Jul 1st, 12:00 AM

# A Design for Framework-Independent Model Components of Biophysical Systems

Marcello Donatelli

Andrea-Emilio Rizzoli

*iEMSs 2008: International Congress on Environmental Modelling and Software*
*Integrating Sciences and Information Technology for Environmental Assessment and Decision Making*
*4th Biennial Meeting of iEMSs, http://www.iemss.org/iemss2008/index.php?n=Main.Proceedings*
*M. Sànchez-Marrè, J. Béjar, J. Comas, A. Rizzoli and G. Guariso (Eds.)*
*International Environmental Modelling and Software Society (iEMSs), 2008*

# A Design for Framework-Independent Model Components of Biophysical Systems

**M. Donatelli[a,**,†], A.E. Rizzoli[b]**

[a] *Agriculture Research Council, 40128 Bologna, Italy;*
[**] *currently seconded to European Commission Joint Research Centre, Institute for the Protection and Security of the Citizen, Agriculture Unit, Agri4cast Action, Ispra, Italy*
[b] *Dalle Molle Institute for Artificial Intelligence, Manno-Lugano, Switzerland;*
[†] *Corresponding author, marcello.donatelli@jrc.it*

**Abstract:** Most efforts in the design of software frameworks for biophysical systems simulation have focused on the compromise between domain specificity and use flexibility. Models in such frameworks fall in two main categories: either framework-specific, or "legacy" code. In the former case, models implemented as software components can take full advantage of the framework services, but they depend on the framework. In the latter case, components are seen as discrete units of software, in general of coarse granularity in modelling terms, and the dependency on the framework is minimal, but the potential for composition and reuse is limited. Thus, modellers who want to use a modelling framework are faced with two choices: if the framework is extensible, implement a framework specific component (i.e. not reusable outside the specific framework); else, the alternative is to provide a component as a black-box, taking little or no advantage of the framework itself. We argue that component design choices, rather than being driven by the specific framework architecture, should rather promote re-usability by including design traits that represent a compromise between generality and specificity in order to maximize the adaptability of components. This paper present: 1) a software design of non-framework specific components, and 2) real-world applications of the design presented.

***Keywords:*** Modelling; Component-oriented programming; Software components.

## 1. INTRODUCTION

In the past decade there has been an increasing demand for modularity and replaceability in biophysical models (e.g. [Jones et al., 2001]; [David et al., 2002]; [Donatelli et al., 2003, 2004, 2006]), both aimed at improving the efficiency of use of resources and at fostering a higher quality of modelling units. Rather than having generalist modellers working on all details of complex integrated models, it is more efficient and effective to assemble sub-models developed by specialists in the specific sectors.

The modular approach developed in the software industry is based on the concept of encapsulating the solution of a modelling problem in a discrete, replaceable, and interchangeable software unit. Such discrete units are called components. A software component can be defined as "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject by composition by third parties" [Szypersky et al., 2002].

Component-oriented designs actually represent a natural choice for building scalable, robust, large-scale applications, and to maximize the ease of maintenance in a variety of domains, including agro-ecological modelling [Argent, 2004]. This concept has been applied to biophysical simulation and has led to the development of several modelling frameworks (e.g. Simile, MODCOM, IMA, TIME, OpenMI, SME, OMS, as listed in Argent and Rizzoli [2004], and in Rizzoli, Leavesley et al. [in press]), which allow the use

of components by linking them either directly, or through a simulation engine, if they expose their interface requesting a numerical integration service. However, targeting model component design to match a specific interface requested by a modelling framework decreases its re-usability. This possibly explains why modelling frameworks, although a great advance with respect to traditional model code development, are rarely adopted by groups other than the ones developing those [Rizzoli et al., in press].

A possible way to overcome the problem of scarce reusability of components is to adopt a component design that targets the intrinsic re-usability and interchangeability of model components (e.g. Donatelli et al., [2006a, b]). Such components can be used in a specific modelling framework via an application of the design pattern Adapter [Gamma et al., 1994]. Such classes act as bridges between the framework and the component interface.

The communication between a component and its client (a framework as above), and more in general the ease of re-use, can be enhanced by addressing the following requirements:

1. the component must target the solution of a sufficiently widespread modelling problem;
2. the published interface of the component must be well documented and it must be consistent;
3. the configuration of the component should not require excessive pre-existing knowledge and help should be provided in the definition of the model parameters;
4. the model implemented in the component should be extensible by third parties,
5. the dependencies on other components should be limited and explicit;
6. the behaviour of the component should be robust, and degrade gracefully, raising appropriate exceptions;
7. the component behaviour should be traceable and such a trace should be scalable (browseable at different debug levels);
8. the component software implementation should be made using widely accepted and used technologies.

In the following we present the choices we made in the design of some components developed by an informal network of biophysical modellers.

## 2. THE DESIGN

The design of a software component implementing a biophysical model requires the ability to identify the right trade-offs in order to deliver a software product that complies with the requirements, while respecting given performance constraints. In particular, we identified a number of design choices, which helped us in the realisation of the software components. We can classify the choices in three broad categories: structure, interaction, and quality. Choices regarding the model granularity and the model interface pertain to the design of the component structure. Choices on the component extensibility and the component dependency are related to the design of the component interaction with a modelling framework. Choices on the component reliability, on the tracing of its execution and even on the technology for its implementation are strictly connected to the measure of the component quality.

### 2.1. Design of component structure

#### 2.1.1 Model granularity

An essential part of modelling is choosing the model resolution power, that is, the model *granularity*. As a simple example, think of a population dynamics model, where the fractional growth rate is a parameter. This model can be refined, letting the fractional growth rate become a function of the population density. The fractional growth rate is therefore described by two models with different granularity: a constant parameter, or a limiting function. A similar situation holds for the alternative formulation of the limiting functions, which can be exchanged to originate different population growth curves.

"Fine grained" components are more likely to be reused for specific computations, in the context of larger modelling problems. The formulations of the limiting functions can be exchanged among different models, even in different contexts, from the growth of population of bacteria, to the growth of plant biomasses. In this sense, fine grained components match the requirement of providing a solution to a sufficiently widespread modelling problem. Yet, the complexity of a fine grained model can rapidly grow and become unwieldy, requiring a lot of domain specific knowledge to make an actual reuse of the basic components. On the other hand "coarse" modelling solutions have the advantage of hiding the complexity of several modelling approaches by providing a pre-made composition of those, yet their reuse in larger systems limits the flexibility of the overall model; in fact a large parte of it, represented by the "coarse" component, cannot be modified, with problems in terms of maintenance and further development.

Design patterns provide us with a compromise: using an implementation of the Façade pattern we can still adopt a "coarse" modelling solution, hiding the complexity of each model solution while preserving a modular structure based on simple model units. Hence, simple model units can either be used in isolation or they can be composed to develop other modelling units. Examples are the CLIMA components [Donatelli et al. 2005] which implement fine-grained models that generate synthetic weather variables. The component architecture adopts the Strategy design pattern in order to allow for the plugging-in of alternative model formulations to generate the model output, since various models can be used for the same purpose. Each component exposes an interface which must be implemented by each model unit, both internal and from extensions (see 2.1.2) of the component (see Figure 1). The choice of inheriting behaviour via an interface, rather than inheriting implementation from a base class, maximizes flexibility and still allows for inheritance from a class in platforms like Java or .NET which do not allow for multiple inheritance. The CLIMA components referenced above match such requirements.

### 2.1.2 Model interface

A component implements one or more modelling solutions for a specific domain. A software component can be seen as a block box that processes inputs and returns outputs. Also a dynamic model can be seen as identified by its state transition function and its output transformation function [Zeigler, 1995]. It is therefore straightforward to package the model in a software component exposing the model inputs and outputs as interface. The use of software components is equally valid for static and dynamic components, in the latter case interface variables are often declared as types with names such as States, Rates, Auxiliary, Exogenous etc.). It is also evident that modelling choices define an abstraction of the domain being modelled by selecting which inputs and outputs to consider and how to describe and detail them. Del Furia et al., [1995] proposed to use object-oriented data structures called *domain classes* to describe and implement such data. Each attribute of a domain class classes has, in the design presented, beside its value, a set of attributes such as minimum, maximum, and default value; units; description.

The value of domain classes goes beyond their meaning as software implementation items, in fact they describe the domain of interest providing information about each variable used in the interfaces, making the interfaces semantically explicit. (Athanasiadis et al. 2008) therefore extended the concept of a domain class including the possibility to refer to a publicly available ontology via the attribute URL. There is a direct mapping between domain classes and ontologies, since the domain class code can be automatically generated from the ontology. The clear advantage is that the ontology is language and platform (and framework) independent and it is therefore possible to provide a description of the domain class, which is totally abstracted of the specific technological solution which is adopted. A simple application to generate the code of domain classes in .NET is currently available [DCC, 2006].

An interesting consequence is that if domain classes and interfaces are implemented in a separate unit from models, the model software unit can be replaced without affecting the client using the Domain classes which are part of the signature of the interface methods (the interface that all strategies must implement). An example of such interfaces, for static component, is shown in Figure 1. A component implementing domain classes and

interfaces and another implementing models are a *unit of reuse*; the model component alone can be defined as a *unit of interchangeability* (see Figure 2).

The interface to access models is used both for simple models ("simple strategies") and for composite models ("composite strategies"), which make use of other simple models by implementing an association to simple strategies. This is an implementation of the Composite design pattern (Bishop, 2008). At the same time, composite strategies hide the complexity of the system, providing a single point to access articulated modelling approaches, hence making an implementation of the Façade pattern, as previously mentioned. Also, the structural implementation of the pattern Composite allows an easy implementation of the behavioural pattern Strategy. In fact, context strategies can be built encapsulating the logic to select alternate models (strategies) according to various criteria, e.g. input data availability.

```
/// <summary>
    /// Interface that all ClimIndices strategies must implement
    /// </summary>
    public interface IClimIndicesStrategy : IStrategy
    {
        /// <summary>
        /// Makes estimate
        /// </summary>
        /// <param name="d">Data-type for daily and monthly weather data,
        /// and site data</param>
        /// <param name="u">Univariate statistics</param>
        /// <param name="dc">Clim Indices</param>
        void Estimate(DataWeather d, UnivariateDataWeather u,
                                                    DataClimIndices dc);

        /// <summary>
        /// Tests pre-conditions
        /// </summary>
        /// <param name="d">Data-type for daily and monthly weather data,
        ///  and site data</param>
        /// <param name="u">Univariate statistics</param>
        /// <param name="callID">Client ID for specific call</param>
        /// <returns>Result of pre-conditions test</returns>
        string TestPreConditions(DataWeather d, UnivariateDataWeather u,
                                                        string callID);

        /// <summary>
        /// Test post-conditions
        /// </summary>
        /// <param name="dc">Clim Indices</param>
        /// <param name="callID">Client ID for specific call</param>
        /// <returns>Result of post-conditions test</returns>
        string TestPostConditions(DataClimIndices dc, string callID);

        /// <summary>
        /// Resets output to NaN / smallest integer possible
        /// </summary>
        /// <param name="dc">Clim Indices</param>
        void ResetOutputs(DataClimIndices dc);

        /// <summary>
        /// Set parameters default
        /// </summary>
        void SetParametersDefault();
    }
```

**Figure 1. The interface of a strategy in a software component that computes weather indices.**

An example of an online ontology browser for the domain classes and components implementing the design described and referenced in this paper, is available at: http://www.apesimulator.it/ontologybrowser.aspx. The software components can also be inspected using a Windows application [MCE, 2006].

We have thus targeted the requirement that the interface of the component must be well documented and it must be consistent. Moreover, thanks to the ontology, the configuration of the component does not require excessive pre-existing knowledge, and help is provided in the definition of the parameters, since they can have default, minimum and maximum values.

## 2.2 Design of the component interaction

### 2.2.1. Component API and component extensibility

A software component exposes an application programming interface (API) that allows the use and integration of the component in a software architecture. The API of the component must implement a pattern like the Create-Set-Call [Cwalina and Abrams, 2006]: objects are created via a default constructor with no parameters, some attributes are set, and finally the model is called.

A component implementing a dynamic model must be iteratively called to compute the updated values of the state and output variables, as specified by the state transition and the output transformation equations. Our design choice was to provide a unique method to call all models. The signature of the method can be like the one that follows:

```
Update(DomainClass d, IStrategy s);
```

where `d` is an input-output object (a domain class) and `s` is a strategy, that is, a particular modelling choice to be used in the state transition or output transformation. Being `DomainClass` and `IStrategy` public, and being inheritance from `DomainClass` allowed, the component method can be extended both from the domain class and the strategy viewpoint. In fact, if an extension of the domain class is needed, the new domain class will inherit from `DomainClass`, whereas a new strategy will implement `IStrategy`. This allows using in clients the same API, usable for both the original models and extended models. This design choice answers the requirement of extensibility of the component. Implementation tests were made both in Java and in C# comparing this solution to direct calls to algorithms, resulting in a negligible difference in performance.

Finally, components should be stateless, to simplify their use in different systems. Sample clients, inclusive of code which show how to use and extend the component, must be made available.

#### 2.2.2 Component dependencies

While dependencies should be kept at a minimum, we found necessary and particularly useful to introduce a dependency to another component, named Preconditions, available both in C# and Java (http://www.apesimulator.it/help/utilities/preconditions) which provides the essential services required by this software architecture:

- it contains the base interfaces for models (`IStrategy`) and domain classes (`IDomainClass`);
- it provides a type (`VarInfo`) used to describe the attributes of each variable (name, min and max values, default value, etc.);
- It is used to guarantee the quality of the implemented solution via the design-by-contract approach.

Given that the instances of the VarInfo type contain information on the variables, the Precondition components uses this information to run a suite of built-in tests. More tests can also be defined using the built-in ones. This component outputs tests results to screen, to a default TXT file, to an XML file, and to a listener to be defined by the client using the component; other output drivers can be developed implementing the interface `ITestsOutput` made available by the component. Other dependencies to specific libraries (e.g. for numerical calculus) can be included, but no dependency to any specific frameworks is implemented. The UML diagram of Fig.1 shows the main components used according to the design presented. The design choices we made in the realization of the Precondition component implement the requirement for limited and explicit dependencies.

### 2.3. Design of component quality

#### 2.3.1 Components reliability

The robustness of a software component is greatly enhanced if the implementation follows the design-by-contract approach [Meyer 1992], which requires that a clear contract between “client” and “server” is established. The server is the software component, the client is an application (or another component) making use of the component resources, and the contract is the respect of the pre- and post-conditions that must hold after the invocation of one of the component’s services.

Each component, implementing a model, must therefore come with its "contract", that is the conditions that identify its domain of applicability, and the limits to the use of its results. This allows developing a more specific suite of unit tests (the documentation of the components must contain at least some of the unit tests performed during implementation). All of this contributes to the transparency of the modelling solution.

Test of pre- and post-conditions is implemented by overloading the component API (client driven choice); however, if an unhandled exception occurs, the test of pre-conditions is run and an informative message describes the error and model and component source of the exception, allowing for continuing execution of the client according to a user choice. It is evident that this design choice satisfies the requirement for components to be robust, degrade gracefully, and raise appropriate exceptions.

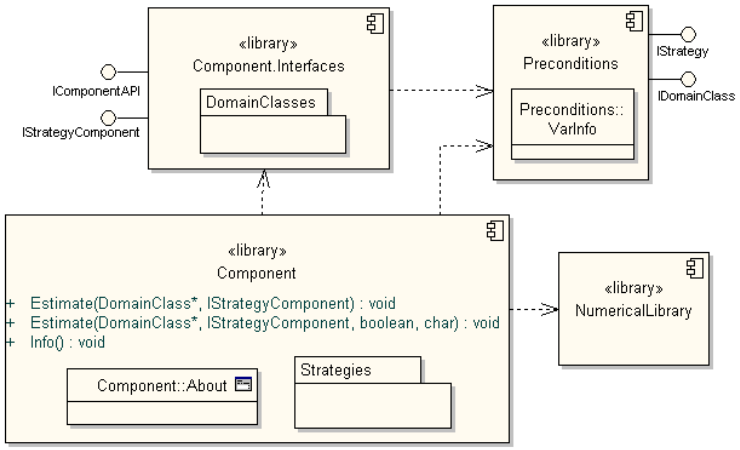


**Figure 2. Example of a component diagram for the proposed architecture**.

2.3.2 Tracing

Being able to closely inspect the behaviour of a component is a powerful tool to ascertain its quality. Traceability is therefore a major quality requirement. The traceability of component behaviour is implemented in the version of the components implemented in C# and running on the NET platform using the `System.Diagnostic.TraceSource` class, in one implementation that allows setting the listeners by the client. Various levels of tracing (critical, error, warning, information, and verbose) can be hence pooled in one or more listeners with all traces from other components and from the client. In Java components this is obtained using a logger. This satisfies the requirement for traceability of the component execution.

2.3.3 Technology

Sometimes, once a solid design has been cast, technology is seen as of secondary importance. Unfortunately this is not the case, and technology and its evolution often drives the design process in a tightly connected feedback loop

We used the Microsoft .NET 2.0 framework to implement our components. However, the object model of .NET allows easy migration to the Sun Java platform. Such migration has been actually made for some of the components referenced. Although the same design can be implemented under different platforms, the advantages of a memory managed environment combined to the object model of .NET or Java makes these platforms a first choice. Specific routines can still be written using "unsafe" C++ (as opposed to "managed" C++ in .NET) to optimize performance, but this should be seen as the exception not to give up all the advantages of a managed memory in complex systems using components with different origin.

## 3. PROOFS OF CONCEPT

Components implementing the solutions above have been made available for public use and other are being developed. The design has been tested on static [Acutis et al., 2008];

[Carlini et al., 2006]; [Confalonieri et al., 2008]; [Donatelli et al., 2006a and b] and dynamic biophysical models [Acutis et al., 2007]; [Trevisan et al., 2007], on agro-management models [Donatelli et al., 2007] and on statistical indices [Bellocchi et al., 2008]. Use of components has been done on applications (desktop and web, including web services) and via frameworks such as TIME (not published) and Modcom [APES, 2007]. Components can be used directly in applications as shown in the sample projects provided in the software development kits, of via adapters in modelling frameworks, as shown in the component diagram of Fig. 3.

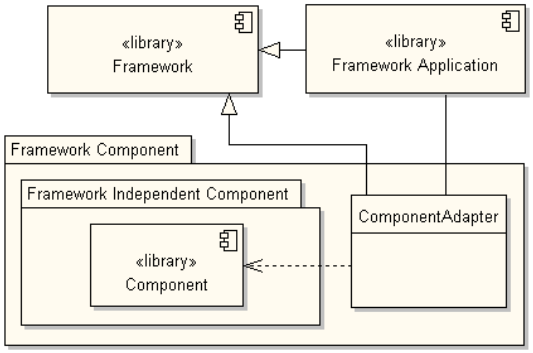


**Fig. 3 Using a framework independent component in a model framework**

## 4. CONCLUSIONS

Shifting the focus from modelling frameworks to a component design that follows the component oriented design results in a greater chance for model re-use. Framework independency stimulates model developers who do not feel constrained by a dependency to groups which develop specific frameworks; at the same time components can be easily used in several frameworks via simple wrappers. Design choices related to the modularity of model implementation and the provision of an explicit ontology for interfaces increases the transparency of the model construct and allows sharing knowledge in quantitative and usable terms. Several functionalities can be easily used to various extents by different clients. The design presented allows for extensibility by third parties which can then build on the domain description and models made available.

## ACKNOWLEDGEMENTS

This publication has been funded by the SEAMLESS integrated project, EU 6th Framework Programme for Research, Technological Development and Demonstration, Priority 1.1.6.3. Global Change and Ecosystems (European Commission, DG Research, contract no. 010036-2). We are in debt to Riccardo Wenger for the implementation of the CLIMA and the Preconditions components in Java. We also gratefully acknowledge Rob Knapen for his work in testing the performance of options for components interfaces in the Java platform.

## REFERENCES

Acutis, M., P. Trevisiol, A. Gentile, D. Ditto, and L. Bechini, In M. Donatelli, J. Hatfield, and A. Rizzoli, editors, *Farming Systems Design 07, Methodologies for Integrated Analysis of Farm Production Systems*, volume 2, page 201. ESA, 2007.

Acutis, M., M. Donatelli and G. Lanza Filippi, PTF: an extensible component for sharing and using knowledge on pedo-transfer functions. *These Proceedings*.

Athanasiadis, I. N., A. E. Rizzoli, M. Donatelli, and L. Carlini, Enriching software model interfaces using ontology-based tools iEMSs Congress, Vermont, July 2006 (in press on Int. J. of Adv. Systemic Studies)

APES, Agricultural Production and Externalities Simulator, http://www.apesimulator.org; http://www.apesimulator.org/help/apes/ help on-line.

Argent, R.M., An overview of model integration for environmental applications – components, frameworks and semantics. *Environmental Modelling and Software* 19(3), 219-234, 2004.

Argent, R.M. and A.E. Rizzoli, Development of multi-framework model components. In: Pahl-Wostl C., Schmidt S., Rizzoli A.E., Jakeman A.J. (Eds.), *Trans. of the 2nd biennial meeting of the International Environmental Modelling and Software Society*, Osnabrück, Germany, vol. 1, 365-370, 2004.

Bellocchi, G., M. Donatelli, M. Acutis, and E. Habyarimana,. A Component and Software Application for Model Output Evaluation. *These proceedings*.

Bishop, J., 2008. C# 3.0 Design Patterns, O'Reilly, 314 pp.

Carlini, L., G. Bellocchi, and M. Donatelli, Rain, a software component to generate synthetic precipitation data. *Agronomy Journal*, 98, 1312-1317, 2006.

Confalonieri R., G. Bellocchi, and M. Donatelli, A software component to compute agro-meteorological indicators. Submitted to: *Environmental Modelling and Software*, 2008.

Cwalina, K., and B. Abrams, Aggregate components. In: *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley, Courier in Westford, Massachusetts, USA. 235-271, 2006

David, O., S.L. Markstrom, K.W. Rojas, L.R. Ahuja and W. Schneider. The object modelling system. In: Ahuja L.R., Ma L., Howell T.A., (Eds.), *Agricultural system models in field research and technology transfer*. Lewis Publishers, Boca Raton, FL, USA, p. 317-344, 2002.

Del Furia, L., A. Rizzoli and R. Arditi, Lakemaker: A general object-oriented software tool for modelling the eutrophication process in lakes, *Environmental Software*, 10(1), 43-64, 1995.

DCC, Domain Class Coder. http://www.sipeaa.it/ASP/ASP2/DCC_GuestBook.asp, installation; http://www.apesimulator.it/help/tools/dcc/ help on-line, 2006.

Donatelli, M., G. Bellocchi, and L.Carlini, A software component for estimating solar radiation. *Environmental Modelling and Software*, 21(3), 411-416. 2006a.

Donatelli M., G. Bellocchi, and L. Carlini, Sharing knowledge via software components: models on reference evapotranspiration. *European Journal of Agronomy* , 24(2), 186-192, 2006b.

Donatelli, M., J. Bolte, F. van Evert and W. Wang. Which software designs for evolution. In: van Ittersum M.K., Donatelli M. (Eds.), Modelling cropping systems: science, software and applications. European Journal of Agronomy 18, 193-195, 2003.

Donatelli, M., L. Carlini, G. Bellocchi, M. Colauzzi, CLIMA: a component based weather generator. In Zerger, A. and Argent, R.M. (eds) *MODSIM 2005 International Congress on Modelling and Simulation*. Modelling and Simulation Society of Australia and New Zealand, December 2005 ISBN: 0-9758400-2-9, 627-633, 2005

Donatelli M., E. Ceotto, and M. Acutis, 2007. A software component to simulate agro-management, In M. Donatelli, J. Hatfield, and A. Rizzoli, editors, *Farming Systems Design 07, Methodologies for Integrated Analysis of Farm Production Systems*, volume 2, page 217. ESA, 2007.

Jones, J.W., B.A. Keating, C.H. Porter. Approaches to modular model development. Agricultural Systems 70, 421-443, 2001.

MCE, Model Component Explorer. http://www.sipeaa.it/ASP/ASP2/MCE_GuestBook.asp installation; http://www.apesimulator.it/help/tools/mce/ help on-line, 2006.

Meyer, B., Applying "Design by Contract", *IEEE Computer*, 25(10), 40-51, 1992.

Rizzoli A.E, M. Donatelli, I. Athanasiadis, F. Villa, and D. Huber, . Semantic links in integrated modelling frameworks. *Mathematics and Computers in Simulation*, doi:10.1016/j.matcom.2008.01.017, in press.

Rizzoli A.E., Leavesley G. et al. Integrated modelling frameworks for environmental assessment and decision support. In Jakeman, A.J., Voinov A., Rizzoli A.E., Chen, S. (eds) *Environmental Modelling and Software*. Elsevier, in press.

Szypersky, C., D. Gruntz and S. Murer, *Component software - beyond object-oriented programming*. 2nd Ed. Addison-Wesley, London, United Kingdom, 2002.

Trevisan M., A. Sorce, M. Balderacchi, A. Di Guardo, 2007. A software component to simulate agro-chemicals fate. In M. Donatelli, J. Hatfield, and A. Rizzoli, editors, *Farming Systems Design 07, Methodologies for Integrated Analysis of Farm Production Systems*, volume 2, page 231. ESA, 2007

Van Evert F., A. Lamaker, 2007. The MODCOM framework for component-based simulation. In M. Donatelli, J. Hatfield, and A. Rizzoli, editors, *Farming Systems Design 07, Methodologies for Integrated Analysis of Farm Production Systems*, volume 2, page 223. ESA, 2007.

Zeigler, B.P., *Object-oriented simulation with hierarchical modular models*. Academic Press, 1990.