
CIRAD, UMR TETIS
34398 Montpellier, France
pascal.degenne@cirad.fr
danny.lo_seen@cirad.fr

Didier Parigot
INRIA, 06902 Sophia Antipolis, France
Didier.Parigot@inria.fr

ABSTRACT
With modeling and simulation, it is possible to study how a system works before trying to predict how it would behave in a variety of situations. However, when modeling landscape processes, issues related to space, time and multiple scales need to be addressed. In order to investigate these issues, a modeling platform based on a Domain Specific Language (DSL) has been developed. One of the main technical challenges of this platform is the ability to build applications with the capacity to themselves dynamically adapt to their environment. In this paper, we present the arguments and motivations behind the choice of the Service-Oriented Computing (SOC) approach when implementing the execution framework of the DSL. The modeling platform is composed of a development environment based on Eclipse IDE, a code generator, and an execution framework. The execution framework, which is the focus of this paper, must meet the constraints set by dynamic landscapes modeling, while capitalizing on the possibilities offered by the SOC approach.

KEY WORDS
service-oriented component framework, landscape applications, service orientation, landscape modeling, DSL

1 Introduction

The modeling of landscapes is useful for the analysis of many important issues which society faces today such as the degradation of natural ecosystems (loss of biodiversity), the emergence and spread of new diseases (changing environmental and climatic conditions), or the uncontrolled urbanization and population migrations (deep social transformations). Such questions, which treat many interacting processes, hide a complexity that is difficult to address otherwise than by modeling. In order to study issues related to space, time and multiple scales, to which the modeling of landscape processes are confronted, a modeling language called Ocelet [1] was developed, and is briefly presented hereafter. In this paper, we discuss in more detail its dynamic execution framework.

In the present case, it is important that the execution model takes into account the aspects of a distributed execution in a ubiquitous environment. Moreover, this type of highly dynamic application must adapt according to its own evolution and change in its execution context. But various business (the domain of landscape modeling) and technical challenges (the management of dynamism and service interactions) complicate the ability to develop such application. In order to meet these constraints, we choose an approach based on a component model [2] to better separate the functional and non-functional aspects, as well as on the service-oriented paradigm. The objective of this paper is to address the above mentioned constraints. It presents an open computing environment for the development of applications in the landscape modeling domain. The proposed service-oriented component framework provides an extensibility mechanism allowing the clear separation between the business logic and context-aware service interactions.

In the next section we present background information about two main aspects of the context in which this work has been carried out: service oriented computing and the modeling of landscapes and their dynamics. Section 3 is a brief description of the landscape modeling DSL Ocelet. Section 4 is where our service-oriented component execution model of the language is described. Finally a case study illustrates the execution model in section 5.

2 Background

2.1 Service oriented computing

SOC [3] is a paradigm that uses services as fundamental elements for developing applications. The main purpose [4] of this approach is to introduce the minimum dependencies between software bricks to promote their reusability and their combination, in order to respond quickly and at low cost to new business needs. This reduction of dependence allows these different bricks to

evolve separately. SOC is based on three actors: i) the Service Provider publishes a service on a Service Broker and generates a description of the service which specifies both the available operations and their invocation mode; ii) the Service Broker contains references to services; and iii) the Service Consumer discovers the services available and their descriptions obtained by running a search. It then establishes a connection with the provider and invokes the service operations.

SOC is an academic initiative that aims at extending service-oriented architecture (SOA) to manage and compose services in a flexible manner and it is organized on three levels:

- The first covers SOA with its minimum functions: publication, discovery and binding services.
- The second is the dynamic services composition. It is responsible for adapting the application at runtime (adding new features; control the execution of the component services and manage dataflow among them; adapting to a new context).
- The third covers the management functions necessary for the overall supervision of applications. It may permit complete visibility into individual business transactions, and deliver application status notifications.

The SOC paradigm allows the development of modular and dynamic applications by supporting loose coupling and late binding between the software bricks. However, these aspects and context-awareness are generally managed by the programmer and are implemented in the business logic. In section 4, we describe how the SOC concepts (except the third level functionalities) can be integrated into a component model to separate these aspects and the business logic.

2.2 Landscape modeling

Given the spatial, temporal and multi-scale issues raised, the modeling of dynamic landscapes has to draw from a variety of paradigms or formalisms - System Dynamics (SD), Discrete Event (DE), Cellular Automata (CA), Agent-Based (AB) and Geographic Information Systems (GIS) [5, 6, 7, 8]. While in GIS the challenge of properly handling Time has been a subject of active research for the last few decades [9, 10, 11, 12], the formalisms that consider Time first (i.e. SD, DE) face the opposite limitation with spatial information [13, 14]. Overall, these issues are being studied in different research communities, using various methodological approaches, as well as in many other thematic areas (e.g. ecology, epidemiology and urban dynamics).

The difficulties that modelers face when working from conceptual models of dynamic landscapes to their simulation on a computer also cannot be ignored. For this reason, we have chosen to develop a DSL that would allow domain experts to concentrate on the conceptual model, while leaving to an associated software tool the transformation of the model into an implementation that runs on a computer. However, in this domain, spatial, temporal and multi-scale issues are still actively being studied. A DSL that can support research on modeling processes in landscapes therefore needs to be flexible, and especially so at the very basic level where landscape features and their interactions are defined. Thus, strong requirements are set on the DSL in terms of expressiveness and ease of use.

An interesting parallel can be made between, on one hand, software components and services within SOA, and on the other, landscape entities and their interactions that need to be modeled. Interacting features in a landscape in many aspects behave like communicating software components, and it is not surprising that many notions used when modeling processes occurring in landscapes, such as dynamics, delays, events, fluctuations, response or agent behavior, are also present in the SOC paradigm. A DSL has therefore been developed for experimenting the modeling of a variety of landscape situations that takes advantage of the flexibility offered by component-service programming.

3 The Ocelet Modeling Language

Ocelet is a landscape modeling language defined according to design principles of a DSL [15]. It must provide the modeling concepts adapted to the simulation of dynamic evolution processes in the landscape. This modeling language is instrumented by tools that are able to handle the non-functional requirements with an automatic code generation. This generation offers greater ease to modelers in the design and allows them to focus only on the functional part of modeling. Around this language, a modeling framework (see Figure 1) has been defined, composed of: i) an environment for model building and analysis that is able to verify that the specification written in Ocelet is correct ii) a code generator, and iii) a dynamic execution framework based on component-oriented programming approach [16].

3.1 Overview of Ocelet language

In this subsection, we briefly present the Ocelet language, but a more complete description can be found in [1], with more focus on landscape modeling aspects. Ocelet is designed around five key concepts:

- **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.

- **Service:** A service is an operation defined by an entity or relation; it has a name, parameters and a possible result. There are two types of services: i) a service provided is defined within an entity, and ii) a service required is invoked within the entity and supplied by another.

- **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.

- **Scenario:** A scenario is a sequence of actions composed of service calls or relation expressions within a composite entity. A scenario is activated for a period of time. Therefore the scenario expresses the spatial and temporal internal behavior of a composite entity.

- **Datafacer:** A datafacer is an atomic entity specialized in data access. The datafacer provides different mechanisms for data persistence.

Other concepts such as properties, attributes and arguments are also used, but they do not require specific descriptions. The relation concept allows a connection between two entities where the semantics are described by the services of the relation. In Section 5, the example of a predation relation between predator and prey shows that the logic of this connection is contained in the relation itself. This is important for these relationships to be reusable.

4 A dynamic service execution framework

4.1 Overall architecture

It is expected that under certain circumstances, applications created with Ocelet would need to run in distributed mode, in a ubiquitous environment.

In this context Ocelet entities must be able to communicate with each other through the network. In addition they must be able to adapt according to their own evolution and context. We say that the application (architecture) is dynamic [15, 17] . To meet these constraints, we adopted a component approach that comes from our service-oriented architecture [16, 18, 19]. More specifically, the execution environment of Ocelet is based on this approach. For each component, a file describing the service (provided and required) is used in order to automatically manage the context-aware service interactions.

The principle is to produce for each Ocelet concept (Entity, Relation, Scenario and Datafacer) a corresponding component. Specifically, each concept is translated into two files that will help build that corresponding component. The first file contains only the business code extracted from the definitions of Ocelet services. The second contains a description of the services provided or required by the concept. The component will only use the services of this file to communicate to the outside. According to the description file of the service, the component generator will produce non-functional code which will manage external communications based on sending or receiving messages synchronously or asynchronously. During the execution, a particular component runs by default. This component, called Component Manager (CM) (see Figure 2), supports the creation of components and establishes connections between them. To make the connection between two components, the Component Manager uses two service description files to match the required and provided services for both components. This matching works both ways.

After the connection process, the two components interact with each other directly without going through the Component Manager. The advantage of this environment is the dynamic aspect of the connections between components. In fact, during execution, each component can request to be connected to another component, if it wishes. The assembly of components for a given application is not necessarily known at the start of a simulation and may change dynamically over time. The components are autonomous and independent. The concept of composite component (containing a set of component) is present in this architecture.

We believe that this architecture makes possible the development of ubiquitous applications. Developing such application on top of a service-oriented component model enables the management of the dynamic context execution. The functional and non-functional aspects are separate. Thus coding becomes easy to every application developers. The next subsections presents different aspects of this architecture in more details.

4.2 Service-oriented component framework

As presented in [20], service-oriented component model help developers to build SOC applications. The motivation to use components to build applications comes forth from the necessity to separate SOC aspects and non-functional requirements from the business logic. To design and implement complex applications, one must take into account standards, code distribution, deployment of components and reuse of business logic. To cope with these changes, applications need to be more open, adaptable and capable of evolving. We present in this section a service-oriented component framework based on generators associated with domain-specific languages for component and deployment description.

The principal goal of this framework is to propose a tool which demonstrates that, with new development methods, it is possible to produce more quickly open and

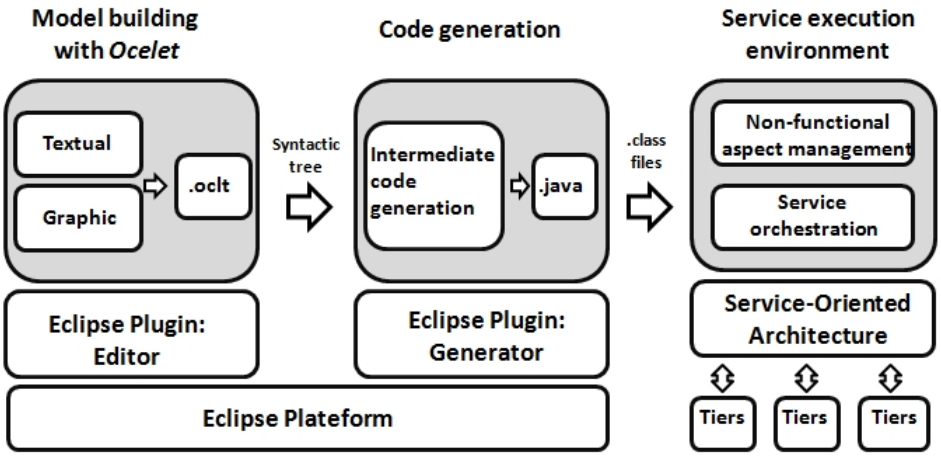

Figure 1. The Ocelet modeling and simulation framework *(from Degenne et al. 2009)*.

adaptable applications compared with the classical development methods. The implementation is based on service exchange and is adapted to the design of applications. It provides the ability to generate non-functional code. The generator handles the generation of the application, providing the glue to enable it to work on a specific platform, according to the context of use. If the platform or the underlying technology evolves, only the generator would be updated, it is not necessary to update the business logic that represents the domain specific expertise. The design of this model is based on: i) the component description, named CDML and ii) the deployment description, named World.

4.2.1 The component description (CDML)

We have defined an abstract Component Description Meta Language i.e. independent from any component technology:

- To extend the classical method-call. In this way, the runtime environment can be taken into account without any modification to the business logic.
- To extend the notion of interface. The provided and required services can be described and discovered, and the interface can dynamically be adapted.
- To add meta-information to a component. This is a generic approach to record information dealing with several concerns such as deployment management, component behavior or other SOC aspects.

When these mechanisms are included, it is possible to build applications with only the required components and to simplify the interconnections with external tools. From a component model instance, a generator can automatically produce the non-functional code, that is to say the container that hides all the communication and interconnection mechanisms (like the transformation of a method call by a sending message), the management of a queue of received messages, and the broadcasting of a message toward the connected components. These mechanisms are totally transparent for the designer of an application. Additionally, it is easy to adapt the architecture in order to introduce a new communication protocol.

4.2.2 The deployment description (World)

The deployment description file is used to describe the initial state of a simulation. It contains a description of the components and connections that have to be created by the Component Manager before starting the application. Of course after that, some components can ask to be connected with each other dynamically as explained in the next subsection. An instance is identified by the couple (name of the component, name of the instance). For example in Figure 2, the instance (cmp1, cmp1-1) corresponds to an instance of component cmp1.

```
<world>
 <connectTo id_src="ComponentsManager"
            type_dest="cmp1" id_dest="cmp1-1" />
 <connectTo type_src="cmp1" id_src="cmp1-1"
            type_dest="cmp2" id_dest="cmp2-1" />
 <connectTo type_src="cmp1" id_src="cmp1-1"
            type_dest="cmp2" id_dest="cmp2-2" />
</world>
```

Figure 2. An example of the deployment description.

4.2.3 The Components Manager (CM)

The Components Manager loads packages of components and creates the instances according to the deployment description file and eventually to the interactive request of users (via the graphical user interface). It stores all the loaded packages and the created instances. In particular, The Components Manager offers the service *connectTo* to connect two components. This service also allows the creation of the components if they do not yet exist. To establish connections, the manager uses the descriptions of components based on their names, output connectors (vs. input) are connected with the input connectors (vs. output) of other component. When connected, the two component instances interact with each other directly without going through the manager. Connection management, which includes creation or destruction of connection, occurs when the Components Manager receives notifications announcing changes in the component registry. These mechanisms allow an application to be made of interconnected component instances that can adapt dynamically with respect to availability. In fact, the Components Manager monitors the execution context and acts on the component by managing its connection policy. In distributed mode, to know

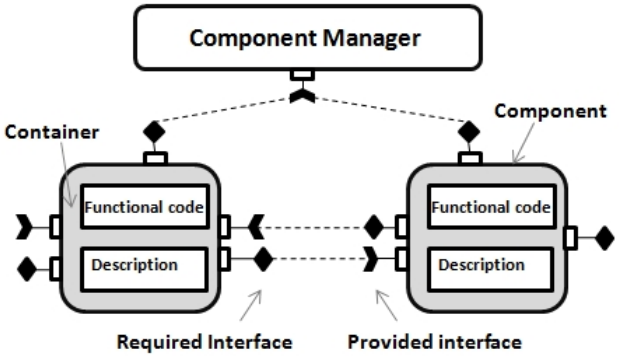

Figure 3. Connection between instances of components.

whether an instance is already created, the CM should not be limited to local search. If the instance does not exist locally then the CM should also extend the search to all connected CMs. For a better modularity and information management, the CM delegates the management of components and instances tables to components modules for each communication mode (local or distributed). The CM has a policy to choose the component module that will make the effective connection. For example, a policy will favor local connections over distributed connections. Moreover, the CM structure allows to instantiate different policies by using the *Command* design pattern [21]. The request to connect components modules is done in two steps. In the first step, the CM interrogates all active components modules on the presence or not of the instance of the destination. Each component module responds asynchronously to the CM. When the CM is in possession of all responses (even negative) then in the second step, it selects according to its policy the component module that handles the effective connection. If in the first step, there is no positive response, the connection request is put on hold until the CM receives a notification, such as a component has been started or discovered.

4.3 Implementation

This approach has been fully integrated into the Eclipse environment [22] and implemented on top of OSGi [23]. Eclipse is built around a very small extensible runtime core and its functionality, (including compilers, workbench, and support tools) consists of plug-ins that can be managed separately. It allowed us to integrate the Ocelet editor and generator plug-ins efficiently. The OSGi service platform provides a computing environment for applications, called bundles, to dynamically deploy services in a centralized environment. It is also a small layer that allows multiple components to efficiently cooperate in a single Java Virtual Machine (JVM) by managing aspects of local service deployment, but leaves service dependency management as a task for component developers.

At the start of execution, the OSGi platform is launched and the Components Manager (bundle) is started by default. In this context, two OSGi services are used and published. The first one, called ContainerService, allows publishing the CDML when a component (bundle) is started. The CM then adds that started component to its table of available components. The second one, called, ContainerProxy, allows publishing the component instance when it is created. The CM then adds that new instance to its table of created instances. The CM can then manage the execution as described in section 4.2.3 in a shielded environment and the components can reuse and cooperate, unlike other classic Java application environments. Moreover, installing a new bundle, registering a new service, or updating an existing component does not need a restart of the JVM. The concerned components are notified of the new state and adapt as a consequence.

This framework constitutes an extension of the SmartTools software factory [18, 19].

4.4 Advantages of the proposed Environment

This execution environment provides the possibility of separating the functional and non-functional aspects (parallelism, communication protocol, sending and receiving messages ...) in a simple way. In fact, the modeler of the application does not need to know how the non-functional code is implemented and it can be written directly with the Ocelet syntax.

Communication between different components is based on an exchange of messages completely transparent to

```
relation Predation(Predator, Prey) {
    requires property number Predator.nbrPopulation;
    requires property number Prey.nbrPopulation;
    requires service Predator.updatePopulation(number);
    requires service Prey.updatePopulation(number);

    service updatePredator(){
        Predator.updatePopulation(delta * Predator.nbrPopulation *
            Prey.nbrPopulation * dt);
    }

    service updatePrey(){
        Prey.updatePopulation(-( beta * Predator.nbrPopulation *
            Prey.nbrPopulation * dt));
    }
}
```

Figure 4. Predation relation written in Ocelet.

the modeler. This communication can be synchronous or asynchronous according to his need. As the execution environment associated with each component is a light process (Thread), an implicit parallel execution is possible.

Encapsulation of Ocelet elements in components allows the reuse of these elements in other models, as the interaction of an element with other Ocelet elements depends only of the services it requires and provides.

5 Application

This section presents an execution scenario illustrating some requirements of landscape modeling. The well-known prey-predator model introduced by Lotka (1925) and Volterra (1926) [24, 25, 26] presented highlights the needs in terms of dynamicity and service interaction. The model is based on a system of non linear differential equations frequently used to describe the dynamics of ecological systems in which two species interact and evolve during time, one a predator and one its prey:

$$\begin{cases} \frac{dx}{dt} = x.(\alpha - \beta y) \\ \frac{dy}{dt} = y.(-\gamma + \delta x) \end{cases} \quad (1)$$

where
α is an expression of the birth rate in the prey population
β is the death rate of prey due to predation
γ represents the natural death rate in the population of predators
δ is the rate of predator population growth per prey consumed
Using Ocelet, two entities (Rabbits for preys and Foxes for predators) and one relation (the Predation relation) are defined; the time flow of the system is also described in a scenario (the Evolve scenario). Ocelet is designed to promote separation of concerns and in the present case the system of equations is split into the following parts:

- The birth rate of prey is calculated by the Rabbits entity through a birth service.
- The natural death of predator is calculated by the Foxes entity through a natural_death service.
- The death rate of prey due to predation and the growth of predator population due to predation have a meaning only if preys and predators meet in a model. They are hence calculated in the Predation relation by two respective services, updatePrey and updatePredator.

The relation provides a connection mechanism relying on a specific component called relation. When two entities are connected by the predation relation, the corresponding relation component acts as an interposition object by providing the updatePrey and updatePredator services. This relation component allows to enrich the connected components without requiring changes in them. The relations therefore offer better decoupling between the business code (inside components) and the connection code (inside relations). It is important to note that the separation between business and connection codes allows to reuse already developed relations, entities, scenarios and datafacers to build new models. Figure 4 shows the Ocelet code of the predation relation that models the death rate of prey due to predation $(-x.\beta.y)$ and the growth of predator population due to predation $(y.\delta.x)$ expressions.

This model is implemented above our service-oriented component framework. In fact, for each Ocelet concept entity (Rabbits, Foxes), relation (Predation) and scenario (Evolve) that the modeler uses to specify the business logic (using an Eclipse plug-in editor developed for this need), Java class files implementing the business code and CDML files describing the service (provided and required) are generated as depicted in Figure 1. A World file describing the initial state of the application is also

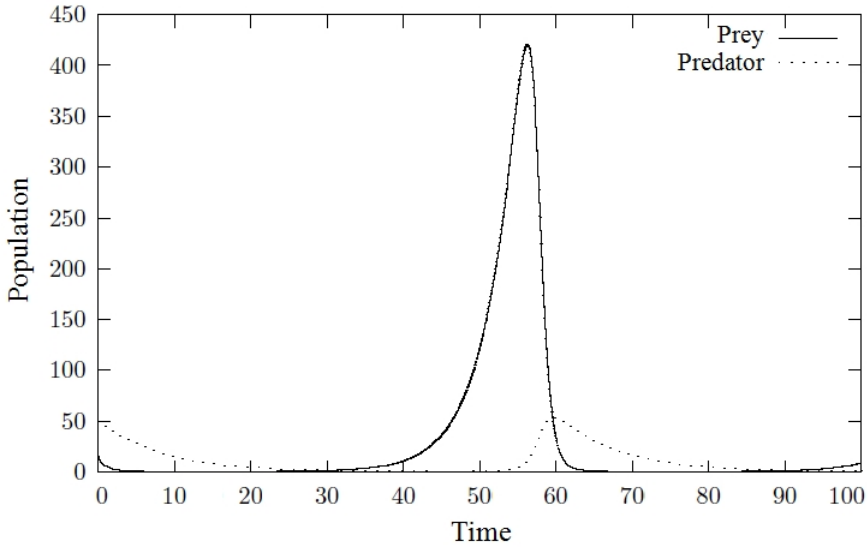

Figure 5. A simulation of the Lotka-Volterra model developed with Ocelet

generated. The component generator will then create a container for every entity, relation and scenario. Each container encapsulates the business code and the service description. Thus, we get components ready to be used or archived in the Java ARchive (JAR) files. The World file can then be used by the Components Manager to load the packages of components, create the instances, and wait a signal from the graphical user interface to start the simulation. The interaction between predators and preys in the Lotka-Volterra model is therefore transformed into a dynamic service interaction between components in a manner completely transparent to the modeler. The results of this interaction are shown in Figure 5 ($\alpha = 0.1; \beta = 0.01; \gamma = 0.05; \delta = 0.001$).

Although this illustrative example may appear simple, the principal aim is to demonstrate how to introduce simplification when building an application that supports dynamism, in landscape processes modeling. Hence modelers can benefit from these mechanisms to create more complex applications.

6 Conclusion

Applications for landscape modeling and simulation are often difficult to build. In fact, developers must at the same time handle the domain constraints and the non-functional requirements in addition to their business logic. In this paper, we present our approach to meet those needs relying on service-oriented computing. Traditional component approaches simplified software system development by allowing developers to create software bricks without taking into account the functional aspect and are generaly limited to the application architecture. Here, our approach uses components as well as service-oriented concepts throughout the development process (from design to execution). This requirement was specifically imposed in our case by the domain context of landscape modeling.

This paper focuses more on the needs in terms of execution than on the concepts of landscape modeling introduced in Ocelet [1]. The originality of the execution service-oriented component framework is to provide to the modeler an environment that supports mechanisms for dynamic extension and a distributed execution.

Finally, this environment is extensible: the Component Generator may be enriched with new features to follow the evolution of Ocelet. More precisely, we are currently investigating a way to enhance the Component Description Meta Language with a semantic description of a components functionality which expresses behavior. This description should ensure a correct composition of Ocelet entities and identify an entity with the appropriate semantics for the purpose of cohabitation of several Landscape-context applications.

7 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).

8 References

[1] P. Degenne, D. Lo Seen, D. Parigot, R. Forax, A. Tran, A. Ait Lahcen, O. Cure, and R. Jeansoulin,

Design of a domain specific language for modelling processes in landscapes, *Ecological Modelling*, In press, 2009.

[2] C. Szyperski, *Component Software : Beyond Object-Oriented Programming* (New York: ACM Press and Addison-Wesley, 1998).

[3] M.P. Papazoglou and W. Heuvel, Service oriented computing, *Communications of the ACM*, 46(10), 2003, 25-28.

[4] M. N Huhns and M. P Singh, Service oriented computing : Key concepts and principles, *IEEE Internet Computing*, 9, 2005, 75-81.

[5] P.A. Burrough and R.A. Mcdonnell, *Principles of Geographical Information Systems* (Oxford University Press, USA, 1998).

[6] A. Borshchev and A. Filippov, From system dynamics and discrete event to practical agent based modeling: reasons, techniques, tools, *In Proceedings of the 22nd International Conference of the System Dynamics Society*, Oxford, England, 2004.

[7] F. Bousquet and C. Le Page, Multi-agent simulations and ecosystem management: a review, *Ecological Modelling*, 176(3-4), 2004, 313-332.

[8] C. Ratze, F. Gillet, J.P. Muller, and K. Stoffel, Simulation modelling of ecological hierarchies in constructive dynamical systems, *Ecological Complexity*, 4, 2007, 13-25.

[9] G. Langran, *Time in Geographic Information Systems* (London: Taylor and Francis, 1992).

[10] D.J. Peuquet, Its about time: A conceptual framework for the representation of temporal dynamics in geographic information systems, *Annals of the Association of American Geographers*, 84(3), 1994, 441- 461.

[11] M. Yuan, Use of a three-domain repesentation to enhance gis support for complex spatiotemporal queries, *Transactions in GIS*, 3, 1999, 137-159.

[12] C. Parent, S. Spaccapietra, and E. Zimanyi, Conceptual modelling for traditional and spatio-temporal applications: The MADS approach, (Springer-Verlag Berlin Heidelberg, 2006).

[13] M.F. Goodchild, Geographical data modeling, *Comput. Geosci.*, 18(4), 1992, 401-408.

[14] D.J. Peuquet, Making space for time: Issues in space-time data representation, *In DEXA 99: Proceedings of the 10th International Workshop on Database and Expert Systems Applications*, Washington, DC, USA, 1999, 404.

[15] M. Mernik, J. Heering, and A. M Sloane, When and how to develop domain-specific languages, *ACM Computing Surveys*, 37(4), 2005, 316-344.

[16] C. Courbis, P. Degenne, A. Fau, and D. Parigot, Un modele abstrait de composants adaptables, *revue TSI, Composants et adaptabilite*, 23(2), 2004, 231-252.

[17] P.K McKinley, S.S. Masoud, E. P Kasten, and B.H.C. Cheng, Composing adaptive software, *IEEE Computer*, 37(7), 2004, 56-64.

[18] D. Parigot, Towards domain-driven development: the smarttools software factory, *In OOPSLA 04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2004, 37-38.

[19] D. Parigot, Smarttools software factory, *EclispeCon 08*, 2008.

[20] J. Liu, J. He and Z. Liu, A strategy for service realization in service-oriented design, *Science in China Series F: Information Sciences*, 49(6), 2006, 864-884.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of reusable object-oriented software* (Addison-Wesley Publishing, 1995).

[22] The Eclipse Foundation, *Eclipse Platform Technical Overview*, 2003.

[23] OSGi Alliance, *OSGi Service Platform Core Specification, release 4 edition*, 2005.

[24] V. Volterra,Fluctuations in the abundance of a species considered mathematically, *Nature*, 118, 1926, 558-560.

[25] A.J. Lotka, *Elements of Physical Biology* (Baltimore, MD: Williams and Wilkins, 1926).

[26] J.D. Murray, *Mathematical Biology: I: An introduction* (New York: Springer, 2002).