Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2006), *20*, 89–94. Printed in the USA.

DOI: 10.1017/S0890060406060070

RESPONSE TO KEYNOTE

Yes, and by the way . . . thoughts on "Whither design space?"

ULRICH FLEMMING

School of Architecture, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA

(Received July 4, 2005; Accepted December 9, 2005)

Abstract

Having no fundamental difficulties with Woodbury and Burrow's article, I explore some implications of their work based on my experience, and that of PhD students advised by me, with developing design support systems. I suggest that such systems need a distinct *task layer* on top of the *computational layer*, where the power of a system rests. I also express unease with some details of the article under review, in the hope of contributing to a wider discussion.

Keywords: Design Space; Design Support System; Knowledge Level; Task Environment

1. INTRODUCTION

Over the last decade, and strictly from the sidelines, I have had several occasions to catch glimpses of Woodbury and Burrow's work. Each time, I have been impressed by their determination, their clarity of purpose, and the formal rigor with which they were proceeding. The same can be said about Woodbury and Burrow's Keynote: its authors are well aware of the implications of their work and know what still has to be done before success can be declared (end of their section 4). I also very much share the intuition on which their work is based, namely, a belief in the potential of computers to support design exploration. As Woodbury and Burrow state, this capability is not offered at all by commercial computer-aided design (CAD) software, and therefore an obvious candidate for academic research. That is not to say that I do not have quibbles here and there, which should come as no surprise, given the scope of Woodbury and Burrow's enterprise. However, I would be perfectly content with discussing these issues over a beer next time we meet, and as far as Woodbury and Burrow's paper is concerned, make some encouraging noises and go back to work in the garden.

Reprint requests to: Ulrich Flemming, 1254 Middletown Avenue, Northford, CT 06472, USA. E-mail: ujf@cmu.edu

However, I am on record with my opinion that our field needs lively discussions of the issues we face more than anything else (Flemming, 2004), and it is in this spirit that I will pick up in the following a few of the quibbles I have. With this, I wish to contribute to the discussion that I hope will be triggered by this Special Issue (kudos to the editors for coming up with the idea) and, in the best case, to provide comments that Woodbury and Burrow themselves may find interesting with respect to the work ahead of them.

2. WHITHER THE KNOWLEDGE LEVEL?

The first issue I would like to pick up is the one raised by Woodbury and Burrow's comments in Section 2.1 about the knowledge level in general and about researchers in the SEED (Software to Support the Early Phases of Building Design) project who, they claim, early on confused knowledge-level concepts like functional units, design units, and technologies with the underlying computations that could support them. This is not the first time I have heard this remark, and I have always been uncomfortable with it. This unease betrayed, I am sure, an instinctive reaction of self-defense on my part because I am as responsible as anybody else for introducing these concepts into the SEED project. However, I also think that my reaction had a more rational base because I knew from my own work with SEED-Layout, a component of SEED (Flemming & Chien, 1995),

how useful these concepts were *at the symbol level*, that is, when they were explicitly represented inside the program. Woodbury and Burrow's paper finally motivated me to get at the bottom of this, and I went back to the original address to the American Association for Artificial Intelligence in which Allen Newell introduced the notion of a knowledge level of a system distinct from and above the symbol level (Newell, 1981). After rereading the speech, I am convinced that my unease was justified. Moreover, I now believe that Woodbury and Burrow's eagerness not to confuse the knowledge level with the symbol level may turn into a red herring, getting in the way of the work they plan to do in the future.

Let me explain. Newell (1981) introduces the notion of a *knowledge level* in a very specific context, that of artificial intelligence agents pursuing goals (like winning a game of chess) by using knowledge about the task environment at hand, where this knowledge is represented in some form inside the program that physically realizes the agent, which Newell calls the *symbol level*. Newell claims it is useful to talk about this knowledge as such, that is, independently of how it is represented, and to be able to do this, he says, we need a more abstract level at which we can describe what this knowledge is in the first place. He gives a general definition of knowledge at the knowledge level in strictly functional terms, namely those of goal-directed rationality.

Now, a system like SEED-Layout, or more generally, the "mixed-initiative" design support systems envisioned by Woodbury and Burrow, do not pursue goals on their own, rationally or otherwise. Such a system is meant to support designers in pursuing whatever goals *they* have, and it does not have to have knowledge about these goals. As a result, Newell's concept of a knowledge level does not extend, in a strict sense, to a mixed-initiative system (although it may be applicable to parts of it, like that portion of SEED-Layout that sizes and resizes, after the action of a designer, the spaces in a layout subject to explicitly represented constraints). Of course, the notion of a knowledge level does not arise in connection with the typed feature structure spaces Woodbury and Burrow developed, which comes as no surprise, given their intent not to confuse knowledge and symbol levels.

However, the flip side is that their space is also *not a design space*, unless Woodbury and Burrow have discovered specializations of typed feature structures that make the resulting space uniquely applicable to design and design only. I have seen no indications of this and must assume that their space could support any domain that could benefit from the capabilities they provide. There's the rub: if Woodbury and Burrow want to develop "effective, medium-scale demonstrations" and later on "serious industrial applications," they must give designers something that is manifestly a design space and allows designers to interact with constructs with which they are familiar from their daily practice or which, at least, make sense to them, given that background. These constructs are likely to include design briefs or architectural program specifications, spatial or physical building components, constraints on the shape and placement of these, and so forth, all of which are semantically very distinct from each other. I cannot imagine that a designer could handle all of these by acting directly on syntactically uniform typed feature structures.

How can a typed feature structure space be turned into a design space in that sense? I see only one possibility: by adding to the program above the typed feature structures a layer that represents aspects of the task environment understandable to designers. For the sake of brevity, I call this layer the *task layer* in the following. Note that this layer is a genuine and necessary part of the symbol level: it is *not* part of a more abstract knowledge level (although it may be interesting to talk about the knowledge embedded in these symbolic representations at that higher level). As I will show below, the task layer is also not identical, and should not be confused with the user interface.

Let me explain this notion of a symbol level consisting in itself of a task layer and a *computational layer* below it, so called because it is the place where the essential computations happen. The software a bank uses to manage client accounts needs symbolic structures that are able to capture the salient attributes of such accounts (client name, address, current balance, etc.). It may be able to execute all of its operations directly on structures representing these attributes; that is, in this type of application, there is no need for a computational layer below the task layer.

A different case is presented by RaBBiT, a program that aims to support architectural programmers independently of the terminology and methods they prefer to use (Erhan, 2003; Erhan & Flemming, 2004). RaBBiT is able to do this because its internal representation and the operations it performs on it generalize and unify the various programming methods Erhan encountered in practice or in the literature. He found that all approaches essentially entail stepwise information refinement that can be entirely realized through what he calls *constructs* and their attributes, with the addition of dependencies showing how constructs can be refined into other constructs. Users of RaBBiT can interact directly with these objects or, if they wish, superimpose a classification hierarchy over them that captures the terminology they, or the firm they work for, use for programming tasks. In other words, RaBBiT "raw" operates entirely at the symbol level, but it also allows users to create a task layer (in minimal form) if they wish. This is possible because the relation between constructs and classified objects is strictly one to one, and can be implemented simply by adding a classification label to constructs that may or may not have a value.

In more complex cases, the relation between the task and computational layers is more intricate and must be managed by the (computer) programmer. A case in point is SEED-Layout, which offers designers the opportunity to define explicit constraints so that the program has something to work with when it tries to size the spaces in a layout. An example is adjacency constraints, which are among the most

https://doi.org/10.1017/S0890060406060070 Published online by Cambridge University Press

common constraints at work in layouts. To define such a constraint, a designer has to indicate which spaces are to be adjacent and specify the minimum length of their common boundary. Internally, the system translates such a specification into one equality and two inequalities in the corner coordinates of the spaces concerned, which, if they are satisfied simultaneously, guarantee the desired relation. The designer, of course, does not specify directly these algebraic expressions. SEED-Layout provides an interface for the interactive specification of adjacency (and other) constraints, and this information is captured internally in symbol-level structures like an adjacency constraint class that is instantiated whenever a designer creates such a constraint and captures the parameters characteristic for this instance. Note that this construct is more permanent than the dialog boxes used to elicit or edit the constraint parameters, which are created and discarded on the fly as needed and never survive the end of a program session. However, adjacency constraints have to be available during an entire program session because they may be applied at different times to alternative layouts containing the same spaces; the constraints also may have to be saved persistently for reuse across design sessions. In other words, the adjaceny constraint, and the functional units collecting all constraints applying to a space, are a genuine and very useful part of SEED-Layout's symbol level, albeit one that manifestly represents directly an aspect of the task environment; and because they are used by SEED-Layout in a seemingly goal-directed fashion, it may even be instructive to talk about them at the knowledge level in Newell's sense, for instance, in a user manual or tutorial.

Before I indicate some of the implications of the above for Woodbury and Burrow's future work, I would like to make a proposal for resolving (as the third leg in the classic dialectical triad thesis, antithesis, and synthesis) our seeming disagreements about the knowledge level. Let us agree to the following:

- design systems of any complexity typically need symbol-level representations of aspects of the task environment, which may form a program layer above the computational one at the symbol level. This task layer is distinct from and should not be confused with any more abstract knowledge level above the symbol level; and
- the need for a symbol-level representation of aspects of the task environment should not blind us to the fact that for the sake of generality, rigor, computational efficiency, and so forth, a design support system should perform its actual computations on a formal representation at a computational layer below the task layer. It is this layer that typically gives the system its computational power and "punch."

The latter point is important particularly for software developers using the use case approach, of which I am a fan (Flemming et al., 2004), or the Uniform Modeling Language, about which I am more ambivalent. The use case approach leads to a systematic discovery of the task-specific concepts that have to be represented internally by a piece of software and of promising ways for "actors" to interact with these concepts through the user interface. However, the approach does not provide any formal or informal tools that would help developers in discovering the need for an underlying more formal computational layer, let alone tools to develop that layer; the approach is totally silent on this issue.

Let me add an observation to this point. I find it striking that whenever researchers develop a formal computational layer to support some design tasks, the resulting representation and algorithms seem to be of interest and applicable also to tasks that are not design or at least not building design related. This is true for both RaBBiT's constructs and Woodbury and Burrow's typed feature structures. Conversely, researchers may find existing methods or software well suited for their task, as is the case with SEED-Layout, which uses established methods of constraint propagation. It appears that aspects unique to architectural design get stripped away when we move from the task to the computational level, which leads me to formulate, as an aside, the following conjecture: those aspects of architectural design that are unique to it and distinguish architecture, as an art able to provoke unique sensual, emotional, and intellectual responses, from other purposefully created artefacts are precisely those that do not lend themselves to computational treatment in the deeper sense, as opposed to the "lightweight" treatment they find in the user interface and at the task level, whose main purpose is to mediate between the user and the computational layer, not to execute complex computations.

3. CHALLENGES OF THE TASK LAYER

To create a task layer that would encourage designers to explore a design space is a challenge no less formidable than that posed by the development of an efficient and formally rigorous computational layer. My own experience (largely with SEED-Layout) suggests that it is, in a way, more arduous because we do not have the deductive reasoning of mathematics available to us when we want to evaluate the validity of certain decisions. When it comes to the task layer, we have to face the contingencies of design practice and the idiosyncrasies of individual designers, which we cannot predict at the outset and may discover only after the fact.

It all starts with terminology. Architectural design, as a discipline, has not developed, despite its long history, an agreed upon set of terms for conducting a discourse about its central concerns, even something as basic as the names of building components has not been generally codified. However, when designing the task layer for a design support system, developers have to commit to a basic termi-

https://doi.org/10.1017/S0890060406060070 Published online by Cambridge University Press

nology and face the danger that it will turn certain users off from the outset (in my past work as a member of various research teams, I have observed repeatedly how difficult it is even for people sympathetic to each other's goals to agree on a common set of terms). Should a collection of design requirements be called a "design brief," an "architectural program," a "problem statement," or a "requirements specification"? We do not know and will find out (if we get that far) only at some future time. Note that the choices we have may reflect different attitudes about the task at hand. If we select in the above example architectural program, we indicate that we want to appeal to design practitioners in general. However, if we select problem statement, we may be indicating to users that the activities supported by the system should be viewed as a form of problem solving.

This brings up a related issue: to which degree should the underlying computational layer be transparent to users.[1] For people like me, or Woodbury and Burrow I assume, who believe that the tools and media used in design have a profound impact on both its products and the processes used, there is the temptation to make novel representations and procedures transparent to users to some degree in the hope that we may observe changes in their way of doing design. For example, the operations by which users can add spaces to a layout in SEED-Layout are the production rules I developed for wall representation adapted to generate what I have called "loosely packed" arrangements of rectangles (Flemming, 1989). These operations are formally well understood, and are the basis, for example, for SEED-Layout's capabilities to size layouts correctly on its own (because it understands the underlying spatial structure and is able to take it into account to avoid overlap between areas without additional computation). The same operations are reflected in the user interface, which "tricks" users into defining implicitly the left-hand side of a production over the current layout, after which the generation of the new layout itself can be left completely to the system. As a consequence, SEED-Layout's user interface does not offer direct manipulation (plans I had in this regard were never realized). Anecdotal evidence suggests that this feature was perceived by novice users not familiar with my prior work as strange, if not directly off-putting: they were looking for the type of direct manipulation they were accustomed to from work with commercial CAD systems. I had hoped that careful training could overcome this hurdle and make users appreciate the advantages of the novel approach, but the project never reached a stage of user testing needed to validate this.

In Woodbury and Burrow's case, the design space could, and should, let designers create and manipulate anything that could conceivably be called a design state, which includes not only descriptions of physical aspects, but also design briefs, requirements, and constraints on the one hand and the results of design evaluations against these on the other hand: if you have a formal representation powerful enough to represent all of these, use it! However, to which degree should the uniformity of the underlying representation be transparent to users, for whom, for example, an architectural program for a building is fundamentally different from a building that satisfies it.

Chien describes the five orthogonal dimensions of the design space created and maintained by SEED-Layout along which users may trace the generation path of individual layouts: the derivation path of a specific layout, alternative layouts corresponding to the same problem formulation, the derivation path of a specific problem formulation, alternative problem formulations, and hierarchical part-of relations between layouts or problem formulations (Chien, 1998; Chien & Flemming, 2002). It is true that the concept of the SEED-Layout design space evolved over several years of working with the system, in distinction to Woodbury and Burrow, who start from this notion, which I view as true progress. It is also true that at any time, a user of SEED-Layout is "located" at and views only a single design state. However, this is strictly a feature of the user interface introduced to ease the already formidable cognitive burden imposed on the user. It is not really the case that the design space itself is considered strictly from this perspective. Chien provides views of larger portions of that space (as *space*, not as individual layouts), possibly simultaneously along several dimensions, and it would be trivial from a programming standpoint to allow the user to work on more than one state simultaneously. The reason for this is that SEED-Layout explicitly represents the relations between all objects in that space along all dimensions, however ad hoc the representation itself may be. I would suggest that Chien's work provides, at the very least, an initial sense of the potential complexity of design spaces that can be explored (see also session 3 in Flemming & Chien, 1999).

We posit the following questions to Woodbury and Burrow: what are distinctions between the typed feature structures populating their design space that are meaningful to users? How can these distinctions be captured at the task layer and made visible (transparent!) through the user interface? Which principles were used when decisions about these issues were made?

The last point is important because we may reasonably assume that even people as smart as Woodbury and Burrow do not get everything right off the bat. However, we learn from failure as much, if not more, as from success, and a clear record of the principles at work would allow us to understand better what we did, in fact, learn from a set of

[1] Here I use the term "transparent," as it is commonly used to mean "showing through" (Latin *trans* for "through" and *parere* for "show"), "easily detected," or "letting something show through," "diaphanous" as in "the motives of my opponent are perfectly transparent." I am *not* using the term in the database sense, where it means the opposite, namely, "hidden" or "invisible." It took me some time to figure this out, and I have never forgiven the database people.

experiments. To close the loop: an elucidation of principles may turn out to be a revealing ingredient of a knowledge-level description of the system, if we stretch Newell's notion a bit.

4. THE STRUCTURE OF SHAPE RULES

Woodbury and Burrow describe the differences between shape grammars and their representation as one between operations and structure. However, if one has a closer look at the shape grammar formalism itself, this distinction almost disappears. In the typical formulation of a rule in terms of left- and right-hand sides α and β, respectively,

$$\alpha \rightarrow \beta,$$

in which the arrow implies the following operation:

$$(\gamma - \alpha) \cup \beta,$$

where γ is the current state (of a design, say), from which the left-hand side is subtracted (by computing the difference between the two), and the result combined with the right-hand side by shape union. (I am omitting the role of parameter assignment and transformation for the sake of simplicity.)

A general-purpose shape grammar interpreter implementing this notion of rule application directly would provide, as general features, a difference operator and a union operator, each guaranteed to work for any pair of well-formed shapes based on some common syntax (if we abstract, again for simplicity, from the possibility of pathological situations). To define a shape rule in such an interpreter, one would have to specify only the left- and right-hand sides, that is, the definition could be done *entirely in terms of structure*, not of operations. Except for parameterization, the syntax used to represent α and β would be the same as the one used to represent γ. A test to establish that α and β are, in fact, treated syntactically the same by the interpreter would be its ability to execute a rule backwards without further specifications on the part of the rule writer.

Such a shape grammar interpreter would be hard to build indeed, and Woodbury and Burrow are correct when it comes to shape grammar implementations using a standard production system shell.[2] In this case, the left-hand side typically consists of a conjunction of predicates asserting the existence of certain objects with certain attributes (which can roughly be viewed as a declarative specification as implied by the formalism), while the right-hand side is a procedure implementing the action implied by the rule, that is, it is described clearly as an operation and syntactically very different from the description of the left-hand side.

[2]The situation may be different for interpreters using a logic-programming language like PROLOG, an issue I cannot pursue here.

However, the fact remains that shape rules can be viewed structurally in line with Woodbury and Burrow's expressed interest. I think this offers some promising avenues to pursue for those portions of the task layer and user interface that offer designers some form of customization when they try to explore the potential of particular formal ideas in the context of a specific project or to codify the way in which certain design tasks should be handled across projects. One of the great advantages of shape grammars I have found in my own work with them (aside from the fact that they are able to handle geometry explicitly and directly) is the fact that they look natural to designers (after some initiation, to be sure) who view a design (practically or conceptually) as the result of a series of transformations, and who understand that the logic of these transformations can be expressed succinctly in terms of shape rules.

Woodbury and Burrow state that shape grammars could be written on top of their representation, and I would indeed like to see them do this. On the one hand, I would like to know how difficult this is or, conversely, if there are structural similarities between the two representations that can be computationally exploited. On the other hand, I would like to see if something akin to a shape grammar could help bring the prima facie strange world of typed feature structures one step closer to being accepted, if not embraced, in design practice.

REFERENCES

Chien, S.-F. (1998). *Supporting information navigation in generative design systems*. PhD Dissertation. Carnegie Mellon University, School of Architecture.

Chien, S.-F., & Flemming, U. (2002). Design space navigation in generative design systems. *Automation in Construction 11(1)*, 1–22.

Erhan, H.I. (2003). *Interactive support for modeling and generating building design requirements*. PhD Dissertation. Carnegie Mellon University, School of Architecture.

Erhan, H., & Flemming, U. (2004). Interactive support for modeling and generating building design requirements. In *Generative CAD Systems* (Akın, Ö., Krishnamurti, R., & Lam, K.-P., Eds.), pp. 121–144. Pittsburgh, PA: Carnegie Mellon University, School of Architecture.

Flemming, U. (1989). More on the representation and generation of loosely-packed arrangements of rectangles. *Environment and Planning B: Planning and Design 16(3)*, 327–359.

Flemming, U. (2004). Computer-aided architectural design: looking back, looking forward. In *Generative CAD Systems* (Akın, Ö., Krishnamurti, R., & Lam, K.-P., Eds.), pp. 17–30. Pittsburgh, PA: Carnegie Mellon University, School of Architecture.

Flemming, U., & Chien, S.-F. (1995). Schematic layout design in SEED environment. *Journal of Architectural Engineering*, *1(4)*, 162–169.

Flemming, U., & Chien, S.-F. (1999). *SEED-Layout Tutorial*. Pittsburgh, PA: Carnegie Mellon University, School of Architecture. Accessed at www.andrew.cmu.edu/user/ujf/download_files.

Flemming, U., Erhan, H., & Özkaya, I. (2004). Object-oriented application development in CAD. A graduate course. *Automation in Construction 13(1)*, 147–158.

Newell, A. (1981). *The Knowledge Level*. Research Report CMU-CS-81-131. Pittsburgh, PA: Carnegie Mellon University, Department of Computer Science.

Ulrich Flemming earned his first professional degree in architecture (Dipl.-Ing.) from the Technical University of

https://doi.org/10.1017/S0890060406060070 Published online by Cambridge University Press

Berlin in 1968, his postprofessional Master's in Architecture from MIT in 1972, and his PhD from the Technical University of Berlin in 1977. He worked in architectural offices in Germany and the United States from 1963 to 1970. Dr. Flemming was the Wissenschaflicher Assistant at the Department of Architecture, Technical University Berlin from 1973 to 1977, a faculty member in the Department of Architecture, SUNY Buffalo from 1978 to 1981, and a faculty member in the School of Architecture, Carnegie Mellon University from 1981 to 2002. His research is in the area of computer-aided (architectural) design, with focus on formal grammars, space planning, knowledge-based and case-based design systems, design databases, human–computer interaction in design, and computer-supported concurrent design.

https://doi.org/10.1017/S0890060406060070 Published online by Cambridge University Press