Semantic links in integrated modelling frameworks

[1]Rizzoli, A.E, [2]M. Donatelli, [1]I. Athanasiadis, [3]F. Villa, [4]R. Muetzelfeldt, and [5]D. Huber

[1]IDSIA-USI/SUPSI, [2]CRA, UVM, [4]Simulistics, [5]AntOptima, E-Mail: andrea@idsia.ch

Keywords: *Integrated modelling frameworks; Ontologies; Model linking; Model reuse.*

EXTENDED ABSTRACT

It is commonly accepted that modelling frameworks offer a powerful tool for modellers, researchers and decision makers, since they allow the management, re-use and integration of models from various disciplines and at different spatial and temporal scales.

However, the actual re-usability of models depends on a number of factors such as the accessibility of the source code, the compatibility of different binary platforms, and often it is left to the modellers' own discipline and responsibility to structure a complex model in such a way that it is decomposed in smaller 're-usable' sub-components. What reusable and interchangeable means is also somewhat vague; although several approaches to build modelling frameworks have been developed, little attention has been dedicated to the intrinsic re-usability of components.

In this paper we focus on how models can be linked together to build complex integrated models. We review and investigate the various approaches to model linking adopted by a number of Integrated Modelling Frameworks and we aim at describing the advantages and disadvantages of each approach.

We stress that even if a model component interface is clear and reusable in software terms, this is not a sufficient condition for reusing a component across different Integrated Modelling Frameworks. This remark reveals the need for adding rich semantics in model interfaces; we do such an attempt through the use of *domain classes* and *ontologies*.

A domain class can be considered as an abstract data structure for defining a set of a model variables and their attributes (Rizzoli *et al.* 1998). A model interface (in terms of inputs, outputs, states and parameters) can be defined using a domain class, providing some advantages: first of all, an instance of a domain class can be accessed at run-time to supply the model component with the appropriate data. Secondly, it annotates model variables with attributes that can be used for pre-post condition checks. Thirdly, it supports compliance with the requirement that asks for model components to be separated from their data structures. And, last but not least, it provides an easy way for linking model components at a higher level. This practice uses shared domain classes for interchanging data across models, taking full advantage of component-based software engineering primitives.

Then, we present an approach based on the formalisation of ontologies to describe models' interfaces and relationships. The use of ontologies is advantageous as it (a) supports the automatic generation of code templates for models and domain classes in different Integrated Modelling Frameworks, (b) it facilitates the application of a reasoner (inference engine) on the structured knowledge, which can detect abnormalities or conflicts in model interfaces, and (c) it supports model linking in a content-enriched way, which can be proven valuable for avoiding common problems related to poor semantics of model interfaces.

Finally, this paper presents a working example of an ontology formalisation developed for the Seamless project[1]. This ontology (called SeamAg) aims to formally describe biophysical models related to agronomic and environmental domain to be developed by a large community of modellers within the Seamless project. Modellers' knowledge, related to model subsystems, variables and interfaces, is kept separated from the actual implementation. The use of the SeamAg ontology for storing model interfaces supports the independence of software design choices from modelling knowledge, which be easily reused, integrated in different environments, or shared with third parties. The potentials of extending the presented ontology-driven approach is discussed not only for model linking, but also in the context of building model component workflows using web services.

[1] http://www.seamless-ip.org

1. INTRODUCTION

Since System Theory introduced the concept of modular and hierarchical decomposition of models (Padulo and Arbib, 1974), researchers were quick in porting this concept into the implementations of their models, which were mostly done in FORTRAN. Subroutines were the logical counterpart to submodels, and function parameters were used to represent model inputs and outputs in the source code implementations. The use of global variables for passing values between submodels was still very common, but this was (and in some cases still is) a very bad programming habit, which has been spotted quite early by Parnas (1972): good modular programs must have subroutines which display a strong cohesion (lots of internal references to variables in the local scope), but that are loosely connected (very few data exchanges among subroutines, well defined by the subroutine signatures, i.e. their parameters).

Procedural programming has been used to write good implementations of mathematical models. This programming paradigm was well suited to representing modelling problems, where the decomposition of a system in simpler functions comes natural. Yet, the software designers were missing more powerful programming concepts, which could better support the representation of data, and not only their flow in the program.

The advent of object-oriented programming answered to the issue of organising and structuring model data. The programming language, together with inheritance, encapsulation and polymorphism, finally supported the concept of abstract data types.

Abstract data types allowed the programmer to define a closer matching between the concept of a system and its software representation, as shown by Zeigler (1991). A system component, e.g. a population in an ecosystem, was described as a data type (a class) with attributes such as its biomass, and with methods implementing the state transitions and output transformations. Thanks to inheritance it was possible to create taxonomies of models, facilitating both the structuring of modelling knowledge and also the reuse of existing knowledge, by overriding methods in child classes (Del Furia *et al.* 1995). The concept of encapsulation allowed to clearly define the interface of the abstract data type, clearly facilitating the implementation of Parnas' ideas of strong cohesion (what is behind the interface) and loose connection (the interface exposed to other abstract data types. Finally polymorphism allowed implementing different behaviours behind a common interface. The simulation of a composite model could be as simple as calling the same `update()` method on a list containing all the submodels.

Nevertheless, after an initial hype, the relevance of object-orientation to writing good modelling systems has been considerably re-dimensioned (Wehie, 1997). For instance, despite the object-oriented formalism, it was still possible to build monolithic models. A monolithic model is a modelling system where everything depends on everything: the model is interspersed with data, with the numerical integration, calibration, optimisation algorithms, with graphical display and everything is entangled. Most object-oriented modelling systems have been developed as monolithic models.

A paradigm shift was needed once again. Such a shift did not require a major rethinking from the software engineering point of view, but it was simply the acknowledgement that software should be built as any complex piece of engineering, by reusing simpler and robust (in the sense of their quality) components.

2. COMPONENT-ORIENTED SOFTWARE ENGINEERING IN MODELLING FRAMEWORKS

Component-oriented software engineering is a current trend, which places the concept of software component at the centre of the development process. Rewording Szyperski *et al.* (2002), software components are software units, which can be deployed independently, they can be easily re-used by third parties and they do not have an externally observable state. These properties enforce the concept of a component as something different from an object, which has a unique identity (components should be externally undistinguishable), and it has an externally observable state.

Implementing models as components has some clear advantages. Reusability is facilitated by the simplicity of the interface and the limited scope of dependencies from other components. While it is still possible to build components with lots of dependencies and a complex interface, this would fail the first requirement of independent deployment, that is, the ability to deliver components, which are well separated from their environment and other components.

Adopting a well-behaved approach to component-oriented software engineering also reduces the risk of building monolithic applications: your own components should be easy to integrate with third party components. This principle, when applied to modelling, leads to develop model components that are independent of the data processing and visualisation components and where the separation of concerns between model computation and graphical user interface is also clear-cut.

Yet, there are different ways to apply component-oriented software engineering to the implementation of models. We distinguish between *model equation components* and *model application components*.

The straightforward way of developing a component of a dynamic model is to define its interface allowing the user to define the simulation horizon, the sequence of model inputs u(·), the initial state x(0) and the sequence of outputs and states, y(·) and x(·) respectively (see figure 1).

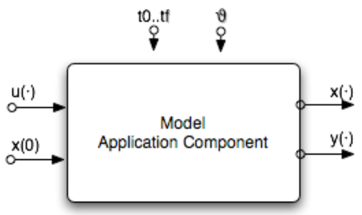

Figure 1. A model application component.

We call this software component a *model application component*. Given the inputs and the parameters, together with the simulation horizon, it is possible to compute the output trajectories. Its interface will allow initialising the model, to set the simulation parameters, and, finally, to call the function that performs the computations.

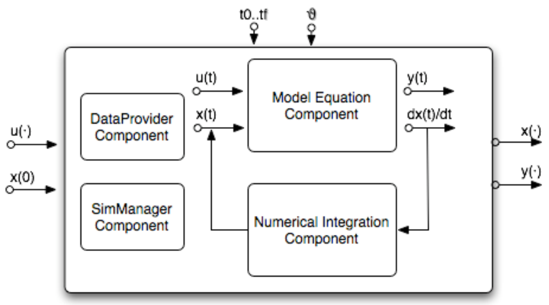

Figure 2. A model equation component embedded into a model application component.

Still, we can decompose further the model execution application, detailing the *model equation component,* which simply computes the rate of change of the state variables and the relative output transformation by means of an update method, and other service components. Such auxiliary components are: the numerical integration component, which integrates the rate of change of the state and feeds it back into the model equation component; the data provider component, which feeds the exogenous inputs u(t) one at a time and optionally stores the outputs and the states, and a simulation control component, required to initialise the model with the initial state and parameters and to manage the invocation of the numerical integration routines.

The model application component is therefore split into the *declarative* part (the equations) and the *imperative* parts (the other components). Such an approach allows for a greater flexibility in term of the development of the models and the simulation algorithms, since these two activities often require different specialist knowledge and this also increases the testability of the smaller and lighter components. Moreover, the reusability of the 'lighter' components across different modelling frameworks is increased, as it will be shown later in this paper.

3. LINKING MODEL COMPONENTS

We define *model linking* as the activity of assembling a set of model equation components together in a composite structure (composite model). A composite model is a complex model that all its sub-models can be simulated simultaneously, and (numerically) integrated in the same time step.

On the other hand, we define *workflow linking*, as the activity of assembling a sequence of model application components, where also the interaction of the user, during the execution of the workflow can take place. In this paper we will focus on model linking, while we refer the reader to previous works on scientific workflows[2] for workflow linking (Lüdascher, 2005). One critical issue in model linking, when assembling model equation components in a composite model, is the difficulty of finding a component design that satisfies the requirement of 'third-party composition'. *My* component must be compatible with *your* component, but more than often this is not the case.

The problem is that component design choices, rather than be peculiar of a specific architecture, should rather promote reusability, selecting design traits which represent a compromise between level reusability and complexity of the design chosen to maximize adaptability of components. Using a pragmatic approach, simplification can be obtained if the target use of components is within a specific

[2] A comprehensive list of software tools for scientific workflows, which are very useful in grid computing applications, is available online at http://www.extreme.indiana.edu/swf-survey/.

knowledge domain; this has an impact not only in simplifying the design of components, but it also clearly defines the scope of the knowledge domain, which is embedded in the modelling exercise, as we will point out in Section 4, where we discuss the role of ontologies in representing modelling knowledge.

Yet, restricting to a knowledge domain has often meant also to restrict to a specific framework, where implementations of model components strongly depend on the modelling framework core. Targeting model component design to match a specific interface requested by a modelling framework decreases its reusability. This can partly explain why modelling frameworks, although in theory a great advance with respect to traditional model code development, are rarely adopted by groups other than the ones developing them.

A possible way to overcome this problem is to adopt a component design, which targets intrinsic reusability and interchangeability of model components (e.g. Donatelli et al., 2005). This may lead, in the worst cases, to the need of a wrapper class (specific to a modelling framework) as proposed by the Adapter pattern (Gamma *et al.* 1994) that makes possible the migration to other modelling frameworks. Nevertheless, the use of appropriate techniques in designing model components interfaces, such as using references to objects as parameters in the interface methods, greatly reduces the overhead due to the extra layer of the wrapper class.

A key design criterion, which enhances reusability and interchangeability, and which allows concurrent development of both components and clients, is separating the model equation component interface and its implementations, in different software units (Löwy, 2003). This is known as the Bridge pattern (Gamma *et al.*, 1994) and it allows defining units of *reusability* (model component implementations and model component interfaces) and units of *interchangeability* (model component implementations alone). As an example application of the concept of separating interfaces and models in the domain of biophysical components, see Donatelli *et al*. (2005b). Note that the model interface is defined by the set of its parameters and input, output, state variables and it is not the model equation component interface, which is the set of methods offered by the software component.

In the model equation interface we define an abstract data type called the *domain class*, following the approach by Rizzoli *et al.* (1998). The domain class is characterised a set of data attributes, which are the inputs, states, outputs and parameters of the model and a set of accessor methods to set and get the attribute values. The data attributes contain the numerical value, the variable's range, the default value, the measurement units. In Section 4, we exemplify how to construct such a domain class from an ontology.

If the implementation of a model component requires data provided by another model, it is sufficient to pass an instance of the domain class of the provider component in the signature of the update method of the receiving component. An example is shown in the diagram of Figure 3.

In the component diagram, `Component1` has a dependency to its interfaces component `Component1.interfaces`. The access method of the component has in its signature a reference to a instance of the domain class A. Let's assume that `Component1` simply reads a data stream from a database and it writes its outputs in `domObjA`, instance of `DomainClassA`.

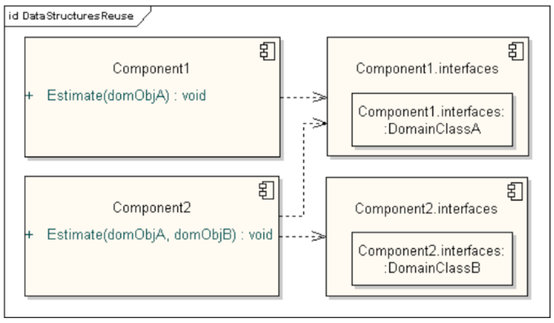
id DataStructuresReuse
Component1
+ Estimate(domObjA) : void
Component1.interfaces
Component1.interfaces: :DomainClassA
Component2
+ Estimate(domObjA, domObjB) : void
Component2.interfaces
Component2.interfaces: :DomainClassB

Figure 3. A component diagram showing the separation of the interface from the implementation.

`Component2` references the `DomainClassA` and using an instance of it in the signature of the update method `Estimate()`. The communication between components is automatically established. Also, there is no dependency among components, and dependencies are to interfaces components only. Components can be replaced and a component linker must primarily check the matching of inputs in a component to outputs of another, in the same domain object.

4. FROM KNOWLEDGE REPRESENTATIONS TO SOFTWARE COMPONENTS

In a component-based approach, assembling a composite model involves the linking of constituent models inputs and outputs. Such an activity can be consistent and sound when models are developed by a small group of modellers. However, common experience has shown that model composition within large developer communities is a struggling task that can easily lead to incoherent results. Reusing "components–off–the–shelf" in environmental modelling is a demanding activity, as usually environmental model components are characterized by poor documentation, insufficient or vague interfaces and suboptimal implementation patterns. Most environmental models have been developed so far without considering reusability and sharing needs as critical requirements of the process. In this sense, although a component binary is by default reusable (in software terms), accessing its interface in a sound fashion (in modelling terms) is a much more complicated task.

Even assuming an effective component-oriented design, most of the problems in component linking tasks emerge due to the poor semantics of the component model interfaces. In the approach presented in this paper the component model interface uses domain classes to describe the model inputs and the outputs classes which also stores information related to variable dimensions, cardinality, units, sampling frequency, model characteristic time, etc.

As model components promote the reuse of models outside a specific framework, in the same way domain classes provide a way to reuse data structures outside the specific domain. Yet, there is a strict dependency on the specific modelling framework. This dependency can be removed. We propose a solution based on declarative modelling and ontologies.

4.1. Declarative models for model equation components

Model equation component can be implemented in source code and they will depend on the specific framework. Wrappers can be written conforming to the Bridge pattern, thus targeting different modelling environments. Yet, model equation components can be successfully designed and implemented adopting the declarative modelling paradigm (Muetzelfeldt, 2004). In fact, an analysis in terms of component architecture does not deny the advantages of using declarative modelling in model building. One key advantage of using a declarative language to store models is the capability to export models according to different implementation requirements and even platforms.

4.2. Ontologies for model interface representation

By analogy, there is no need to write the implementation of domain classes in source code, which is specific to a framework, when we can successfully formalise a common ontology using a representation language such as OWL for defining advanced semantics of a model interface, in order to overcome the common problems described above and to support effective and sound model component linking.

An ontology-mediated approach for defining model interfaces involves the definition of models and their interfaces using a common (public) ontology, where modellers share their knowledge. This ontology could also accommodate the declarative models, but this is not a necessity. A clear requirement is rather to add rich semantics in a model's interface, in terms of model parameters, inputs, outputs, and states (as discussed above), which are publicly available and shared. Through an ontology formalisation, modellers can communicate both knowledge and models independently from their implementation, as their interfaces are no longer tighten up to a particular modelling framework, or programming language, rather they are defined in a independent format in the ontology, that can be later transformed to specific implementations.

4.3. A working example: from ontologies to model interfaces

We focus on the role of ontologies to represent model interfaces and as an example, we present an ontology we developed for biophysical models in agricultural production simulation, within the Seamless project, called SeamAg ontology[3]. We conceptualise the modelling task in three levels: (a) the *Seamless agro-environmental domain* (SeamAg Domain), (b) the *Modelling domain*, and (c) the *Application domain*.

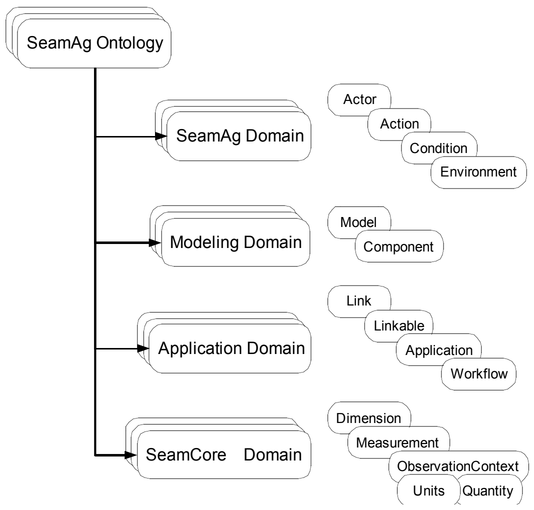

Figure 4. An abstract view of the SeamAg ontology.

[3] A working version of the SeamAg ontology is available at: http://seamless.idsia.ch/ontologies/

Each level (domain) is a discrete perspective of the modeled world, each one orthogonally to the others. The three domains can be seen as complementary abstractions that define links between the three modelling levels. The SeamAg Domain defines all concepts involved within the agro-environmental process, following the Actors – Actions – Conditions – Environment scheme. Components and models are defined in the Modelling Domain, while the Application Domain describes the way model application components are assembled in stand-alone software applications, defining workflows. An abstract view on the ontology structure is depicted in Figure 4.

The SeamAg ontology has been built upon a core ontology, which provides fundamental concepts, related to physical quantities, units, space and time. The SeamAg ontology has been developed in OWL using the Protégé-OWL (Noy et al. 2001, Horridge et al. 2004). OWL is the W3C standard Web Ontology Language, which intends to be used for explicitly representing the meaning of terms in vocabularies and the relationships between those terms (McGuinness and Harmelen, 2004). Note that OWL has more facilities for expressing meaning and semantics than XML, RDF, and RDF-S, and thus OWL goes beyond these languages in its ability to represent machine interpretable content on the Web.

Such a generic-purpose ontology can accommodate the specification, development and exploitation of model interfaces: to describe a model using the SeamAg ontology, we start from the SeamCore domain, that defines Dimensions, Units and Quantities. A Quantity is considered as a concept that can be observed, measured, or computed in Seamless. A Quantity has the following properties: (a) data type (e.g. float), (b) default unit (e.g. Celsius), (c) dimension (e.g. temperature), and (d) domain (e.g. soil). In this respect, we can define a *SoilTemperatureQuantity* that is a *Temperature*, measured in *Celsius*, on *Soil*, and is stored as a *float* number. The *SoilTemperatureQuantity* can be observed in different Spatial and Temporal Contexts, (e.g. as a *time series of hourly observations*, or in several *soil depth levels*), which defines a *Measurement* of Soil Temperature that can be used as a *Model*'s input, output or state.

In such a way, a Model Interface is defined as a collection of *Measurements* associated with its inputs, outputs and states. Wider collections of *Measurements* that are associated with particular domains define the Domain Classes. E.g. we could define the *SoilDomainClass* as the collection of all measurements that are measured on *Soil*. Such collections can be manually entered by a user or they can be automatically built, using the built-in reasoning features of an ontology. Having defined a Model Interface or a Domain Class in the ontology, an OWL file can be parsed to generate the source code of the model interface or the domain class respectively. In this way, a modeler can exploit the knowledge structured in the ontology in different modeling frameworks or different programming languages.

The adoption of an ontology-driven approach for defining a model interface has clear advantages as it enables the reusability of models in a more easy way, while common problems related to poor semantics of model interfaces can be effectively tackled.

5. DISCUSSION

Ontologies should be taken with a grain of salt, since they are not the solution to every modelling and knowledge representation problem, they could be a mean to reach such a solution, but sometimes it looks like they are part of the problem. Often we are torn between taking the side of the 'semantic knight' rather than the 'wily hacker'[4]. Formalising an ontology will only help you in structuring your knowledge, but it will not replace the knowledge engineer. The hacker could be tempted to find 'ad hoc' solutions, based on powerful techniques such as introspection (i.e. reflection), but ontologies are a useful complement to the ability to extract knowledge and structure from the code by introspection. Via reflection we can discover what is stored into binaries, but it is by means of ontologies that we can structure and represent the information we will extract via reflection, such as coherence of units, time steps and check of pre post conditions.

Ontologies also play a fundamental role when model linking happens over the web. The semantic web tries to provide a standard way to automatically discover and run web services. In computational 'grids' for environmental science (Jeffery, 2004) it will be possible to create 'workflows' of web services to execute a given task. Ontologies provide the semantic layer on top of metadata languages such as RDF, thus allowing for 'reasoning' when building such workflows.

Finally, it is worth to remark that, in the majority of cases, and as long as latency will be an issue in modern Internet networks, web services will be

[4] The 'semantic knight' and the 'hacker' are two personae of a parody of a famous Monty Phyton joke made by Michael Champion: http://lists.xml.org/archives/xml-dev/200504/msg00260.html

implemented as model application components, rather than model equation components, since the communication overhead in integrating a model equation component a thousand time for a single simulation step would make this approach practically unusable.

6. CONCLUSIONS

In this paper we have presented an approach to linking models based on semantically enriched model components. The key points of this approach are: design of lightweight model equation components, with no dependency from the modelling framework core; definition of domain classes in the component interface to abstract the dependency of the model from the data and to foster the extensibility of models via design patterns. Finally, the use of ontologies for structuring and representing the knowledge on data structures made possible the automatic generation of semantically rich component interfaces onto which reasoning possible.

7. ACKNOWLEDGMENTS

This publication has been partially funded under 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)

8. REFERENCES

Del Furia, L., A. Rizzoli, and R. Arditi, 1995. Lake-Maker: a general object-oriented software tool for modelling the eutrophication process in lakes. Environmental Software, Vol. 10, No. 1, pp 43-64.

Donatelli, M., G. Bellocchi, and L.Carlini, 2005. A software component for estimating solar radiation. Envrionmental Modelling and Software. (in press).

Donatelli, M., L. Carlini, G. Bellocchi, M. Colauzzi, 2005b. CLIMA: a component based weather generator. Proc. of the MODSIM05 conference, Melbourne, Australia, December 11-15, 2005.

Gamma, E., R. Helm, R. Johnson, J. Vlissides. 1994. Design Patterns: elements of reusable object-oriented software. Addison-Wesley, Boston, MA.

Horridge, M., H. Knublauch, A. Rector, R. Stevens, C. Wroe. 2004. A Practical Guide To Building OWL Ontologies Using the Protégé-OWL Plugin and CO-ODE Tools, Technical Report, Ed. 1.0, The University Of Manchester.

Jeffery, K.G. 2004. Next generation GRIDs for environmental science. In: Pahl-Wostl, C., Schmidt, S., Rizzoli, A.E. and Jakeman, A.J. (eds), *Complexity and Integrated Resources Management, Transactions of the 2nd Biennial Meeting of the International Environmental Modelling and Software Society*, iEMSs: Manno, Switzerland, 2004. ISBN 88-900787-1-5

Löwy, J., 2003. Programming .NET components. O'Reilly & Associates, Sebastopol, CA.

Ludäscher, B., I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, Y. Zhao. 2005. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows*, to appear.

McGuiness, D. L., and F. V. Harmelen (Eds). 2004. OWL Web Ontology Language Overview, W3C Recommendation, Available online: http://www.w3.org/TR/owl-features/

Muetzelfeldt, R.I. 2004. Declarative Modelling in Ecological and Environmental Research. European Commission Directorate-General for Research, Position Paper no. EUR 20918. European Commission, Brussels, B.

Noy, N. F., Sintek, M., Decker, S., Crubezy, M., Fergerson, R. W., and Musen, M. A. Creating semantic web contents with Protege-2000. IEEE Intelligent Systems 16, 2 (2001), 60-71.

Padulo, L., and Arbib, M.A. 1974. Systems Theory: a Unified State-Space Approach to Continuous and Discrete Systems. W.B. Saunders, Philadelphia, PA.

Parnas, D.L. 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, Vol. 15, No. 12, pp. 1053 – 1058.

Rizzoli, A.E., Davis, J.R., Abel, D.J. 1998. A model management system for model integration and re-use. *Decision Support Systems*,Vol. 4, No. 2, pp. 127-144.

Weihe, K. 1997. Reuse of algorithms: still a challenge to object-oriented programming. In: *Proceedings of the 12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, 34-48.

Szyperski, C., Gruntz, D., Murer, S. 2002. *Component Software – Beyond Object-Oriented Programming, Second Edition*. ACM Press, New York, NY.

Zeigler, B. P. 1991. Object-Oriented Modeling and Discrete-Event Simulation. *Advances in Computers,* Vol. 33, pp. 67 – 114.