Metapattern in Context

for Logius (Peter Waters)

Jeff Rothenberg

RAND-Europe

May 14, 2010

1. Introduction

It has been said that if you fall asleep in a civil engineering class and wake up realizing that the professor has just asked you a question that you did not hear, a safe answer is "drainage".

One candidate for a similarly universal answer in computer science might be "representation". In order for computers to be useful, they must be made to represent real world entities, occurrences, and concepts and to manipulate these representations so as to help us understand, predict, manipulate, and interact with the real world. These representations run the gamut from concrete bits, numbers, text, objects, events, and procedures to abstract relationships, knowledge, processes, and concepts.

Metapattern, developed by Pieter Wisse, is a representational technique aimed at the conceptual modeling of contextual relationships.[1] As such, it is useful for analyzing and understanding complex real-world systems and processes whether or not that understanding is intended to lead to the design and implementation of ICT systems. This paper attempts to situate the Metapattern approach within the landscape of other representational approaches that have been developed within computer science, discusses its relationship to these approaches, highlights its advantages, and suggests how it may be put to the best use.

2. Background

For the past 50 years, the leading edge of computer science research in representation has been the sub-field of artificial intelligence (AI). AI researchers find it difficult to agree on exactly what AI is, but an operational definition might be that AI attempts to enable computers to do things that they are not yet able to do. Although one of the main motivations for AI is to model and understand how human intelligence works, an equally strong motivation is simply to create computer systems that do useful things that were previously considered the sole domain of humans, such as recognizing printed text, responding to voice commands, driving robotic vehicles around the surface of Mars, or planning optimum driving routes from one town to another. These two motivations are often conflated within AI and have led to widely differing approaches within the field.

[1] Note that the term "Metapattern" is usually attributed to Gregory Bateson, who used it to mean literally patterns of patterns. The term is used here as defined by Wisse, not Bateson.

AI languages and representational techniques tend to fall into two distinct groups, one based on formal logic and the other on more ad hoc approaches. However, these do not necessarily map to the above distinction between modeling and pragmatic problem solving motivations. Neither do these two groups of techniques map cleanly into the ability or inability to perform automated reasoning: although rule-based techniques may be based on formal logic, frame-based and other techniques are also typically associated with some form of reasoning. Lisp, the paradigmatic AI programming language, has a formal foundation (in the lambda calculus) but is rarely used in a purely applicative style; and although it can be used to represent formal logical assertions, this is not a precondition for subjecting Lisp code to automated reasoning. In general, AI languages exhibit a mixture of support for both automated reasoning and the implementation of practical computer systems, corresponding to the conflated motivations for AI, described above.

Over the decades, the goals of AI have evolved, as earlier problems were solved and became part of mainstream computer science practice. Where it was once difficult to represent and manipulate formal logic statements, linguistic expressions, music, voice, handwriting, or complex structured data, all of these have (in varying degrees) succumbed to techniques that originated in AI and were subsequently appropriated and refined by software engineering. Along the way, AI has also played a major role in the development of programming language concepts, introducing the functional or "applicative" programming paradigm in Lisp (based on Church's Lambda Calculus) over 50 years ago and the logic programming paradigm in Prolog nearly 40 years ago. Current AI challenges include such problems as understanding and translating "natural" language (English, Dutch, etc.), recognizing complex visual or audio patterns, constructing and adapting sophisticated plans, automating the programming process itself, performing common-sense reasoning, controlling robots, coordinating the actions of multiple agents, and supporting or replacing human decision making. Because AI continually redefines its goals, setting its sights on problems that it cannot yet solve and losing interest in each one just as its solution begins to emerge, it has achieved an undeserved reputation for failure, as embodied in the sarcastic edict "True AI is five years away—and always will be!" Yet a considerable body of current computer science practice owes its success to efforts that began in AI. With respect to representational techniques, AI has a long history of developing novel approaches to representing knowledge, processes, and rules of inference, having pioneered such fundamental methods as list processing, tree search, frames, rule-based systems, case-based reasoning, roles, agents, neural nets, and many others.

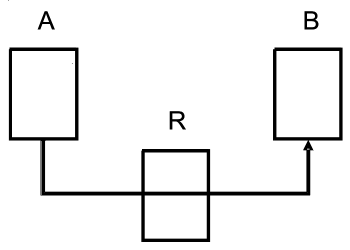
A second sub-field of computer science that is deeply involved in representation is data modeling. This involves the development of data structures, methods of representing relationships, and techniques for diagramming and designing complex representational objects for use by computer systems. The advent of relational database theory, entity-relationship (ER) models, and object-oriented databases are some of the most valuable contributions of this field to computer theory and practice. Relational concepts have also found their way into some programming languages (such as the above mentioned Prolog), while object-oriented concepts appeared originally in Simula (and the AI language Smalltalk) and have more recently become mainstream techniques in Eiffel, C++, Java, etc. Beyond the specific techniques that these formalisms offer, they have also created a wealth of important and useful concepts, such as relational database normalization, keys, and joins. Unlike AI, data modeling has been highly successful at delivering concrete, practical techniques and systems that are in everyday use by huge numbers of users, while (perhaps even more significantly) managing to retain credit for most of its contributions.

It is far beyond the scope of this paper to provide a survey of AI or data modeling. Instead, I attempt to show where the Metapattern formalism fits within related efforts in these and other sub-fields of computer science, as a way of identifying Metapattern's uniqueness and value.

3. What is Metapattern?

The reader is assumed to be familiar with the Metapattern approach; this paper is not intended as an introduction to it or a tutorial on its use. Metapattern is described by Wisse as a conceptual information modeling technique. It is intended to be used primarily for analyzing complex problems rather than designing systems to solve such problems. In [Wisse] Metapattern is specifically aimed at modeling context and time. The recent use of the approach by GBO for modeling semantic issues involved in creating eGovernment systems focuses on context, which is seen as the source of semantic variation among terms, concepts, procedures, and policies across distinct government agencies and systems. (To the extent that temporal issues can be thought of as contexts themselves, this subsumes time within context. For example, if a given term takes on different definitions over time, as reflected in evolving legislation and organizational processes, these differences can be represented by contexts, each corresponding to a given time period or epoch.) Metapattern can therefore be seen as having two primary attributes, i.e., its intention for use in the modeling of context and its focus on conceptual analysis rather than design. As a paradigm, Metapattern also espouses a number of modeling principles, primarily having to do with the importance of context and relationships in conceptual models.

In concrete terms, Metapattern is a diagramming method, which like many other such methods (e.g., ER diagrams or UML class diagrams), consists of lines, arrows, and boxes, with various annotations. In [Wisse] a fairly rich set of graphical elements is employed, e.g., to distinguish contexts from attributes. In its more recent application to modeling eGovernment semantics, Metapattern diagrams primarily use two primitive graphic elements (or "graphemes"): boxes and arrows. Three boxes are connected by a single arrow into "triads" that constitute the basic graphical "phrases" of which all Metapatterns consist.[2]



[2] The terms "triad" and "phrase" are my own. They are used here to facilitate discussing Metapatterns but are not found in other literature on the subject.

This triad is interpreted as meaning that R is a characteristic of A within the context of B (or vice versa). Diagrammatically, Metapattern is therefore a very simple method, essentially built on binary relationships between conceptual (and sometimes concrete) entities. Binary relationships are logically sufficient to represent any degree of complexity, as can easily be seen in binary trees or the expressive power of Lisp lists (which are based solely on the "dotted pair" concept). Being based on binary relationships, Metapattern models can be thought of as utilizing an extreme form of database normalization. The combination of such binary relationships in Metapattern models can then be thought of as performing relational "joins" to form more complex relationships.

Because they rely on these simple, universal building blocks, Metapattern diagrams have a minimum of notational and graphical subtlety, which makes it easy to grasp simple examples. The inevitable trade-off, of course, is that more complex examples result in complex diagrams that can require large numbers of binary relationships in order to express realistic problem domains.

This trade-off between what we might call "grapheme-level" simplicity (i.e., the number and complexity of primitive graphical elements used in a diagramming method) and the expressiveness of the method is inescapable. Diagrams having low grapheme-level complexity can achieve high expressivity only by allowing their graphemes to have highly general (and therefore relatively unspecified) meaning and by using large numbers of them. This trade-off is present in all diagramming methods, and each must attempt to find its own "sweet spot" that balances its primitive-level complexity against its diagram-level complexity. Because complexity in many problem domains is irreducible beyond some point, any diagramming formalism that captures such a domain must exhibit a high degree of complexity, one way or another. Whether this complexity resides at the primitive level or the diagram level is largely a matter of taste on the part of the formalism's designer and users.

Despite its apparent simplicity, there are a number of subtleties in Metapattern diagrams, which make them more complex than they may first appear. First, the vertical positioning of the graphical sub-elements of a Metapattern triad carries meaning: boxes that appear higher in the diagram are more general than those that appear lower. Second, the arrow connecting the three boxes of a triad represents the fact that the upper two boxes play distinct "roles" with respect to the lower box; although the direction of the arrow is unimportant, it is a necessary feature of the diagram, since it allows distinguishing these roles. Nevertheless, the arrow may be confusing to novice users of the paradigm. Third, a given box may participate in any number of binary relationships, which is of course necessary to allow more complex relationships to be built up from binary ones. Graphically, this is represented by multiple arrows connected to the bottom of the bottom box of a triad. In this usage, the bottom box of a triad represents the entire relationship represented by that triad. Fourth, a second style of box (typically distinguished by being horizontally rather than vertically oriented and having a less bold outline than normal triad boxes) may be used to represent types of entities, concrete instances of entities, or attributes of such entities; staggered, overlapping boxes of this style may be used to represent distinct aspects of such entities. Finally, the top of a Metapattern diagram is always bounded by a bold line that is referred to as the "horizon" of the model; this added grapheme represents the scope of a given diagram, but again, its full meaning may not always be immediately obvious to a novice user.

Metapattern diagrams are still noticeably simpler than many other graphical techniques, which often use multiple types of lines and arrow heads, junction boxes, subdivisions within entity boxes, cardinality annotations, etc. to enhance their expressivity, at the cost of greater visual complexity. However, in most cases, such added complexity is used to enable these other methods to serve as ICT system design tools, in addition to conceptual analysis tools. Metapattern achieves its relative simplicity by limiting its scope to analysis, which can be seen as an advantage or a disadvantage, depending on the user's needs and expectations. The argument in favor of Metapattern's stance along this axis is that conceptual analysis is often useful even without—or prior to—designing a computer system and is simpler and more agile when it is unburdened by design considerations.

The remainder of this paper focuses on the conceptual and representational aspects of Metapatterns rather than their diagrammatic properties.

4. Where does Metapattern fit?

The most salient aspect of Metapattern is that it is primarily an analysis tool, not a design tool. That is, despite the fact that a Metapattern model may represent types of entities or attributes and even concrete entities and attributes, it is not primarily intended (or particularly well suited) to represent data models or knowledge structures to be used in designing ICT systems. Instead, Metapatterns are intended to help analyze a problem domain by showing the conceptual relationships among some subset of the contexts that are present in or relevant to that domain and which are of interest to the analyst. Although it would be possible to use Metapatterns to help design ICT systems, other tools provide a richer set of mechanisms for this purpose.

By avoiding issues having to do with concrete representation and the design of data or knowledge structures, Metapattern eliminates considerable complexity and saves the analyst the effort of worrying prematurely about design and implementation issues, such as normalization, memory efficiency, access mechanisms, performance, etc. This has the potential to facilitate analysis by unburdening it from irrelevant details. It is much easier to explore alternative conceptual models when doing so does not require revising myriad design and implementation choices that are tied to a previous model. However, it is often the case that design and implementation considerations impact conceptual modeling choices and even reveal added conceptual complexity that is not apparent prior to designing and implementing a system based on a given conceptual model. The traditional "waterfall" approach to analysis, design, and implementation—which assumes that each of these processes can be completed before the next is begun—has well-known flaws [Royce], resulting from the fact that conceptual understanding of a problem domain is often incomplete prior to attempting to build an ICT system that deals with that domain. Furthermore, the later in the overall process a conceptual shortcoming is discovered, the more costly it is to address it, since many intermediate steps must be “unwound” and redone in light of the new insight.

Nevertheless, Metapattern’s focus on conceptual modeling and analysis is important and useful. It is vital to perform these steps early in the process of addressing any problem, to ensure that the problem and its domain are understood as well as they can be before proceeding. Performing such analysis early can often avoid the danger of designing and implementing an inappropriate or infeasible solution or attempting to solve an inappropriate or infeasible problem. Furthermore, the waterfall paradigm itself was developed as a way of combating the dangerous tendency to leap into designing and implementing a solution before spending adequate time analyzing and understanding the problem domain. Abandonment of this sequential approach runs the risk of throwing out the baby with the bathwater.

However, assuming that analysis concludes that a solution to a given problem is appropriate and conceptually feasible, it is important to proceed to the design phase (and ultimately the implementation phase—if only by means of prototyping) without spending inordinate effort on analysis, in order to allow feedback “upstream” from these later phases to trigger and inform reconceptualization, as necessary. In the context of Metapattern, this implies the need for relatively streamlined (i.e., automated) mechanisms that can transform Metapattern models into first-cut designs for concrete representations suitable for use in ICT systems. Without such streamlined mechanisms, it would be prohibitively expensive to revise a Metapattern model in light of new downstream insights and then generate a new design and implementation to be tried again, in a cyclic or “spiral” fashion [Boehm].

In this regard, most other data modeling and AI knowledge representation mechanisms are more concrete than Metapattern. They tend to straddle the boundary between conceptual modeling and design: that is, they typically include some conceptualization features but also directly facilitate concrete design. Many formal programming methods developed in AI embody approaches based on program refinement or program transformation, e.g., [Balzer], which transforms an abstract model of ICT system requirements into a sequence of increasingly concrete designs and implementations. Alternative AI techniques rely on executing formal specifications directly, rather than transforming them first, e.g., [Zave]. These program refinement and executable specification approaches tend to use formalisms that are either based on formal logic or are a form of pseudo-code. As such, they lack the visual clarity of a purely diagrammatic technique like Metapattern, but they can be more expressive and specific. Finally, model-driven architecture (MDA) approaches [Kleppe], derived in part from early automatic programming research in AI, often use conceptual models to drive the design process, modifying these models rather than the designs themselves in order to incorporate feedback from early prototype use.

Most data modeling methods also provide rich facilities for describing entities and their attributes. Though many (such as ER) also describe arbitrary relationships among these entities, most recent methods tend to have an object-oriented flavor, which distinguishes the “is-a” (class/subclass) relationship and the “part-whole” relationship, both of which are oriented toward the modeling of entities.

In contrast, Metapattern's focus is on the modeling of context. This focus is not unique: many AI modeling methods have long recognized the importance of context, particularly when dealing with natural language, terminology, and ontology, as embodied in the work of Schank and Sowa and the representational systems KL-ONE, LOOM, Cyc, TopicMaps, RDF and OWL, etc. [Schank, Sowa, Brachman, MacGregor, Lenat, Maicher, W3C]. However, by focusing on context as its primary concern, the Metapattern approach elevates context to a distinguished (and well-deserved) level of prominence.

Despite its focus on context, Metapattern does offer some insights into entity—and even attribute—design issues, by suggesting which attributes should go with each level of an entity in a set of overlapping contexts. For example, a Metapattern model of a "person" may consist of a number of sub-contexts, such as Person (abstract), Member of an organization, Legal entity, Occupant (of a home), Resident (of a city or country), etc. These contexts in turn imply what specific information belongs at each of these levels, for example, employee information at the "Member of an organization" level, address information at the "Occupant" level, and citizenship information at the "Resident" level. Therefore, even though Metapattern models are not design instruments, per se, they may still aid the design process in various ways.

In this regard, however, it is worth noting that Metapattern's relative lack of concern for ICT issues extends to its lack of computer-based tools for constructing or interpreting its own models. That is, it is essentially a paper-based modeling mechanism.[3] Although other diagrammatic modeling approaches, such as UML class diagrams and many AI-based modeling formalisms, can also be drawn on paper, sophisticated computer-based tools have been developed to help users construct their diagrams, and moreover, to interpret them automatically once constructed. Automatic interpretation of this sort enables diagrammatic models to drive subsequent design efforts as well as performing various kinds of analysis on the models. This is done by automatically constructing a logical representation of the entities and relationships present in the diagram (typically captured as the diagram is constructed with a computer-based tool) and then performing automated analysis on this logical representation. In many cases, the logical representation can itself be edited or manipulated to produce a modified diagram, thereby allowing the user to work from the diagram or its logical representation, whichever is more appropriate for a given purpose. Since the Metapattern formalism is intended for analysis rather than design, it does not provide such tools. While this is not a significant drawback during analysis, it does mean that additional work would be required to derive design or implementation details from a Metapattern diagram.

Metapattern models of context are relatively simple to read and understand, especially if they are annotated and explained as in some of the most recent GBO publications utilizing Metapatterns [Waters]. They are more difficult to construct, but that is of course true for nearly any modeling paradigm. In addition, there is no single "canonical" Metapattern model for a given set of contextual relationships, which again is true for nearly any modeling approach. This lack of a unique canonical form for a given contextual domain makes the process of developing

[3] Of course, this can be done on a computer, using general-purpose drawing tools, but the result would simply be a drawing, having no special semantics accessible to the computer.

Metapattern models something of an art, where the value of the result will depend on how well the model is oriented toward the analytic task at hand. This again is inherent in any modeling endeavor.

Once constructed, a well-designed Metapattern model can reveal many insights about the domain in question. In many cases, these insights include strong hints at what might be appropriate design choices for an ICT system to solve specific problems in the domain. A Metapattern model can therefore be used as a basis for system design, which could be performed "by hand" or, as suggested above, using automated or semi-automated tools. In its current form, Metapattern does not include such tools, though Wisse, et al., have prototyped some prototype tools of this sort, thereby demonstrating their feasibility.

Another important area that straddles analysis and design is the development of methods for resolving semantic differences across contexts. The use of Metapatterns to analyze contextual relationships among interacting government agencies has revealed that such agencies often use the same terms in rather different ways. Such variations in meaning across contexts represent real, underlying differences in the tasks performed in those contexts. For example, tax assessment and collection may define individuals and their relationships to each other (such as dependency) quite differently from the ways these are defined for benefits or issues involving legal custody. These differences are typically defined in legislation, regulations, and policies, but they are even more fundamental than that—inhering in the purpose and nature of the tasks being performed. Although it may be tempting to imagine that these semantic differences can be reconciled by mutual agreement and compromise, this denies the reality that the differences are often irreducible and significant. Such variations in meaning are important and desirable, and they cannot be ignored or wished away.

The fundamental problem of semantic interoperability (whether embodied in eGovernment systems or as performed traditionally by human interaction) is how to resolve differences in meaning across distinct contexts while understanding and respecting those differences. Here "resolving" does not in general mean simply finding the common ground between different meanings (i.e., their intersection), since that intersection may be empty—or may not be useful. Instead, resolving differences may mean expanding the context of each meaning until related contexts are found for each one—and using these overlapping, expanded contexts to connect the two meanings.

This desire to resolve semantic differences implies the need to develop techniques to combine, translate, and mediate variant semantics across multiple contexts. Such techniques should be developed prior to designing ICT systems intended to perform specific tasks within a given domain. Metapattern models should prove to be a useful input to this effort. For one thing, insights derived from contextual analysis of a domain may suggest techniques specific to that domain. Moreover, analyzing multiple Metapattern models of multiple domains may suggest general strategies for resolving semantic differences, which could be applied to any domain.

5. Conclusions

Although Metapattern is not primarily a tool for designing ICT systems, its contextual models can offer many useful inputs to the design process, such as indicating appropriate ways of partitioning entity attributes according to the set of contexts in which an entity is defined. Furthermore, it can play a vital role in analyzing and understanding contextual sources of semantic variation across organizations that need to interoperate meaningfully, for example to provide eGovernment services. Finally, the use of Metapatterns to analyze the contextual structures of domains may suggest techniques for resolving semantic differences across those contexts.

Metapattern's focus on context is by no means unique. However, its avoidance of most other concerns that are extraneous to the analysis of context gives it a somewhat unusual ability to create relatively simple, understandable contextual models that are unburdened by the need to represent class-subclass or part-whole relationships, detailed entity attributes, cardinality, normalization, design constraints, data structure choices, and myriad other factors.

In this respect, Metapattern deserves serious consideration as a tool for analyzing contextual relationships and the semantic variation that emerges from such relationships. This role promises to be crucial in meeting one of the central challenges of our age, namely the integration of heretofore separate, loosely connected processes, organizations, and information systems to provide more consistent, efficient, convenient, and higher quality services that meet the goals of large-scale enterprises, such as multi-national corporations and eGovernment.

References

Balzer Robert, Neil Goldman, David Wile. *"On the Transformational Implementation approach to programming," Proceedings of the 2nd international conference on Software engineering*, San Francisco, California, pp. 337 – 344, 1976.

Boehm, B. W., "A Spiral Model of Software Development and Enhancement," *Computer*, Vol. **21**, No. 5, May 1988, pp. 61-72.

R.J. Brachman and J. Schmolze, "An Overview of the KL-ONE Knowledge Representation System", Cognitive Sci 9(2), 1985.

Anneke Kleppe. *MDA Explained, The Model Driven Architecture: Practice and Promise,* Addison-Wesley, (2003), ISBN 0-321-19442-X.

Douglas Lenat. "Hal's Legacy: 2001's Computer as Dream and Reality. From 2001 to 2001: Common Sense and the Mind of HAL". Cycorp, Inc. http://www.cyc.com/cyc/technology/halslegacy.html.

Robert M. MacGregor. "Using a Description Classifier to Enhance Deductive Inference" in *Proceedings Seventh IEEE Conference on AI Applications*, pp. 141-147, 1991.

Lutz Maicher and Jack Park. *Charting the Topic Maps Research and Applications Landscape*, Springer, ISBN 3-540-32527-1.

Royce, W. W. "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings of the Western Electronic Show & Convention*, August 1970, Session A/1, pp. 1-9.

F. C. Shank. *Conceptual Information Processing*, North-Holland, 1975, ISBN 0-7204-2507-7.

John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Brooks Cole Publishing Co., Pacific Grove, CA, 1999.

W3C: http://www.w3.org/TR/owl-ref.

Waters, Peter. *Towards authentication and authorisation across organisations and domains: An experiment in thought*, Draft, 27 January 2010.

Wisse, Pieter. *Metapattern: context and time in information models*, Addison-Wesley, 2001, ISBN:0-201-70457-9.

Zave Pamela and Raymond T. Yeh. "Executable requirements for embedded systems," Department of Computer Science. University of Maryland, *Proceedings of the 5th international conference on Software engineering*, San Diego, California, pp. 295 – 304, 1981, ISBN:0-89791-146-6.