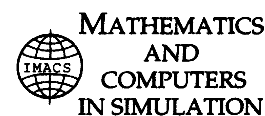
In Mathematics and Computers in Simulation, **78** (**2-3**):412-423, 2008.



Available online at www.sciencedirect.com

Mathematics and Computers in Simulation 78 (2008) 412–423

www.elsevier.com/locate/matcom

Semantic links in integrated modelling frameworks

Andrea E. Rizzoli [a,*], Marcello Donatelli [b], Ioannis N. Athanasiadis [a], Ferdinando Villa [c], David Huber [d]

[a] *Istituto Dalle Molle di Studi sullIntelligenza Artificiale (IDSIA-USI/SUPSI), c/- IDSIA, Galleria 2, 6928 Manno, Switzerland*
[b] *JRC-IPSC, Agri4cast action, Ispra, Italy*
[c] *Gund Institute, University of Vermont, Burlington, VT, USA*
[d] *AntOptima, Lugano, Switzerland*

Available online 18 January 2008

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 mathematical 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 particular between different modelling frameworks. In this paper, we focus on how models can be linked together to build complex integrated models. We stress that even if a model component interface is clear and reusable from a software standpoint, this is not a sufficient condition for reusing a component across different integrated modelling frameworks. This reveals the need for adding rich semantics in model interfaces.
© 2008 IMACS. Published by Elsevier B.V. All rights reserved.

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

1. Introduction

Since researchers in system theory introduced the concept of modular and hierarchical decomposition of models [17], modellers were quick in porting this concept into the software implementations of their models, which were mostly done in FORTRAN. Subroutines were the logical counterpart to submodels, and subroutine arguments 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 [18]: 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 arguments).

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

* Corresponding author. Tel.: +41 58 666 6664; fax: +41 58 666 6661.
E-mail address: andrea@idsia.ch (A.E. Rizzoli).

0378-4754/$32.00 © 2008 IMACS. Published by Elsevier B.V. All rights reserved.
doi:10.1016/j.matcom.2008.01.017

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 provided and early answer to the need of organising and structuring knowledge in models. This programming paradigm, together with inheritance, encapsulation and polymorphism, also 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 representation in software, as shown by Zeigler [24]. 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 [23]. The concept of encapsulation allowed to clearly define the interface of the abstract data type, 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 for the implementation of 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 [1]. 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. While the object oriented paradigm did provide a syntax that had the potential of supporting a better system design, it did not provide enough semantics to suggest and help enforce better design practices.

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. Modelling frameworks were an answer to this need. A modelling framework is a set of software libraries, classes, and components, which can be assembled by a software developer to deliver a range of applications which use mathematical models to perform complex analysis and prognosis tasks. In particular, referring to the environmental modelling domain, frameworks such as TIME [1], OpenMI [2], JAMS [11], and MMS [12] are all successful examples of frameworks where a good balance has been obtained between generality and correspondence to recognizable building blocks for simulations.

Modelling frameworks turned out to be mostly used within the groups that originally developed them, with very little reuse of models developed by other researchers. Yet, the number of implementations of available models is constantly increasing and only recently one modelling framework addresses the issue of model linking (OpenMI [2]). Yet, linking components by wrapping them in a common interface requires a lot of pre-existing knowledge, code must be manually written, and the whole procedure is lengthy and error-prone.

Starting from this observation, we have found that component-based software engineering can be coupled with domain-specific knowledge on how to assemble and combine software components in modelling frameworks. Such knowledge could be formalised by using the *domain class* concept. A domain class can be considered as an abstract data structure for defining a set of a model variables and their attributes [5,19]. 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 runtime 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 model components must expose and share their data structures. And, last but not least, it provides an easy way for linking model components at a higher level, using shared instances of domain classes (domain objects) for exchanging data across models, taking full advantage of component-based software engineering primitives.

Domain classes, describing the interfaces and relationships among models, can be represented by means of emphon-tologies. Ontologies are formal descriptions of a conceptualization of a domain of interest, including a set of statements that define concepts and relationships between them. In an ontology, concepts (also referred to as classes) are linked by *properties*, the statement of an attribute associated with a concept. The value of a property can be another concept as well as a complex object or a textual value. Instances (or Objects) are the statement of an entity "exist" in some – real or virtual – world, and represents the "incarnation" of a stated concept. An ontology that defines a set of instances along with the concepts they incarnate is often called a knowledge base, although the term is not rigorous.

The use of ontologies is advantageous as it (a) supports the automatic generation of code templates for models and domain classes in different 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. A useful ontology usually contains more than just a list of terms and their definitions. Ontologies can be used simply as *controlled vocabularies*, a set of concepts with no properties, whose names define a set of usable terms. A step further is a *taxonomy*, where concepts are arranged in a generalization (*is-a*) hierarchy. A *schema* is an ontology where properties other than *is-a* are defined for classes. A generalization hierarchy may or may not be present.

The main rationale for the existence of ontologies, however, is to enable reasoning on natural systems, either by a human actor or by an automated program (*reasoner*). The main operations in reasoning are *subsumption* (inferring that concept A is more general that concept B) and classification (inferring that instance x is the incarnation of concept A). These uses of ontologies open new doors to natural system modelling that will be explored in the following.

This paper is structured as follows: first we outline the role of component-oriented software engineering in the SeamFrame modelling framework, which is a modelling framework developed in the context of the SEAMLESS project[1]. The next section details how sharing the common knowledge by means of domain classes can support the linkages of model components. Then, the role of ontologies as a formalisation of the modelling knowledge and their role in supporting automatic code generation is described. Finally, we present a working example of an ontology formalisation developed for the SEAMLESS project. 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 can be easily reused, integrated in different environments, or shared with third parties. The potential for 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.

2. Component-oriented software engineering in the seamframe modelling framework

Component-oriented software engineering is a software design and implementation discipline, which places the concept of software component at the centre of the development process. Rewording Szyperski et al. [20], software components are software units, which can be deployed independently, they can be easily re-used by third parties and they do not provide 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 the development of 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. In the context of the SEAMLESS project we have developed a modelling framework, named SeamFrame, where 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 $u(\cdot)$, the initial state $x(0)$ and the sequence of outputs and states, $y(\cdot)$ and $x(\cdot)$, respectively (see Fig. 1).

[1] SEAMLESS is a project financed by European Commission in the context of the 6th framework programme.

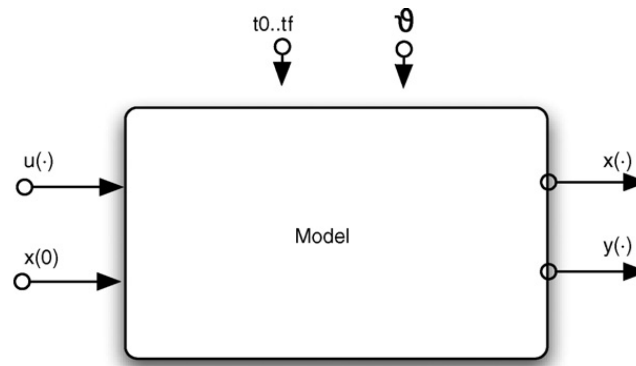


Fig. 1. A model application component.

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 for the initialisation of the model, to set the simulation parameters, and, finally, to call the function that performs the computations.

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 (see Fig. 2). 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 model equation component and the other service components (numerical integration, data providers, etc.). 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 potentially a complex model where all its sub-models can be simulated simultaneously, and (numerically) integrated in the same time step.

On the other hand, we define workflow linking 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 SeamFrame we have adopted OpenMI as the underlying standard to support workflow linking. In this paper, we will focus on model

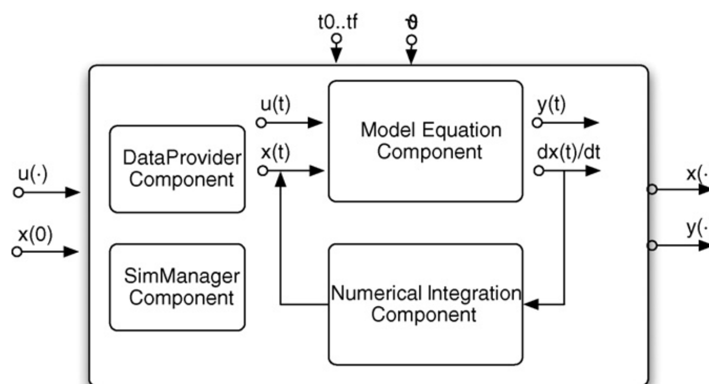


Fig. 2. A model equation component embedded into a model application component.

linking, while we refer the reader to previous works on scientific workflows for workflow linking [14]. 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 often than not this is not the case.

Using a pragmatic approach, simplification can be obtained if components specifically target a given knowledge domain. If the structure of the knowledge domain is made explicit and it can be shared, then it becomes easier to develop cross-framework components. 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, and thus why there are so many of 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. [4,3]). 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 [6] 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 in different software units the description of the data modified by model equation component from the implementations of the model that change the data [13]. This 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 [4].

To describe the data modified and exchanged by a model component we use abstract data types called domain classes, following the approach by Rizzoli et al. [19]. A domain class is characterised by a set of data attributes and a set of accessor methods to set and get the attribute values. The data attributes contain the numerical value, the variable range, the default value, the measurement units. In Section 4, we exemplify how to automatically code such a domain class from its representation in 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. In Fig. 3 we show how model linking works: 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 an instance of the domain class *A*. Lets assume that `Component1` simply reads a data stream from a database and it writes its outputs in `domObjA`, instance of `DomainClassA`.

`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

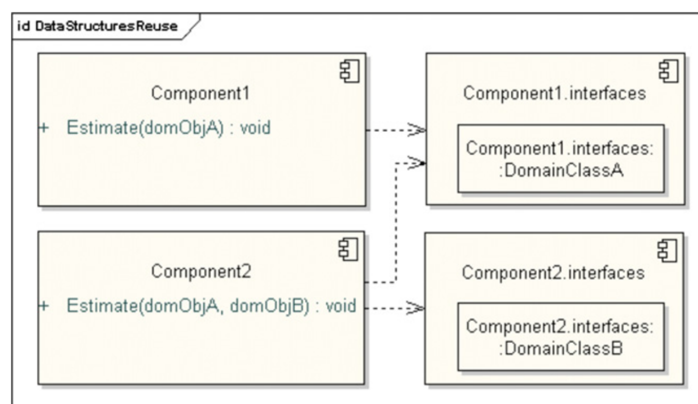


Fig. 3. A component diagram showing the separation of the interface from the implementation.

among components, and dependencies are to the domain classes interfaces 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 class.

Yet, there is an open problem: *my* component depends on the domain class, which is specific to *my* framework. It looks like we have solved the problem by delaying it. It appears that we need a framework-independent representation of knowledge on data structures. Ontologies provide such a representation.

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 interpretation of the 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 outputs, also storing information related to variable dimensions, cardinality, units, measurement 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

When model equation components are implemented in source code they will depend on the specific framework, which provides the data types used to denote the model interface. If we want to reuse such a component in another modelling framework, the Bridge or the Adapter pattern can be used to write wrappers targeting different frameworks. Still, model equation components can be successfully designed and implemented adopting the declarative modelling paradigm [15].

A mathematical model is a set of equations, and we need to translate the model into a set of machine-readable expression in order to solve it. Implementing a model with a procedural programming language requires writing a set of imperative instructions that explicitly define the algorithms solving the set of equations. In declarative modelling, the modeller simply provides a machine-readable specification of the equations, according to a given formalism, and the model compiler or interpreter will apply the suitable algorithm to solve the equations. Various simulation languages provide some sort of declarative modelling language, and Simile[2] and Modelica[3] are examples of modelling frameworks that explicitly support the declarative modelling paradigm.

Declarative modelling can be successfully married to model building according to a component-oriented architecture. 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. The declarative model is a machine-readable expression, which can be automatically processed to generate source code, specific to a platform and a modelling framework. For example, a UML class diagram is a declarative model of a software construct, which can be translated in a series of imperative instructions in a programming language such as Java or C++.

The typical structure of a declarative model consists of the list of model variables, and then the model equations, which define the relationships among the variables. When a variable is defined, the declarative modelling language,

[2] http://www.simulistics.com.
[3] http://www.modelica.org.

```
<?xml version="1.0"?>
<model>
    <input>
        <name>rainfall</name>
    </input>
    <state>
        <name>grass</name>
        <initial_value>100</initial_value>
    </state>
    <state>
        <name>water</name>
        <initial_value>100</initial_value>
    </state>
    <param>
        <name>k1</name>
        <value>0.01</value>
    <param>
    <param>
        <name>k2</name>
        <value>0.02</value>
    <param>
    <rate>
        <name>growth</name>
        <of>grass</of>
    </rate>
    <rate>
        <name>transpiration</name>
        <of>water</of>
    </rate>
    <equation>
        transpiration = rain   k2*grass*water
    </equation>
    <equation>
        growth = k1*grass*water
    </equation>
    <output>growth</output>
</model>
```

Fig. 4. Declarative representation of a plant growth model in XML (adapted from Muetzelfeldt [15]).

besides its role (inputs, states and outputs and parameters), allows the specification of its characteristics, such as type, dimension, unit, range of validity. Recently, XML, the extensible mark-up language, has become a widespread tool to support the representation of declarative models. In Fig. 4, we make an example. A plant growth model, where the rate of change of the biomass G is influenced by water availability W, which in turn is affected by the grass biomass and rainfall R, is expressed by the differential equations:

$$\frac{dG}{dt} = k_1 G W \tag{1}$$

$$\frac{dW}{dt} = R - k_2 G W \tag{2}$$

which are declaratively represented in an XML where self-explanatory tags such as <state>, <param>, <equation> are used to define the role played by the different elements in the model.

4.2. Ontologies for model interface representation

As we use declarative modelling to automate the re-implementation of a mathematical model in specific frameworks, there is no need to directly implement domain classes in source code, again specific to a given framework, since we can formalise the domain class structure by means of an ontology as knowledge representation formal-

ism. Ontologies provide a formal support to express conceptualizations, and a number of tools and representational languages such as OWL support the creation of ontologies. OWL is the W3C standard Web Ontology Language; while XML provides syntax to represent content, OWL provides semantics allowing the description of properties, classes and their relationship [7]. Yet, an OWL document is an XML document, since it uses XML as syntax language.

In the modelling context, ontologies can be used to define the extended semantics of a model interface, in order to abstract from a specific modelling framework and to support effective and sound model component linking. The same ontology used to annotate the model interface could also be used to support the declarative representation of the model equations, but this is not a necessity.

The ontology of a model interface adds the semantics of those parts of the model that are publicly available and shared: model parameters, inputs, outputs, and states. Referring to a shared ontology, modellers can communicate both knowledge and models independently from their implementation, as their interfaces are no longer tied 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. Ontology-driven model specification

Ontologies also enable a deeper definition of declarative modelling, one which relies on ontologies not only to annotate interfaces, but to define the actual logics of a model being developed. Dynamic models, like data, always conform to a conceptualization, and there is in fact no philosophical difference between specifying data or models when this is done using ontologies [21,22]. By consequence, any sort of model can be successfully specified as a set of instances of appropriate ontologies. The main practical difference between data and models is the increased conceptual richness necessary to describe how things change in time and space. This requires at least notions of linkage between concepts with causative or dependency relationships that are normally not necessary when specifying data. It also requires developing ways to interpret this causality. The set of abstractions (concepts) that allows conceptualizing and expressing those cause–effect relationships and their results is the adopted modelling paradigm, of which examples abound (e.g. ordinary differential equations, stock-and-flow, or individual-based). A modelling paradigm, like any consistent conceptualization, can be captured into an ontology. Many existing modelling software systems (e.g. STELLA, SIMILE) conform to one implicit ontology, which defines their notion of entities familiar to the user such as state variables, flux variables, etc. Advanced integrative systems [22] can load different ontologies, which, supplemented by the necessary software, enable them to manipulate models adopting heterogeneous modelling paradigms. Such systems are in the best position to enable integration of independently developed models adopting different paradigms into a higher-level, multiple-paradigm model.

The wording declarative modelling can be used to refer to a specification of models that is based on the attributes and semantics of the natural systems rather than the algorithm that calculates their results. Ontologies support declarative modelling by providing, at the same time, schemata for model declaration and meaning for these schemata. Instances of ontologies represent declaratively expressed models that refer to concepts laid out in the ontologies. Such declarations contain enough information to enable a software infrastructure to simulate the behaviour of the systems represented over a user-defined temporal and spatial extent. Thanks to the rich meaning made possible by ontologies, a workflow environment can properly connect models to data, and feed quantities calculated by simulation to other models in the same environment.

Concept-driven modelling environments can be devised where all concepts used to model a natural system are explicitly defined by standard ontologies, and all technological details related to the calculation of the model are hidden from the user. This approach unifies and outgrows common notions such as metadata and modelling paradigm and opens perspectives of seamless data/model integration and intelligent, hypothesis-specific database querying. These advanced knowledge-based systems (e.g. IMA [21]) are typically not committed to a particular set of concepts except for a core ontology, carefully designed for generality, paradigm neutrality, and extensibility. Their rationale is the notion that an accurate description of natures entities is enough information to allow a system to describe data, calculate and integrate models, as long as enough knowledge has been built into the system to allow their description. The uncoordinated extensibility of such environments allows domain experts to produce knowledge describing specific disciplinary contexts. Users can adopt the correspondent concepts to produce representations of natural systems that the system knows how to resolve into numeric states.

The philosophy of declaring a model can be summarized into laying out (1) reference concepts and properties to define the identity of each modelled entity, and (2) the properties that capture the causal relationships that are the key to distinguish a model from data. Causality in conventional models is usually expressed through equations, defined to calculate the value of variables. Equations, by naming the values of other variables, implicitly define causal relationships that are viewed as dependencies from a processing point of view. An ontology-based framework can make these dependency relationships explicit, and add semantics to them by means of specialization. So, for example, a generic depends-on relationship can be specialized into a flows-into relationship between a state variable and a flux variable (rate), in order to inform the underlying software architecture that the flux must be integrated over time. The notion of variable, so central to conventional approaches, can similarly be enriched and made dependent on the modelled entity. For example, in an individual-based paradigm, variables describe quantitative traits of modelled individuals, but maintain the link to the individual which is the main entity considered. No conflicts need exist between paradigms, whose conceptual boundaries often become blurred when a explicit knowledge-based approach is used, particularly if notions of scale are formally defined [22].

There are limits to the declaration of models in current ontology frameworks, particularly if reasoning capabilities must be preserved. These limitations stem from the fact that any dependency other than linear requires second-order logics statements to be fully formalised, and the handling of second-order logics is beyond the capabilities of existing reasoning systems. The dominant paradigm, Description Logics, can only operate on first-order statements. Even with these limitations, reasoning can be profitably used to enforce correct design and consistent definition of models. As an example, a biodiversity ontology used to model an ecological community can state that coexistence of populations in a community (e.g. as captured in the coexisting-population relationship) also implies coexistence in space and time. Software implementations of the approach can be taught to automatically check that the specification is consistent with coexistence in both space and time before the model can be accepted or calculated. This prevents users from defining inconsistent models and may help retrieving of compatible data sources from databases when the model is applied. Reasoning of such kind can be used to assist proper design and application of a model by enabling a software system to enforce model design disciplines, transcending the mere engineering realm, and therefore facilitating the use of complex simulation models by non-scientists such as decision makers, and ensuring their correct application at the same time.

4.4. A working example: from ontologies to model interfaces

In this section, 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. 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*. The SeamAg ontology has been developed in OWL using the Protégé-OWL [8,16,9].

Each level (domain) is a discrete perspective of the modelled world, each one orthogonal to the others. The three domains can be seen as complementary abstractions that define links between the three modelling levels. The SeamAgDomain 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 Fig. 5.

The various domains of the SeamAg ontology have been built upon a core ontology, the SeamCore domain, which provides fundamental concepts, related to physical quantities, units, space and time. SeamCore defines *Dimensions*, *Units* and *Quantities*. A *Quantity* is considered as a concept that can be observed, measured, or computed in SeamFrame. 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 `SoilTemperature`.

Such a generic-purpose ontology can therefore accommodate the specification, development and exploitation of model interfaces. A *Model Interface* is thus 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*,

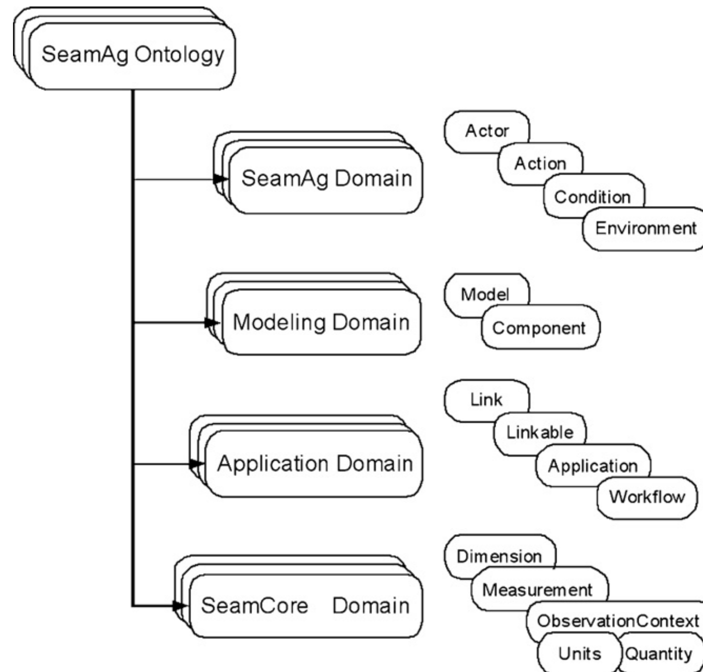


Fig. 5. An abstract view of the SeamAg ontology.

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 modeller can exploit the knowledge structured in the ontology in different modelling 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, while common problems related to poor semantics of model interfaces can be effectively tackled. The SeamAg ontology is developed by more than ten groups in the SEAMLESS project. To support the collaborative work on a shared ontology, we developed a web-based application called AgrOntologies, where a modeller can (a) specify model variables in detail, or even reuse variables defined by others, (b) define model interfaces, and finally (c) put together models in components. A screenshot of the interface is presented in Fig. 6.

AgrOntologies can export an XML file which contains the information required by the application Domain Class Coder (DCC) to generates the C# code of domain classes. A domain class generated by DCC is made by two classes, the first one to hold the variables values, and a companion class to hold the variables attributes. The former is an abstract class to be used as type in the component interface, which then allows extensions via subclassing of its default implementation. The other class, conventionally called with the postfix `VarInfo`to the value class name, contains attribute values, which are declared as static properties and have only the get accessor method. VarInfo values are used by a component to test pre and post conditions which uses the VarInfo type. When these classes are included in a component assembly, its content can be browsed via reflection using the application model component explorer (MCE). This component allows discovering the domain classes, their attributes and types, and the VarInfo values for each attribute.

5. Discussion

Ontologies 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

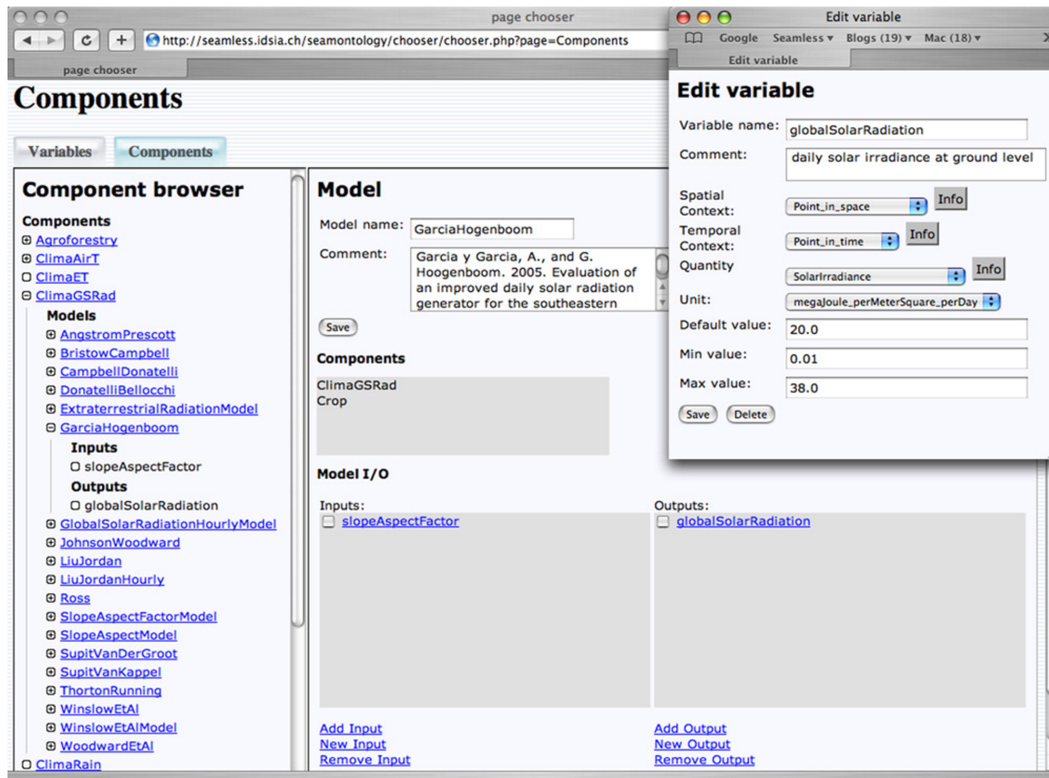


Fig. 6. A screenshot of the collaborative ontology development tool AGRONTOLOGIES.

the side of the “semantic knight” rather than the “wily hacker”.[4] Formalising an ontology will only help you in structuring your knowledge, but you need to possess and be able to use your knowledge to fill the ontology. 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 in 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 [10] 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.

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.

[4] From a parody by Michael Champion of a Monty Python joke appeared on the `xml-dev` mailing list (http://lists.xml.org/archives/xml-dev/200504/msg00260.html).

Acknowledgments

We are grateful to Robert Muetzelfeldt for the valuable comments and suggestions in the preparation of this manuscript.

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).

References

[1] R. Argent, An overview of model integration for environmental applications—components, frameworks and semantics, Environ. Model. Software 19 (3) (2004) 219–234.
[2] M. Blind, J. Gregersen, Towards an open modelling interface (openmi)—the harmonit project, in: C.P. Wostl, S. Schmidt, A. Rizzoli, (Eds.), Complexity and Integrated Resources Management, Transactions of the 2nd Biennial Meeting of the International Environmental Modelling and Software Society, iEMSs, iEMSs, Manno, Switzerland, 2004.
[3] M. Donatelli, G. Bellocchi, L. Carlini, A software component for estimating solar radiation, Environ. Model. Software 21 (3) (2006) 411–416.
[4] M. Donatelli, L. Carlini, G. Colauzzi, Clima: a component based weather generator, in: Proceedings of MODSIM 2005 International Congress on Modelling and Simulation, Modelling and Simulation Society of Australia and New Zealand, MSSANZ, Melbourne, Australia, 2005. pp. 627–633.
[5] L.Del Furia, A. Rizzoli, R. Arditi, Lakemaker: a general object-oriented software tool for modelling the eutrophication process in lakes, Environ. Software 19 (1) (1995) 43–64.
[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley, 1994.
[7] T. Gruber, A translation approach to portable ontologies, Knowl. Acquisition 5 (2) (1993) 199–220.
[8] D.L.M. Guinness, F. van Harmelen, Owl web ontology language overview, W3c recommendation, WWW Consortium, http://www.w3.org/TR/owl-features/, 2004.
[9] M. Horridge, H. Knublauch, A. Rector, R. Stevens, C. Wroe, A practical guide to building owl ontologies using the protégé-owl plugin and co-ode tools, Tech. rep., The University Of Manchester and Stanford University, http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf, 2004.
[10] K. Jeffery, Next generation grids for environmental science, in: C.P. Wostl, S. Schmidt, A.E. Rizzoli, (Eds.), Complexity and Integrated Resources Management, Transactions of the 2nd Biennial Meeting of the International Environmental Modelling and Software Society, vol. 2, iEMSs, iEMSs, Manno, Switzerland, 2004. pp. 491–498.
[11] S. Kralisch, P. Krause, Jams—a framework for natural resource model development and application, in: A. Voinov, A. Jakeman, A. Rizzoli, (Eds.), Proceedings of the iEMSs Third Biennial Meeting of "Summit on Environmental Modelling and Software", iEMSs, iEMSs, Manno, Switzerland, 2006.
[12] G. Leavesley, P. Restrepo, L. Stannard, L. Frankoski, A. Sautins, The modular modeling system (mms) - a modeling framework for multidisciplinary research and operational applications, in: M. Goodchild, L. Steyaert, B. Parks, M. Crane, M. Johnston, D. Maidment, S. Glendinning (Eds.), GIS and Environmental Modeling: Progress and Research Issues, GIS World Books, Ft. Collins, CO, 1996.
[13] J. Löwy, Programming. NET components, O'Reilly & Associates, 2003.
[14] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Taoand, Y. Zhao, Scientific workflow management and the kepler system, Concur. Comput.: Pract. Exp. 8 (10) (2006) 1039–1065.
[15] R. Muetzelfeldt, Declarative modelling in ecological and environmental research, Position Paper EUR 20918, European Commission Directorate-General for Research, European Commission, Brussels, Belgium, 2004.
[16] M.S.N.F. Noy, S. Decker, M. Crubezy, R.W. Fergerson, M.A. Musen, Creating semantic web contents with protégé-2000, Intel. Syst. 16 (2) (2001) 60–71.
[17] L. Padulo, M. Arbib, Systems Theory: A Unified State-Space Approach to Continuous and Discrete Systems, W.B. Saunders, Philadelphia, PA, 1974.
[18] D. Parnas, On the criteria to be used in decomposing systems into modules, Commun. ACM 15 (12) (1972) 1053–1058.
[19] A.E. Rizzoli, J. Davis, D. Abel, A model management system for model integration and re-use, Decision Support Syst. 4 (2) (1998) 127–144.
[20] C. Szyperski, D. Gruntz, S. Murer, Component Software: Beyond Object-Oriented Programming, 2nd edition, ACM Press, 2002.
[21] F. Villa, Integrating modelling architecture: a declarative framework for multi-scale, multi-paradigm ecological modelling, Ecol. Model. 137 (2001) 23–42.
[22] F. Villa, A semantic framework and software design to enable the transparent integration, reorganization and discovery of natural systems knowledge, J. Intel. Inform. Syst. 29 (1) (2007) 79–96.
[23] K. Weihe, 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 (OOP-SLA'97), 1997. pp. 34–48.
[24] B.P. Zeigler, Object-oriented modeling and discrete-event simulation, Adv. Comp. 33 (1991) 67–114.