Languages and metamodels for modelling frameworks

[1]**H. Harvey**

[1]School of Civil Engineering and Geosciences, University of Newcastle upon Tyne, UK.
E-Mail: hamish@hamishharvey.com

Keywords: *modelling, simulation, software, frameworks, languages*

EXTENDED ABSTRACT

The problem of decision making in environmental management is increasing steadily in complexity. A variety of pressure is leading to a requirement to consider the impacts of possible courses of action on a much wider basis than heretofore. Modelling and simulation is becoming ever more fundamental to the decision making process. The models are becoming more complex, as is the context in which they are used. In flood defence asset management, for example, we see deterministic models of water flow being coupled with probabilistic models of flood defence reliability, and the result set within a calculation which derives distributed measures of economic risk, attributing that risk to particular parts of the flooding system.

Even to provide the deterministic modelling element of next generation decision support systems a more modular approach to software construction is needed than that used to build the modelling systems of the past. This is recognised in the effort being invested in developing I will refer to here, somewhat indiscriminately, as "software frameworks".

The design of a software framework consists in large part in the design of a *language* or of a set of languages. Languages in general are media for the expression of thought (and thus for communication) but, as George Boole observed, they are much more profoundly "instrument[s] of human reason": without them abstract thought is not possible. Formal languages enable a level of precision in thought and communication which cannot be achieved using informal languages. With well designed notation they also enable a level of *con*cision which (Alfred North Whitehead), "By relieving the brain of all unnecessary work … sets it free to concentrate on more advanced problems". These benefits come at a cost, though, of effort in formalisation of concepts, and barriers to change once they are formalised.

All languages which control the behaviour of our digital computers, which we may refer to as *computer-based* languages, are necessarily *formal* languages. Like all languages, computer-based formal languages are *tools* for use by humans. They are media of human–computer communication, tools by means of which humans condition the behaviour of the computer. They are also however, and with at least equal importance, tools for human–human communication and tools of thought.

Language design is difficult; it involves creating an conceptual apparatus for thinking about a class of problems. To avoid embedding assumptions, such as abstractions similar to those provided by familiar languages, requires great effort. The scope of such a language must be very carefully chosen. Too general, and the language will provide little advantage of a general purpose programming language. Too narrow, and the need to integrate frameworks, already evident in papers in these proceedings, arises; this is potentially more problematic than integrating modelling systems in the first place. Changing a language once it is in use leads to conflict between conceptual parsimony—critical to its performance as a tool of thought—and backward compatibility; too often, parsimony loses.

The languages defined by modelling systems must deal with concepts in at least two, sometimes three distinct domains of abstraction. Since their purpose is simulation, one of these is the domain of computation. To this a system must add abstractions from either a modelling paradigm (for example the stocks and flows paradigm popular in operations research and ecological modelling), or from an application domain, or both. The raison d'être of a framework over a monolithic modelling system is extensibility. A framework allows functionality to be packaged in software components, so adds a fourth, software-architectural domain to the set of concepts to be manipulated. Furthermore, the languages defined by a framework must themselves be extensible, and the software-architectural abstractions chosen have implications for the design of all other domains.

These four domains are currently rarely considered explicitly, let alone separated in implementation. Frameworks often mix abstractions belonging to two or more domains in a single language, even conflating abstractions from different domains. The current glut of modelling frameworks, of which often none quite fit the task in hand, can in part be ascribed to this. The treatment of each domain in a separate language in a layered framework offers a route forward.

1 INTRODUCTION

Abbott (1991) describes the development of hydraulic modelling as it has proceeded through four generations. The latest, fourth generation corresponds to the commercial "modelling system" which made the power of computational hydraulics available even to those not initiated in the arcana of its methods. By providing hydraulic modelling facilities as a software *package*, a whole new industry was spawned to develop and support the use of such tools.

Fourth generation tools have been in use for the last twenty-some years, and are now beginning to show their age. Meanwhile, while hydraulic modelling was undergoing refinement and commercialisation, a number of complementary modelling communities have grown up. Demand is now growing for tools which enable the development of models which integrate the productions of these various communities in support of decision making in environmental management.[1] These demands are driven by many factors, including developments in public expectations for the management of the environment, legislation, uncertainty theory, and available computer power.

The scope of the required tools is such that it is infeasible to develop them as monolithic applications. As a result, interest in software supporting a modular approach to environmental modelling has surged. Several such facilities, which we will refer to generally as "frameworks", have been developed. This diversity is to be expected at this relatively early stage, as is the fact that the all of these frameworks are quite limited in scope. As a result, new projects are liable to consider the available options and then decide to build another framework, finding that those on offer all fall short in one way or another. It can only be a matter of time before work begins on framework-integrating frameworks, if it has not begun already.

The process of developing modelling tools and systems before the rise of frameworks was, at the software-architectural level, largely empirical, even while the knowledge encapsulated within these tools was firmly grounded in theory. The "casting about" that is being exhibited in the field of framework design, however, stands as early evidence that this empirical approach is here less effective. A primary reason for this lies in the most profound difference between a monolithic modelling system, however integrated, and a modelling framework: the framework, being open to independently developed extensions, must contend with what in knowledge representation circles is called an "open world". The developer of a monolithic modelling tool can rely on perfect knowlege of all of the parts which must function together as a system. The developer of a framework, however, is limited to *imposing restrictions* on the developers of components for that framework, in order that *something* can be known about those components.

The task of deciding *what* restrictions to impose, however, is exceedingly difficult. Because each set of framework developers have a different background, they bring a different set of assumptions to the task, for example about the nature of models and the uses to which they will be put, and about the nature of software components and the context in which they will be deployed.

A more theoretical analysis of the situation can pinpoint the source of some of the limitations of current frameworks and help avoid the fossilization of unchecked assumptions—and the corresponding imposition of avoidable limitations—in frameworks of the future. This paper offers the beginnings of such an analysis, starting with the observation that *every modelling system implements one or more formal, computer-based languages*. These languages are generally implicit in the design of the system, but are manifested in the configuration files for the system and, if it exists separately from these, in its user interface.

Section 2 discusses the nature and roles of language, while section 3 provides a brief summary of the structure of formal languages. In section 4, a number of properties with which languages can be differentiated are introduced, including the distinction between imperative and declarative languages. Declarative languages enable the representation of concepts. Section 5 identifies a number of different *domains of abstraction* from which these concepts are drawn, and explores the structure of existing classes of modelling system and framework using the conceptual framework established. Section 6 concludes with a discussion of some implications of the language-centric view of modelling frameworks.

2 LANGUAGE

Humans communicate by expressing ideas using language. Everyday communication takes place using spoken or written natural language, along with gestures, body language and so on. Language is more than a tool for communication, however. George Boole (cited in Iverson, 1980) wrote, "That language is an instrument of human reason, and not merely the expression of thought, is a truth generally admitted." The capacity to form language is intricately bound up

[1] At the same time, there is an increasing though less widely recognised need for tools which allow such models to be used within a comprehensive *decision analysis* framework. The conceptual framework developed in this paper extends readily into this domain, but the issues involved will not be discussed here.

with that for abstract thought.

Formal languages, such as the languages of mathematics, trade the flexibility of natural language for increased precision. When participants in communication share knowledge of such a language, they can reliably and efficiently communicate any complex concepts of the kind which that language is optimised to express. The use of dedicated notations, again as in mathematics, also enable a level of *con*cision which (Alfred North Whitehead), "By relieving the brain of all unnecessary work ... sets it free to concentrate on more advanced problems". So the ability to *invent* specialised languages is the foundation of our ability to manage complexity.

The computer is an uncompromisingly formal device, and if it is to be a party to communication, then the language in which that communication is to take place, which we will refer to as a *computer-based* language, must of necessity be a formal language. The inflexibility of the computer in fact means that computer-based languages must meet much more demanding requirements than other formal languages. The need to accommodate the limitations of the computer, combined with the dramatic value to be obtained from having the computer work on some problem if that problem can be reduced, in whole or in part, to calculation, tend to obscure the fact that computer-based languages are still just as much tools invented by and for humans as other formal languages. Abelson and Sussman (1996) suggest that, "Programs must be written for people to read, and only incidentally for machines to execute." That is, programming languages must function as media of human-human, as well as human-computer, communication.

A language provids a set of concepts, or abstractions, and the use of that language involves a commitment to the description of the universe of discourse in terms of those abstractions. In natural languages, cultural differences in the classification of the things in the world are reflected in language: consider that it can be hard to say certain things in certain languages. The Sapir-Whorf hypothesis[2] states that our native language, in providing a basic set of abstractions, *controls* the way we divide the world into concepts.

The most readily identified computer languages are programming languages. Considerable, if largely anecdotal, evidence exists that programming languages also conform to the Sapir-Whorf hypothesis: programmers experienced in substantially different languages (structured and object oriented or, more fundamentally, imperative, functional, and logic programming languages) see the same problem in very different ways. Natural language is extraordinarily flexible, and humans are capable of learning new concepts from definitions and descriptions. In contrast, formal languages are rigid, and, being external to their human users, resistant to refinement through learning processes. Thus formal languages limit what their users can express, impose constraints on *what we are likely to think*, and do so in an inflexible, unforgiving way. This is a significant issue in process modelling; it in a decision making context, where inflexibility can prejudice the decision making process.

[2] `http://www.wikipedia.org/` has an informative article on the S-W hypothesis.

3 FORMAL LANGUAGE STRUCTURE

Formal languages have a well defined structure which includes *syntax* (form, notation) and *semantics* (meaning). Harel and Rumpe (2000) provide an unusually clear exposition of these concepts, while Van Roy and Haridi (2004) provide a more comprehensive account from the perspective of programming language design. The syntax of a formal language can be further subdivided into its *concrete* syntax—the physical (usually visual) representation of expressions—and its *abstract* syntax—their structure. The semantics of a language is then defined in terms of its abstract syntax.

More than one concrete syntax may be defined for a given abstract syntax; textual and graphical concrete syntaxes may both be defined, for example, with both able to represent the same abstract structures. The only concrete syntax which is necessary in a computer based language is in fact the representation in memory of the abstract syntax. Figure 1 shows two examples of concrete syntax for a simple calculator language which might both correspond to the same instance of abstract syntax.

```
1:  3**2 + 4**2;
2:  (+ (** 3 2) (** 4 2))
```

Figure 1. Examples of concrete syntax (line numbers for reference only)

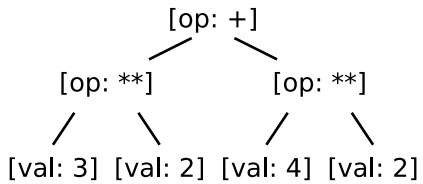

Figure 2. Depiction of corresponding abstract syntax

The notion of an abstract syntax is rather difficult to grasp, but very important. It is difficult primarily because, being abstract, instances of it can only ever be *represented* using some concrete syntax. Figure 2, for example, shows an example of a third concrete

syntax, in this case a graphical syntax, for the same language. This graphical syntax conveys better the structure of the abstract syntax used for most expression-based languages.

The semantics of a formal language ascribe a single meaning to every syntactically legal expression in the language. Formal language semantics can be specified in a number of ways, each being suited to different types of language and useful for different purposes. The combination of abstract syntax and semantics—whether formally defined or informally intuited—is often referred to as the *metamodel* of a language. Most users of computer-based languages work on the basis not of a detailed understanding of a formal semantics, but of a more-or-less informal "feel" for the meaning of the elements of the abstract syntax, for the metamodel.

The power of language as a tool of thought is shared between the metamodel and the concrete syntax. The metamodel is fundamental; it provides the abstractions with which users of the language must work. The syntax is then a *notation* for those abstractions. A well designed notation can make the manipulation of complex ideas much easier. It acts, as it were, as an amplifier of the power of the abstractions.[3] Iverson (1980) discusses the importance of notation as a tool of thought in the context of programming language design.

4 TYPES OF LANGUAGE

The most obvious examples of computer-based formal languages are programming languages, but there exist a great many computer-based languages which would not normally be thought of in this way. Since every computer-based language is used to condition the behaviour of a computer in some way, however, the distinction between those which do and those which do not deserve the name "programming language" could be seen as rather artificial.

Part of what lies behind this distinction is the contrast between *imperative* and *declarative* languages. An imperative language is one which primarily instructive, in which the language elements constitute components which can be combined into a *script*, which then directs the operation of a machine. The semantics of the language elements are defined in terms of the behaviour they will induce in some machine. A declarative language is primarily descriptive, or representational; the semantics of elements are defined in terms of the things or concepts they represent.

The languages most familiar in scientific and engineering computing, including object oriented languages, are imperative. Machine code is purely imperative, and a computer can only be induced to do something by triggering the execution of an imperative program. Part of the power of object orientated programming languages lie in the fact that such languages encourage the treatment of the computer's memory in a representational manner. These languages are themselves not declarative, however: the programmer must still write imperative code to explicitly construct such in-memory representations.

An imperative program can read information encoded in a declarative language and base its behaviour on that information; similarly it can write an encoding of computed results in a declarative form. The formats of the configuration, input, and output files of modelling and simulation software are declarative languages, even if much of the information they hold is encoded implicitly through position.

Computer-based language may be differentiated in terms of their *expressive power*. Two concepts are easily confused here, however. Given two languages, one may be capable in an absolute sense of encoding a wider range (or simply a different range) of concepts. Alternatively, they may be exactly equivalent in this absolute sense, but different in a relative, and possibly subjective one: in that the human user finds it easier to express a given concept in one than in the other.

Either or both of these two types of expressive power may lie behind the designation of a language as *domain specific*, as distinct from *general purpose*. A domain specific language, or DSL, is a language optimised for some particular purpose. This optimisation may go so far as to limit the absolute expressive power of the language, or it may simply prioritise certain constructs of importance to the domain at the expense of others. The configuration file format of a one-dimensional hydraulic modelling tool is a DSL for describing the abstractions involved: river reaches and cross sections, for example. The Modelica[4] language and both the graphical and equational languages of VenSim[5] are DSLs for expressing models in particular modelling paradigms.

Expressive power in its relative sense is related to the idea of *level of abstraction*. In imperative programming languages, level of abstraction runs from machine code, through languages like FORTRAN, C,

[3] Consider, as a simple example, the concepts of the set of natural numbers and of set membership. These can be used independent of any particular dedicated notation using natural language. They are much easier to use, and to combine with other concepts into even more powerful composites, when we can indicate that the value of a variable x must be a member of the set of natural numbers by simply stating "$x \in \mathcal{N}$".

[4] `http://www.modelica.org/`

[5] `http://www.vensim.com/`

and C++, to object oriented languages with automatic memory management, and on. Declarative languages, by "abstracting away" the computer as a calculation device entirely, can be said to inhabit a higher level of abstraction than any imperative language.

A language may or may not be *extensible*, which property is a critical difference between the languages defined by a modelling system and those of a modelling framework. The latter must be extensible because the framework is itself by definition extensible. An extensible language should not be confused with a language framework. XML is not a language, it is not even strictly a language framework; it is rather a *syntax* framework. Specific languages based on XML may or may not be extensible; the XML framework provides for the definition of extension points but these must be explicitly used.

Languages may exhibit different types and degrees of extensibility. Programming languages provide an easily developed example: many structured programming languages are extensible only by the addition of functions or procedures; object oriented languages allow the limited definition of new data types; some functional programming languages allow much more advanced type system extensions; languages in the Lisp family are quite unique in allowing syntax extension. A similar range of possibilities exists with declarative languages.

A particular class of extensible, declarative languages of increasing interest in model-based decision support as in software engineering generally is that of *knowledge representation* languages. Sowa (1999) provides an extensive survey of the field of knowledge representation.

5 DOMAINS OF ABSTRACTION

When designing a modelling framework, what are the languages which are defined, and what sets of abstractions should they provide? An examination of modelling systems and frameworks reveals at least four distinct groups of such abstractions: those belonging to the domains of *software architecture*, *computation*, *modelling*, and *application*. These domains represent different types of knowledge and a modelling framework will draw on one or more of them depending on its purpose.

Modelling systems built for specific purposes often provide a user interface defined at least partly in terms of application domain concepts. The user of a one-dimensions hydraulic modelling system defines a network of pipes and channels using a map-based interface, for example. Indeed it is this presentation of application domain concepts in the interface—making computational hydraulic knowledge accessible to specialists in the application area—that characterises the fourth generation hydraulic modelling tools mentioned in section 1. Such tools internally define a metamodel, which is then expressed in the concrete syntaxes of the graphical and textual devices used in the user interface and the configuration file formats for the tool. This is an *application domain* language.

Modelling languages which are independent of application domain make manifest the abstractions of a particular modelling paradigm or consistent set of modelling paradigms: they are *modelling domain* languages. VenSim, for example, offers graphical and equational languages in which to define models using the stocks and flows paradigm of system dynamics modelling, while Modelica supports the description of models using differential, algebraic, and discrete equations.

The abstractions of both of these domains are essentially representational. Computer-based modelling, meanwhile, is largely directed towards the *simulation* of the resulting models. Every modelling system must therefore define a mapping from either application or modelling domain abstractions into *computational abstractions*.

As we move from monolithic modelling systems to extensible modelling frameworks, *software-architectural* abstractions to do with the modularisation of software are introduced. "Plug-in components" of any kind, to be useful, must introduce functionality defined in terms of abstractions in one or more of the other three abstraction domains. The languages dealing with these domains must therefore be extensible, and these extensibility mechanisms must be designed together with the component architecture.

While these various domains can be distinguished, they are rarely if ever used in isolation. Most modelling software defines abstractions in at least the computational and one of the representational domains. Perhaps the only exception is in the earliest simulation activities, which were undertaken by writing custom, one-off simulation programs. In this case, the programmer translates manually from a—mathematical, so formal and declarative, but not computer-based—model to a simulator expressed using the abstractions provided by the chosen programming language (usually FORTRAN). The resulting software contains only what might be called the "fossilized remains" of the application and modelling abstraction domains: for example references in the names of functions, constants and variables.

All but the most primitive application-specific simulation software can simulate a *class* of systems.

Such software is defined primarily in terms of computational abstractions. A specific simulator is created by combining the software with a configuration specifying the particular system to simulate in terms of application domain abstractions. Modelling domain abstractions are present only in fossilized form, having been used by the programmer in the process of writing the software. Much software developed in academia falls in this class, as do fourth generation computational hydraulic modelling systems.

Modelling languages such as Modelica and the VenSim language mentioned above, which are quite independent of application domain, have by definition metamodels which include only modelling domain abstractions. The tools which implement these languages define a mapping between these abstractions and computational domain abstractions, allowing translation according to this mapping to generate a simulator. No application domain abstractions are present in such languages; the names used by the model developer to label instances of the modelling abstractions serve as surrogates.

Many application-specific modelling systems are not extensible, and the languages they define are correspondingly fixed. Modelling languages are generally extensible in a limited way. New instances of modelling constructs can be defined as composites of existing instances, for example. This object level extensibility is equivalent to the definition of new functions in a standard programming language. It does not allow the addition of new modelling abstractions. By supporting plug-in components, a modelling framework can enable this and many other types of extension.

6 IMPLICATIONS

The foregoing analysis provides some insight into the current state of modelling framework design, and suggests directions for future development. Most importantly, while it is generally recognised that carefully defining the scope of a framework before starting work on it is important, this is not simply a matter of being careful not to "bite off more than we can chew", or of limiting the cost of framework development. A firm basis is needed on which to decide questions of scope. Viewing framework design as a language design problem helps provide this basis.

The term "component" is in common use, but means very little. The vague idea that it conveys must be solidied through very many decisions, from the abstract to the implementational levels. A common use of plug-ins in a modelling framework is in defining computational modules. This form of extensibility is of particular significance when a lot of legacy simulation code exists. The nature of components at the software-architectural and computational levels can have profound impacts at higher levels, however. The concept of a component in many modelling frameworks—imbued with the character of the often object-oriented, always imperative implementation language—is that of a machine. For composing simulators this may suffice, but it is not clear that it provides a sound conceptual basis on which to build a new generation of model-based decision support tools.

Scope should be chosen to maximise the conceptual parsimony of the languages defined by the framework. As Einstein said, "Everything should be made as simple as possible, but no simpler." The second clause of this paraphrase of Ockham's Razor is crucial: it is the difference between naive simplicity and parsimony. The most powerful, flexible languages are those with a few core abstractions, carefully chosen for orthogonality. Languages generally become less flexible and less powerful with the addition of features, unless those features are defined in terms of such a core. The need for parsimony is partly because these languages are to be used by humans as tools of thought. The properties of complex languages can be effectively impossible to reason about. Even for the designers of a language, parsimony is the only way to ensure coherency and completeness.

The arbitrary circumscription of scope is to be avoided. Limiting scope in this way can make powerful abstractions impossible to find, and results in the creation of multiple frameworks with overlapping purposes. When the facilities of two or more of these frameworks are needed, two options exist: to develop another framework with the union of their capabilities, or to integrate the existing frameworks. This integration problem is even more severe than that of integrating simulation software, and is likely to result in a hybrid monster cut through with inconsistencies, overlapping abstractions, and limitations. As the problems we contend with get more complex, and more facilities must be integrated within a single tool, this issue will become ever more important. As an example, many current modelling framework designs do nothing to support, or even in some cases actively inhibit, the sorts of calculations involved in quantitative risk analysis.

The domains of abstraction identified above provide a conceptual framework within which to identify sets of abstractions which are complete and parsimonious. Their recognition allows us to reason about language scope "vertically" as well as "horizontally". *Separation of concerns*, a fundamental tenet of programming, suggests that abstractions from the different domains should be treated in separate languages or language "layers", with explicit mappings defined between

abstractions in these layers.

In the layered set of languages suggested by this view, the abstractions of various modelling paradigms would be defined by providing mappings from the language encoding these abstractions into a language encoding a set of computational abstractions. These would be general enough to allow one computational language to be used as the target for many modelling paradigms, and even for the more general computational tasks involved in model application. This greatly eases integration, and allows a single set of paradigm-agnostic model analysis tools to be developed. Domain abstractions can be mapped into modelling paradigm abstractions or, as appropriate, directly into computational abstractions. The latter case arises when an established computational hydraulic code is integrated with the framework, in which case the relevant modelling paradigm abstractions are "fossilized" in the legacy code.

Abstractions from different domains are often conflated in thinking about modelling systems and frameworks and mixed in their implementation. The failure to separate these layers, and to ensure the parsimony of each layer, leads to frameworks which have unexpected limitations, often as a result of embedding the tacit assumptions of their developers. This in turn leads to the current situation where potential framework users evaluate and reject all of the increasingly large number of available frameworks, choosing instead to develop another.

These languages or language layers must be defined in some way. Whereas in modelling systems the languages defined are largely internal to the system, in a modelling framework they are tools of component developers and integrators. A common approach is to build on the facilities provided by the implementation programming language. This approach was taken by the OpenMI framework developers, using the C# language. In contrast the Integrated Modelling Architecture (IMA) (Villa, In press) implements a declarative language, building *on* rather than *in* the implementation language (C++). The IMA modelling language is an *ad hoc* language with an XML-based concrete syntax. Framework developers are beginning to explore the possibilities offered by logic-based knowledge representation languages, which have strong advantages in allowing all domains of abstraction to be defined as well segregated components of a single language.

Whichever approach is taken, the framework developer should be aware that with a choice of language platform comes a set of *ontological commitments* (Davis et al., 1993). In simple terms, this means that the language platform imposes some underlying abstractions, and limits the developer to defining languages which involve specialisations of these. With imperative (including object oriented) languages, for example, these commitments tend to lead to a conception of a modelling framework which is biased towards the process of simulation, rather than the essentially human process of modelling and model use. Many KRLs, meanwhile, do not support the representation of uncertain quantities.

In taking up this language-centric perspective on modelling frameworks, excessive formalisation should be avoided. While everything to be manipulated by computer must be formalised, informal representations are capable of vastly greater subtlety than any formal structure. Formalisation is difficult, and should only be undertaken if the rewards are commensurate with the effort involved (the reward for formalising a model, the ability to simulate it, clearly often is). The risk in excessive formalisation when modelling is conducted in support of decision making is to severely prejudice the decision making process.

References

Abbott, M. B. (1991), *Hydroinformatics: Information Technology and the Aquatic Environment*, Avebury Technical, Aldershot, UK.

Abelson, H., and G. J. Sussman (1996), *Structure and Interpretation of Computer Programs*, MIT Electrical Engineering and Computer Science, The MIT Press, second edition.

Davis, R., H. E. Shrobe, and P. Szolovits (1993), What is a knowledge representation?, *AI Magazine*, 14(1), 17–33.

Harel, D., and B. Rumpe (2000), *Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff*, Technical report, Mathematics & Computer Science, Weizmann Institute Of Science, Mathematics & Computer Science, Weizmann Rehovot, Israel.

Iverson, K. E. (1980), Notation as a tool of thought, *Commun. ACM*, 23(8), 444–465, doi:10.1145/358896.358899.

Sowa, J. F. (1999), *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Course Technology.

Van Roy, P., and S. Haridi (2004), *Concepts, Techniques, and Models of Computer Programming*, The MIT Press.

Villa, F. (In press), A semantic framework and software design to enable the transparent integration, reorganization and discovery of natural systems knowledge, *Journal of Intelligent Information Systems*.