

Reprinted from  
Technology, Instruction, Cognition & Learning  
(TICL), 2003, 1, 1, 7-58.

**Domain Specific Structural Analysis for Intelligent Tutoring Systems:  
Automatable Representation of Declarative, Procedural and Model-Based Knowledge with Relationships  
to Software Engineering**

Joseph M. Scandura<sup>1</sup>

**Abstract**

ITS development has been idiosyncratic largely due to an inability to represent expert knowledge (content) as internally consistent hierarchies. This article shows how to construct consistent behavioral (specification) and design (knowledge) hierarchies that are correct with respect to their specifications. Any given concept (or operation) can be represented hierarchically in a manner that insures each level of representation is behaviorally consistent with every other level. Each level is represented explicitly in terms of exactly one of three kinds of refinement: component, category and operation. The introduction of abstract (equivalence classes of) values provides necessary precision while avoiding intractable mathematical problems. Combinations of these refinement types are adequate for representing procedural as well as declarative knowledge and both domain specific and domain independent knowledge. Interacting systems (models), for example, are shown to involve both declarative and procedural knowledge. In short, the process of structural analysis (SA) is postulated to provide a sufficient basis for reducing any given concept or process, no matter how complex, to arbitrarily simple elements lending themselves to automation.

**Introduction**

This article is the second in a planned series on the Structural Learning Theory (SLT), new developments in software engineering and its application in building and delivering intelligent tutoring systems (ITS). The first article summarized the current status of SLT with emphasis on recent developments having direct relevance to ITS. This article appeared in *Instructional Science* (Scandura, 2001a) and in a Special Monograph on structural learning, the latter of which includes two appendices, one providing an historical overview of development and application of SLT and the second a perspective on relationships to logic programming. The third (planned) article will detail an Intelligent Tutor Authoring System (ITAS) based on SLT, including the AutoBuilder development system and a General Purpose Intelligent Tutor (GPIT) capable of delivering instruction without change on any given content.

This second article fills a critical gap, both in extending and adding needed rigor in Knowledge Representation (KR). This rigor is an essential prerequisite to the third article.

**Background**

The enormous potential of intelligent tutoring systems (ITS) has been recognized for decades (e.g., Anderson, 1988; Scandura, 1987; Sleeman & Brown, 1982; Wulfeck & Scandura, 1977). Explicit attention to knowledge representation, associated learning theories and decision making logic makes it

---

<sup>1</sup> The author wishes to thank Thomas Honigmann, Hasan Sayani, Peter Brusilovsky, Valerie Shute and Alice Scandura for their helpful comments on this article from the perspectives of mathematician, software engineer, ITS and instructional design. Clearly remaining errors are the author's sole responsibility.

possible to automate interactions between the learner, tutor and content to be acquired (commonly referred to as the expert module). In principle, ITSs may mimic or even exceed the behavior of the best teachers. This potential, however, has never been fully realized in practice. As Psotka, Massey & Mutter (1988) put it in introducing lessons they learned from early ITS research “although advances in the field are dramatic and far reaching, our vision still far exceeds our grasp” ( p.1).

The situation today remains much as it has (been). The central importance of the content domain, modeling the student and the interactions between them remain as before (e.g., duBoulay, 2000). A major bottleneck in the process has been representation of the content domain (expert model). Knowledge Representations (KR) used in ITS have been based on a variety of theories. Among the earliest have been production systems, which play a major role in Newell & Simon’s early research on problem solving (e.g., 1972) and Anderson’s ACT-R theory (e.g., 1988, 1990, 1993, 1995, 1998), and (SLT) rules and higher order rules, which play a central role in Scandura’s Structural Learning Theory (e.g., 1971, 1973, 1977; Scandura & Scandura, 1987). Other approaches that are becoming popular among ITS developers include: Falmagne et al's (1990) knowledge spaces and their extension by Düntch and Gediga (1995, 1998), Sowa's Conceptual Graphs (1984, 1999), Formal Concept Analysis (e.g., Wille, 1992, Ganter & Wille, 1999; Stume, 1998) and Ontologies (e.g., Gruber, 1995; Guarino, 1998).

The process of creating such knowledge representations is commonly known as “knowledge engineering” in the ITS community and “task analysis” in instructional design, both of which are relatively informal in nature and go back many years. Knowledge engineering, for example, typically involves assisting domain experts to formalize what they know (e.g., Anderson, 1988). Task analysis (e.g., Gagne, 1965) traditionally begins by asking what one needs to do in order to achieve some predetermined educational goal(s). Although space precludes discussion here<sup>2</sup>, it is worth noting that most approaches to KR make a sharp distinction in the way they treat declarative and procedural knowledge, on the one hand, and domain specific and domain independent knowledge, on the other.

Given the time consuming nature and complexity of current approaches, a variety of methods and tools have been proposed to assist in the process. Structural (cognitive task) analysis, for example, moved earlier work on task analysis forward by introducing the notion of higher order (domain independent) knowledge and detailing a systematic process for specifying what the learner needs to know to perform desired behavior (e.g., Scandura, Durnin & Wulfeck, 1974; Scandura, 1977, 1984a). Taking a different tact, Shute & Torreano (in press) developed an automated knowledge elicitation tool to assist in the process of knowledge engineering. Such methods and tools facilitate the process. However, the problem remains that there are no universally applicable methods for representing content (expert knowledge) as internally consistent hierarchies. As Psotka et al (1988, p. 279) put it, “The fundamental problem is organizing knowledge into a clear hierarchy to take best advantage of the redundancies in any particular domain. Without extensive experience in doing this, we are largely ignorant of the kind of links and predicates to use. We are still far from cataloging the kinds of links needed for these semantic hierarchies. The best predicates that might describe these knowledge structures simply, beyond ISA and PART OF hierarchies, still need to be defined. Too little work has been aimed at developing new representations for information and relationships.” He goes on to mention the need to consider OO frameworks and/or structures containing objects, relationships and operations (pp.279-280).

In the absence of a generalizable solution to this problem, ITS development has been largely idiosyncratic. Tutor modules, and the way they interact with student models have been heavily dependent on the content in question. The widespread use of production systems (e.g., Anderson, 1983, 1988; Newell & Simon, 1972) for this purpose is a case in point. Production systems have the theoretical appeal of a simple uniform (condition-action) structure. This uniformity, however, means that all content

---

<sup>2</sup> Editor's Note: TICL invites thoughtful commentary for publication in subsequent issues.

domains have essentially the same structure – that of a simple list -- where even minor changes in ordering productions can have a major effect on behavior. In this context it is hard to imagine a general-purpose tutor that might work even reasonably (let alone equally well) with different content.

Consequently, ITSs have tended either to be developed de novo for each content domain, or built using authoring tools designed to reduce the effort required (e.g., by providing alternative prototypes, Warren, 2002). Whereas various industry standards (e.g., IMS, SCORM) are being developed to facilitate reuse of learning objects (e.g., diagnostic assessments and instructional units) in different learning environments, such standards are designed by committees to represent broad-based consensus. Cohesiveness (internal consistency) and simplicity are at best secondary goals. Specifically, such standards may allow but do not include a uniform way to represent content at arbitrary levels of abstraction in a behaviorally equivalent manner.<sup>3</sup>

A recent article on the Structural Learning Theory (SLT) proposes another approach to this problem (Scandura, 2001a,b).<sup>4</sup> SLT was designed from its inception explicitly to address interactions between the learner and some external agent (e.g., observer or tutor), with emphasis on the relativistic nature of knowledge (e.g., Scandura, 1971, 1988).<sup>5</sup> The former articles (i.e., Scandura 2001a,b)<sup>6</sup> summarize the rationale and update essential steps in carrying out a process called Structural (cognitive task) Analysis (SA). In addition to defining key elements in problem domains, and the behavior and knowledge associated with such domains, special attention is given to the hierarchical nature of SA, and its applicability to ill-defined as well as well-defined content.<sup>7</sup> Among other things, higher order (domain independent) knowledge is shown to play an essential role in ill-defined domains. SA also makes explicit provision for diagnostic testing and a clear distinction between novice, neophyte and expert knowledge. While broad, however, this overview omits some essential features (which at the time were pending patent approval, see below) and the precision necessary to allow unambiguous automation on a computer.

---

<sup>3</sup> The third planned article in this series shows how such hierarchies make it possible to build general-purpose tutors that can *without modification* intelligently guide highly adaptive testing (diagnosis) and instruction irrespective of the content in question – solely by reference to the KR in question.

<sup>4</sup> The SLT has evolved over a period of several decades beginning in the 1960's. Scandura (2001a,b) summarizes the current status and new developments in SLT. Appendix A in Scandura (2001b) provides a useful overview of major developments and related publications over the years. Some key characteristics of SLT are listed here for the reader's convenience: a) the central importance of structural (cognitive task) analysis, b) distinctions between lower and higher order knowledge (used to distinguish between domain specific and domain independent knowledge), c) the representation of knowledge at different levels of abstraction (used to distinguish between levels of expertise, make testing more efficient and/or to guide the learner and/or instruction), d) explicit processes for assessing what a learner does and does not know relative to a given body of content (i.e., the learner model), e) a universal control mechanism (playing a central role in problem solving, being implementable in a manner that is totally independent of higher as well as lower order knowledge), and f) assumed fundamental capacities such as processing capacity and processing speed. More directly related characteristics are detailed for the first time in this series.

<sup>5</sup> Originally, SLT was founded on basic research on cognition and problem solving (Scandura, 1971, 1973, 1977). Although SLT covers similar ground from a cognitive perspective, however, it differs in many detailed respects from cognitive theories widely used in ITS (e.g., Newell & Simon, 1972; Anderson, 1983). These theories, for example, differ with respect to the preferred mode of knowledge representation -- production systems vs. SLT rules (in addition to condition action pairs, the latter explicitly includes and makes specific provision for all basic software engineering constructs such as sequence, selection and iteration refinements). As in ITS research, the process of knowledge representation (a particular form of cognitive task analysis, called structural analysis) also plays a central role in SLT.

<sup>6</sup> SA was the first method to emphasize the difference between task analysis as applied to human behavior (e.g., Miller, 1956; Gagne, 1965) and the (e.g., cognitive) processes making such behavior possible (e.g., Scandura, 1970; 1977, 1982, 1984a, b; Scandura, Durnin & Wulfbeck, 1974).

<sup>7</sup> The process of structural analysis has a long history (e.g., Scandura et al, 1971; Scandura, Durnin & Wulfbeck, 1974; Scandura, 1982, 1984a, b). Most of the earlier work through the early 1980s concentrated on the identification of (domain independent) higher order knowledge as well as lower order (domain specific) knowledge. The use of ASTs to represent knowledge, however, came largely as a result of later work in software engineering (e.g., Scandura, 1991, 1994, 1997, 1999, 2001c).

Parallel research in software engineering provides the necessary rigor. This research makes very explicit what has until recently been an informal process (of SA). SA has evolved to the point where it is fully automatable on a computer (U.S. Patent 6,275,976, Scandura, 2001c) and sufficient for representing the knowledge not only associated with domain specific content but also with domain independent knowledge and ill-defined domains.

Associated methods represent a major step forward in software methodology. Programming is inherently a bottom-up process: Traditionally, the goal has been to find some way to represent data structures and processes (a/k/a to-be-learned content) in terms of some predetermined set of executable components. These components include functions and operations in procedural programming and objects in OO programming.

Instead of emphasizing the assembly of components to achieve desired ends, the emphasis in SA is on representing what must be learned (or executed in software engineering) at progressive levels of detail. Like structured analysis<sup>8</sup> and OO design in software engineering, SA is a top-down method. However, it is top-down with a big difference: Each level of representation is guaranteed to be behaviorally equivalent to all other levels. The realization of SA in AutoBuilder also supports complementary bottom-up automation. Not only does the process lend itself to automation, but it also guarantees that the identified competence is sufficient to produce the desired (i.e., specified) behavior.<sup>9</sup>

### **Objectives and Approach**

*Introduction.*-- As emphasized above, constructing a suitable knowledge representation (KR) is a crucial first step in building any ITS. Despite the development of systematic methods, such as structural analysis, and new tools, such as that by Shute & Torreano (in press), to help in the process, the quality of any KR depends primarily on informal judgments made by the person(s) doing the analysis. Methods and tools can help only to the extent that they encourage -- ideally require -- internal consistency in such KRs. This is not an easy thing to do!

Recent developments by the author and his associates in software engineering are believed to provide the theoretical foundation necessary to construct internally consistent KRs. Specifically, this article formalizes earlier work in structural analysis, detailing the rationale and theoretical constructs essential in creating internally consistent and correct hierarchical abstract syntax tree (AST) representations of software specifications and designs. More generally, it reveals a new approach to the problem of correctness in software engineering (U.S. Patent 6,275,976, Scandura, 2001c).

The method proposed herein also provides a way to represent knowledge in an internally consistent form ready for use in building ITS. This method is believed to be sufficient for representing the to-be-acquired behavior and underlying competence associated with essentially ANY content in a uniform and consistent manner. Although beyond the scope of this article, it also is believed that KRs obtained in this manner will provide a uniform foundation for building ITSs in a highly efficient manner. Specifically, in a third planned article in this series, we shall describe general-purpose intelligent tutoring (GPIT) systems that are guaranteed to work equally well on any domain specific knowledge -- as long as that knowledge is represented as described in this article. What, when and how information is presented to the learner is all handled automatically by the GPIT without change. Moreover, these GPITs have options allowing them to provide highly efficient and optimized adaptive instruction, progressive instruction from simple to complex

---

<sup>8</sup> Whereas processes and data are refined independently in structured analysis and in OO design, both are refined in parallel in structural analysis (SA).

<sup>9</sup> See Scandura, 2001a, b and [www.scandura.com](http://www.scandura.com) for updated information on a General Purpose Intelligent Tutor, which working in conjunction with content represented in this manner, can guarantee specified learning in a minimum time (e.g., with the fewest possible test and/or instructional interactions between tutor and learner).

or simple performance aids. There is no need for customization, irrespective of whether the content is procedural or declarative in nature, or whether it involves simple facts or complex interacting systems.

*Discussion.*-- We begin our development by asking the fundamental question of what it means to know something, and how to represent knowledge in a way that has behavioral relevance? Exactly what is observable behavior in the first place? A variety of answers have been proposed to these questions -- ranging from constructivism in cognition to object-oriented design in software engineering. Inevitably, these proposals have either been too vague and/or informal to readily lend themselves to automation (e.g., in software) or they have been highly technical, non-intuitive or otherwise essentially unavailable to non-specialists.

This article addresses the question of representation from a different perspective. It presents a new way to represent real world concepts that directly reflects the reality under consideration in a rigorous yet easily understood and operational manner. In contrast to English and other natural languages, we seek a representation adequate for those aspects of reality that lend themselves to automation by means of digital computer. Natural languages are at once extremely rich and general, and far too vague and imprecise to provide an adequate basis for unambiguous specification. They can be used to describe most if not all matters cognitive but they do so at the cost of ambiguity. We are not concerned here with literary flexibility -- e.g., the use of pronouns instead of intended nouns, or even prepositions, which add flavor to basic entities and/or relationships in the real world. On the other hand, we do not want to lose such meanings entirely. Rather, we seek to capture all essential flexibility, as long as it can be captured in a rigorous yet clear manner that lends itself to automation.

Specifically, U.S. Patent 6,275,976 (Scandura, 2001d) reveals a universally applicable representation that readily lends itself to automation. Any cognitive concept or idea can be represented precisely at any desired level of detail. There is an exact unambiguous relationship between different levels of representation, insuring that each level represents exactly the same idea or concept. This representation is meant to be completely general: It is equally applicable to research in cognition, instructional technology and software engineering (cf. Scandura, 2001a, b, c, d).

The representation also is fully operational: Both the behavior of a system and the system itself can be represented with the precision necessary for automation. Additionally, designs can be shown to be correct relative their specifications. We show that any specified behavior as well as the design(s) responsible for that behavior can be guaranteed to be equivalent at each level of abstraction. Moreover, we show how to construct designs that can be guaranteed to produce specified behavior. Examples include declarative knowledge and interacting systems (models) as well as procedural knowledge. Also revealed is a close relationship between possible kinds of specification and design refinements. The article concludes with a summary of the process of SA and examples providing a perspective on the role of domain independent knowledge.

The concepts introduced have motivated development of a software development system, called AutoBuilder. AutoBuilder is a major extension of the SoftBuilder development system (see [www.scandura.com](http://www.scandura.com)), and is based on patented technologies insuring that designs and/or software at each level of abstraction are both internally consistent and correct with respect to their specification.

Differences and relationships between traditional software engineering and cognitive concepts become apparent in the following discussion. This discussion includes both general motivation and necessary precision using easily-understood software and/or mathematical terminology. The article, however, does not require any specialized knowledge.

**Basic Ideas**

Concepts-- A basic assumption is that any concept or idea (hereafter concept), no matter how simple or complex, can be represented at the highest level of abstraction by a simple label. The word or label can either be newly invented or part of some predetermined vocabulary. The concept of a cognitive theory, for example, might be represented by "cognitive\_theory". Alas, we are talking here about a variable (i.e., cognitive\_theory) that can be made more precise in a variety of ways.

Refinement-- The following analysis may be less obvious. It is based on a strong hypothesis: There are only three basically different ways in which any given any entity, process, concept or idea may be made more precise. We assume that any concept may be defined more precisely, but equivalently, in terms of any one of exactly three basic kinds of entities: a) components, b) categories and c) operations. *Library*, for example, may be defined more precisely in terms of a relationship between books (component/elements) in the library. Similarly, (the set of) *automobiles* may be defined more precisely in terms of the manufacturers (Chevrolet, Mercedes, Ford, etc.), each representing a distinct category or subset of that set (of automobiles). Finally, the concept of *pole-vaulting* can be characterized more precisely in terms of the operation itself.

The same term may be refined in different ways. First, the concept “room” may be represented more precisely in terms of relationships between the floor, ceiling, walls and other components in a room. Second, “room” can be defined in terms of the various kinds of rooms that can exist (e.g., kitchens, bedrooms and combinations thereof -- such as efficiency apartments). Third, “room” can be defined in terms of operations (e.g., for constructing rooms).

Equivalently, a concept may be viewed either statically or dynamically (i.e., in terms of sets or functions - cf. Scandura, 1968). When viewed statically as a set, the concept can be decomposed (refined) into its elements or into subsets (of the set). When viewed dynamically, the concept can be viewed in terms of operations, each defined as an ordered pair of (input and output) sets.

Refining room into its component elements may be represented:

$$room \rightarrow relation (floor, ceiling, ...)$$

where *relation* is a specified relationship between the child elements *floor, ceiling, ...* . As shown below, elements in a relationship may (or may not) be independent of one another, a fact that has important implications in representing behavior.

Refining *room* into categories (subsets thereof) may be represented:

$$room \rightarrow kitchen, bedroom, ...$$

The child categories (e.g., *kitchen* and *bedroom*) represent subsets of *room*.

Refining (i.e., defining) room as an operation might be represented:

$$room \rightarrow construct\_room (boards, nails; ; room\_frame)$$

where the term *construct\_room* is used to emphasize that the term denotes an operation. Input and output variables in the child operation represent inputs and outputs in an operation. The operation viewed as a whole represents the intended meaning of the parent concept (i.e., *room*).

A key requirement in hierarchical analysis is that each child in any given refinement may be further refined. Thus, *floor* (above) might be refined into various categories of floors (*wood, brick*, etc.). Similarly, *kitchens* might be refined into components like *sinks* and *stoves*, into various (sub)categories of kitchens (country-style, modern, etc.), or as actions performed in a kitchen.

It might appear therefore that the above formulation makes it possible to represent any concept, idea or process in an arbitrarily precise manner. That is, it must be possible to continue the refinement process indefinitely, making any concept, idea or process, whether static or dynamic, as precise as may be necessary or desired.

The perceptive reader, however, may anticipate a problem with the above analysis. While easy to see how individual components and categories (elements and subsets, respectively, of a set) may be refined indefinitely, it is not clear how this is to be accomplished with relationships, or for that matter with operations. For example, where room (above) is defined as a relationship between floor, ceiling, etc., it is not clear how to further refine the relationship.

Automation (irrespective of the programming language) demands representation in terms of predetermined primitives (e.g., add, equal, etc.). Accomplishing this in a strictly hierarchical manner is possible only if one can guarantee indefinite refinement in a manner, which ensures that each level of refinement is equivalent to every other. These issues are resolved in the following analysis of definition (relational) refinements and their close relationship to operation refinements. We shall also see how these ideas apply to complex concepts or systems (often called models). Specifically, we shall see how models may be refined into two or more interacting components in which the components themselves are operations.

The above discussion pertains to concepts that can be described using nouns and verbs in a natural language. Refinement as such, however, says nothing about instances (examples) of concepts (i.e., variables) much less about behavior. The next section addresses the issue of behavioral equivalence at different levels of abstraction.

*Abstract Values.*— The issue of equivalence at different levels of hierarchical representation has a long history in software engineering (e.g., Hamilton & Zeldon in Martin, 1985, Linger et al, 1979, Sowa, 1984, 1999). In order to prove equivalence in a procedural hierarchy, for example, it is necessary to prove that representations at different levels of abstraction generate exactly the same behavior. One must show that corresponding inputs at each level of abstraction generate equivalent outputs. This is a very difficult, if not impossible, task. Proving that any non-trivial software is correct (i.e., does what it is supposed to) does not lend itself to automation.

While ignoring absolute equivalence, we seek a richer language that allows us to talk about behavior (or equivalently about values of variables) -- retaining the precision and lack of ambiguity necessary for automation. Concepts may represent variables or constants (i.e., variables with a single fixed value). In general, a variable (i.e., concept) may have any number of values (e.g., *room* represents any number of specific rooms). In practice, however, we need concern ourselves only with (usually small) numbers of distinctions (e.g., dirty versus clean rooms).

The need for precision requires that this be done in an operationally precise manner: We accomplish this by partitioning the allowable instances of each concept (i.e., values of a variable) into equivalence classes. For purposes of any given analysis, each value in an equivalence class is treated the same as any other value in that class. Thus, clean rooms may be distinguished from dirty ones, but all dirty rooms are treated the same. More precisely, the values of each variable are partitioned into mutually distinct and exhaustive equivalence classes. Each value of a variable belongs to exactly one equivalence class. In the present formulation such equivalence classes are called *abstract values*. (Abstract values provide a way to modify the meaning of concepts, whether static or dynamic, much as adjectives and adverbs do with nouns and verbs.)

In the following sections we see how abstract values provide a practical way to insure internal consistency and (abstract) correctness. The introduction of abstract values greatly simplifies the task of proving correctness, turning a complex process requiring a skilled mathematician into one that can be automated. More generally, introducing abstract values allows us to talk about input-output behavior at a higher level of abstraction – with all but only the degree of precision necessary for any given purpose.

On-going research on building general purpose intelligent tutors (e.g., Scandura, 1987, 2001a,b), for example, rests on a fundamental assumption backed by considerable empirical research (e.g., Scandura, 1971, 1973, 1977): Any process may be refined sufficiently that the terminal elements (i.e., operations) are atomic in the sense that behavior associated with any one input-output pair is equivalent to every other one. Put differently, any component (concept or process) can in principle be reduced to atomic components having a single abstract value. Obviously, this is not the case at higher levels of refinement. However, we shall see that there is a close relationship between abstract values and various kinds of refinement.

### Kinds of Behavioral Refinement

Abstract values provide the foundation for insuring equivalence between parent (higher level) and child variables. Specifically, the equivalence of parent and child variables in any refinement is based on mappings between abstract values of parent and child variables. The following analysis addresses possible kinds of refinement and the behavior associated with each, refinements in which variables are refined into categories, relationships between components or dynamic operations. Whereas category and dynamic (operation) refinements are straight forward, component refinements come in three flavors.

Category Refinements.— Parent-child mappings, which define relationships between parent and child variables in a refinement, are predetermined when variables are refined into child variables representing categories. Abstract values of the parent variable in a category refinement are necessarily shared by all categories (i.e., all child variables). This, if *room* has abstract values *presentable* and *unpresentable*, then all categories of *room* -- e.g., *bedroom* and *kitchen* -- necessarily have the same abstract values. Thus:

*room* < *presentable*, *unpresentable* > -- category  
*bedroom* < *presentable*, *unpresentable* >  
*kitchen* < *presentable* *unpresentable* >

Parent-child mappings in this case are predetermined (is an identity mapping) for each category:

PARENT-CHILD MAPPINGS -- from *bedroom* and *kitchen* to *room* -- are the identity mapping:

< *presentable*; *presentable* >  
 < *unpresentable*, *unpresentable* >

Component Refinements: Independent Child Components.-- Component refinements take an especially simple form when abstract values of the child (component) variables into which a parent variable is refined are independent of one another. In this case, the relationship between the parent variable and its children can be defined as a simple mapping from abstract values of the children variables to abstract values of the parent. For example, suppose *room* -- with abstract values *presentable* and *unpresentable* -- is refined into the components *bed* and *carpet* -- with abstract values *made* and *unmade* and *clean* and *unclean*, respectively. This can be represented more succinctly as:

*room* < *presentable*, *unpresentable* > -- component

*bed* <made, unmade>  
*carpet* <clean, unclean>

The mapping (defining the abstract equivalence between parent and children in this refinement) may be represented:

PARENT-CHILD MAPPINGS  
 <made, clean; presentable>  
 <unmade, \*; unrepresentable>  
 <\*, unclean; unrepresentable>

What this means is that a *room* being *presentable* is the same as (is abstractly equivalent to) *bed* being *made* and *carpet* being *clean*. Similarly, *room* being *unrepresentable* is equivalent to either *bed* being *unmade* or *carpet* being *unclean*.

Prototype Refinements: Variable Number of Components with a Common Structure.-- Component refinements may have a variable number of children, each with the same structure. In this case, we introduce one child, which acts as a prototype for the components. For example, if *carpet* is (further) refined into any number of component child variables, these child variables might be represented collectively by *current\_rug*. Such refinements are distinguished by the name *prototype*.

*carpet* <clean, dirty> -- prototype  
*current\_rug* <vacuumed, not\_vacuumed>

PARENT-CHILD MAPPINGS  
*current\_rug; carpet*  
 < \*vacuumed; clean>

The PARENT-CHILD MAPPING < \*vacuumed; clean> means that ALL rugs (represented by *current\_rug*) must be vacuumed for *carpet* to be *clean*; otherwise *carpet* is *dirty*.

Both component and prototype (as well as category) refinements are inherently hierarchical in the sense that the children in such refinements can be further refined (in the same way as the parents). This is not so obvious in component refinements involving (non-unary) relationships between child variables. Indeed, defining a variable in terms of a relationship between other variables is sometimes referred to as a *non-hierarchical refinement*. Before presenting a solution for non-hierarchical refinements, we first introduce dynamic (operation) refinements. We also introduce the notion of abstract input-output behavior, which plays an essential role in defining equivalence in dynamic refinements.

Dynamic Refinements.-- Dynamic concepts, typically but not necessarily verbs, represent action. Refinement in this case amounts to defining the variable in terms of the operation it represents.

*construct* <good\_construction, poor\_construction> -- dynamic

The child operation in this dynamic refinement can be represented:

*construct* (*boards, nails; ; room*)

Although syntax is arbitrary, the above operation is represented in the High Level Design (HLD) language (see [www.scandura.com](http://www.scandura.com)) -- our preferred embodiment. In HLD, parameters before the colon

represent input variables, those after the semi-colon represent output parameters and those in between represent variables that serve as both input and output.

NOTE: Some terms like *spring*, *present*, etc. may be defined either as an operation or as a static entity.

Abstract Input-Output Behavior.-- Refining concepts (i.e., variables) in terms of operations introduces two new dimensions to our analysis: First, it introduces the notion of an operation – i.e., a dynamic action or process as opposed to a static concept (or variable). Second, because operations have input and/or output (I-O) variables, I-O mappings play a direct role in Parent-Child (P-C) mappings.

For example, the child operation *construct* (*boards, nails: ; room*) in the above dynamic refinement is a mapping from its input variables, *boards* and *nails*, to its output variable *room*.

Let us assume that the operation parameters have the following abstract values:

*boards* <*strong, weak*>  
*nails* <*good, bad*>  
*room* <*sturdy, not\_sturdy*>

These abstract values make it possible to define I-O mappings associated with the child operation:

INPUT-OUTPUT MAPPINGS

<*strong, good; sturdy*>  
 <*weak, \*, not\_sturdy*>  
 <\*, *bad; not\_sturdy* >

These child input-output mappings, in turn, make it possible to define P-C mappings for the original dynamic refinement:

PARENT-CHILD MAPPINGS

<*strong, good; sturdy*> → <*good\_construction*> <sup>10</sup>  
 <*weak, \*, not\_sturdy*> → <*poor\_construction*>  
 <\*, *bad; not\_sturdy*> → <*poor\_construction*>

Notice that the I-O mappings (which define the behavior of the child operation *construct*) are inputs in the above P-C mappings. These I-O mappings are themselves mapped into the equivalent parent abstract values, *good\_construction* and *poor\_construction*. That is, each abstract I-O mapping for the child operation *construct* (*boards, nails: ; room*) is mapped into a unique abstract value of the parent *construct*.

In effect, the existence of two sets of mappings in dynamic refinements means that behavioral equivalence depends on the I-O mappings as well as the P-C mappings. The next section shows how dynamic refinements make it possible to (hierarchically) refine non-unary relational refinements.

Component Refinements: Definitions or Relations Between Components.-- In so called non-hierarchical refinements, parent abstract values depend on (non-unary) relationships between the values of child components.<sup>11</sup> Parent-child relationships in this case cannot be represented as a single mapping between parent and child abstract values. For example,

<sup>10</sup> Or, equivalently, < <*strong, good; sturdy*>; <*good\_construction*>>

<sup>11</sup> Relationships between child (sub)categories are not meaningful in specifications. Unlike component refinements, child categories necessarily have the same abstract values as their parents. For example, if *room* may be *presentable* or *un-presentable*, then so must the (sub) categories *bedroom* and *kitchen*. Relationships between

*bedroom* <*tasteful*, *not\_tasteful*> -- definition

*bed*  
*carpet*  
 <*nicely\_arranged* (*bed*, *carpet*),  
*poorly\_arranged* (*bed*, *carpet*) >

There is no way to define *tasteful* or *not-tasteful* (the abstract values of *bedroom*) in terms of abstract values of *bed* and *carpet*, irrespective of how these child variables are individually partitioned (into abstract values).

P-C mappings (defining the abstract equivalence between parent and children in this refinement) in a definition refinement are necessarily one-to-one. For example, the parent abstract values <*tasteful*, *not\_tasteful*> correspond to the child relationships <*nicely\_arranged* (*bed*, *carpet*), *poorly\_arranged* (*bed*, *carpet*) >. We represent this P-C mapping as:

PARENT-CHILD MAPPING  
 <*nicely\_arranged* (*bed*, *carpet*); *tasteful*>  
 < *poorly\_arranged* (*bed*, *carpet*); *not\_tasteful*>

The relations *nicely-arranged* and *poorly-arranged* convey general intent, but leave too much to the imagination. Exactly how are we to determine whether *bed* and *carpet* are *nicely-arranged*? There are, in fact, two options:

- a) Replace the definition refinement with an operation refinement. For example, the above P-C mappings define an equivalent child operation, namely

*nicely\_arrange\_operation* (*bed*, *carpet*: ; *room*)

where *bed* and *carpet* are inputs and *room* is the output, each variable having pre-designated abstract values (e.g., *room* -- <*tasteful*, *not\_tasteful*>). Indeed, it is a mathematical fact that any relation can be reformulated as an equivalent operation (and vice versa).

In this case, subsequent refinements of the operation take place as described above. The second option requires further explanation.

- b) Represent (i.e., refine) the relation *nicely-arranged* in terms of more elementary operations and relations (e.g., arrange *bed* symmetrically on *carpet* or place *carpet* to side of *bed* ...).

A key assumption is the possibility of refining any such relation more precisely, ultimately in terms of well-defined (i.e., unambiguous) operations and relations. Any definition refinement may (and should) be further refined as far as necessary to reduce the relationships involved to some pre-established well-defined base (e.g., pre-existing executable operations and relations such as add, subtract, equal and greater\_than). The ability to reduce abstractions to such a base is essential for automation.<sup>12</sup>

---

(sub)categories in this context are meaningless because a room in any instance is either *bedroom* or *kitchen*. Relationships between bedrooms and kitchens (e.g., relative size or functionality), are not rooms.

<sup>12</sup> Reducing abstract operations and relationships to executables is analogous to proof in mathematics. That is, making an abstract process executable is equivalent to proving an abstraction (theorem) in terms of assumed axioms.

More specifically, the goal is to make it possible to reformulate (i.e., define) refinements involving non-unary relations between components (which do not lend themselves to formal verification) in terms of ones that do.

The concept of “refinement” itself provides an instructive example. Specifically, a refinement is said to be consistent if and only if the top path generates the same result as the bottom path. We represent this as follows:

*refinement* <consistent, ~consistent>      -- too informal to verify formally

*Refinement* can be defined (refined) more precisely as:

*top\_path*  
*bottom\_path*  
 <*top\_path* = *bottom\_path*>  
 <*top\_path* <> *bottom\_path*>

This is better. We know in this case what “=” and “<>” (not =) mean, but what exactly are we comparing? *top\_path* and *bottom\_path* are almost as abstract as *consistent* and *~consistent*.

Notice, however, that each path denotes an operation (see Dynamic Refinements above). We can further refine (i.e., define) each as an operation: Thus, *top\_path* may be precisely defined as *top\_path* (*C-IN*; *P-OUT1*), where *C-IN* is the child input in the refinement and *P-OUT1* is the output obtained by the P-C mapping *C-IN* → *P-IN* followed by the *P-IN* → *P-OUT1* mapping. In effect, the original *C-IN* → *P-OUT1* mapping (i.e., the *top\_path* operation) is further refined into a sequence (see below) of (in this case) well-defined operations. The essential point is that one can refine any concept, no matter how abstract, into well-defined operations and relations.

Similarly, *bottom\_path* is defined as the *C-IN* → *C-OUT* mapping followed by *C-OUT* → *P-OUT2* mapping.

In effect, <*top\_path* = *bottom\_path*> (i.e., *consistent*) can be defined unambiguously as a simple well-defined procedure:

$$C-IN \rightarrow P-IN; P-IN \rightarrow P-OUT1; C-IN \rightarrow C-OUT; C-OUT \rightarrow P-OUT2; P-OUT1 = P-OUT2$$

Everything is now well defined. We know exactly what it means for two values to be equal, and can consequently automate the process (in essentially any implementation or programming language, including HLD).

More generally, relational (definition) refinements can always be further refined indefinitely -- thereby reducing the entities (variables) and operations involved to some pre-established base set of well-defined executable operations and relations. In this case the operations are well-defined mappings and the sole relation is equality.

The same approach can be applied to the room example above.

### **Interacting Systems - Commonly called 'Models' in Instructional Theory**

Interacting (e.g., distributed) systems involve two or more interacting components. They are like definition refinements – where each child component may be defined as an operation (see Table 1).

**Table 1: Comparison Between Relational and System/Interaction Refinements**

	<b>Definition Refinement</b>	<b>System/Interaction Refinement</b>
<b>Parent</b>	variable with 2 or more abstract values	system with 2 or more components
<b>Children</b>	relationship between two or more variables	relationship between two or more components

On deeper analysis, it becomes apparent that system refinements are nothing more than a component refinement followed by one or more dynamic refinements. Consider a railroad *crossing\_system* with abstract values  $\langle \text{safe}, \sim\text{safe} \rangle$  (where ‘safe’ means ‘safe&operational’ and ‘~safe’ means ‘unsafe-or-non-operational’). The first (i.e., definition) refinement may be represented as:

<i>crossing_system</i>	$\langle \text{safe}, \sim\text{safe} \rangle$ - component
<i>train</i>	$\langle \text{IN-crossing}, \text{OUT-crossing} \rangle$
<i>gate</i>	$\langle \text{open}, \text{closed} \rangle$
<i>signal</i>	$\langle \text{red}, \text{green} \rangle$

where the child relations  $\langle \text{IN-crossing}, \text{red}, \text{closed} \rangle$  and  $\langle \text{OUT-crossing}, \text{green}, \text{open} \rangle$  correspond to ‘safe’; and all other child relations correspond to ‘~safe’ (unsafe).

More specifically, the Parent-Child mappings in this component refinement are as follows:

$\langle \text{IN-crossing}, \text{red}, \text{closed}; \text{safe} \rangle$
$\langle \text{OUT-crossing}, \text{green}, \text{open}; \text{safe} \rangle$
$\langle \text{IN-crossing}, \text{green}, \text{open}; \sim\text{safe} \rangle$
$\langle \text{IN-c}$
$\text{rossing}, \text{green}, \text{closed}; \sim\text{safe} \rangle$
$\langle \text{IN-crossing}, \text{red}, \text{open}; \sim\text{safe} \rangle$
$\langle \text{OUT-crossing}, \text{red}, \text{open}; \sim\text{safe} \rangle$
$\langle \text{OUT-crossing}, \text{red}, \text{closed}; \sim\text{safe} \rangle$
$\langle \text{OUT-crossing}, \text{green}, \text{closed}; \sim\text{safe} \rangle$

Normally, the first step in building a distributed system is to specify a "good" system -- i.e., to specify the allowable combinations of abstract values of *train*, *gate* and *signal*. The first constraint, for example, says that when train is in the crossing (i.e., has a value belonging to the abstract value *IN-crossing*), then *signal* must be *red* and *gate* must be *closed*. As above, the first two abstract value combinations must be preserved in order for *crossing\_system* to be *safe*. In practice, *train* and *gate* are components that communicate via *signal*. When *train* is in the crossing, for example, it sends a message to *signal*, which turns *red* and sends a signal to *gate* to go down.<sup>13</sup>

Subsequent refinements dynamically refine *train*, *gate* and *signal* into corresponding operations. For example, *gate* is refined into the *gate* operation.

---

<sup>13</sup> There is a close relationship (mathematical equivalence) between definition refinements (in which abstract values of parent variables are defined in terms of relationships between values of child variables) and operations for determining whether a given relation holds. In brief, we can always define any given relationship in terms of sufficiently low level relationships, such as testing to see if two values are equal, that can be handled by contemporary automatic theorem provers (Ingo Dahn, personal communication).

PARENT *gate* <*closed, open*> - dynamic

CHILD *gate* (*signal*: ; *position*)  
*signal* <*red, green*>  
*position* <*up, down*>

#### INPUT-OUTPUT MAPPINGS FOR CHILD

<*green; up*>  
 <*red; down*>

#### PARENT-CHILD MAPPINGS

< <*red; down*>; *closed*>  
 < <*green; up*>; *open*>

The other components, *train* and *signal*, are refined similarly.

NOTE: From a software perspective, these operations may or may not have a visible interface and effectively are servers that interact by passing messages between themselves. In HLD, the servers may be viewed as event handlers with an interface and associated callback. Each component operation has a specification and a design (e.g., for the callback).

Although *train*, *gate* and *signal* interact with each other, *crossing\_system* as a whole is self-contained and does NOT have any external inputs or outputs connecting it to the outside world. As shown in the concluding examples, systems may or may not have external inputs and outputs.

### Kinds of Design Refinement

As shown above, there are only a small number of basic kinds of behavioral refinement. In each case, equivalence is defined in terms of abstract behavior. Unlike behavior, designs are prescriptions for producing behavior. They are used to represent processes, called cognitive processes in cognitive science and instructional design, and programs in software engineering.

We consider the question of equivalence in design hierarchies below. This section details the kinds of design refinements. A well known maxim in structured programming (e.g., Dijkstra, 1976) -- not to be confused with structural analysis -- is that any program can be written using three basic constructs: sequence, selection and loop. Each type of construct has common variations. The HLD language includes the following: Sequence, Parallel, Navigation Sequence, Selection (If..Then and Case), Loop (While..Do and Repeat..Until), Class Inheritance (where a parameter is the root of an object hierarchy) and Interaction (where a parameter represents an operation).

Consider the operation

*clean* (:*room*; )

where *room* is both an input and an output variable.

The abstract behavior of *clean* is defined in terms of the mapping from *room*'s abstract input values to its abstract output values. Thus, given an *unpresentable* room as input, the operation *clean* would generate a *presentable* room. There can be any number of different unpresentable rooms, just as there can be any number of presentable ones.

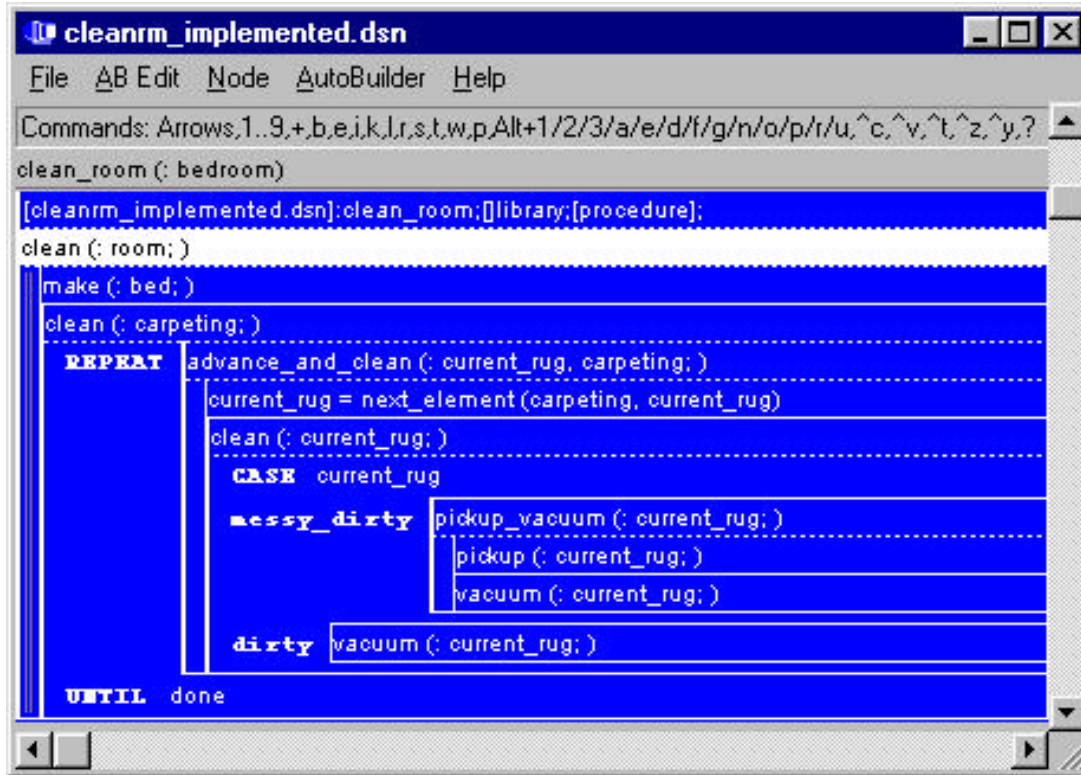


Figure 1. Sample hierarchy of High Level Design (HLD) operations for cleaning rooms.

As shown in Figure 1, the operation *clean* represents a high level design for cleaning rooms. *clean* can either be implemented directly as an executable, or as in Figure 1, the operation can be refined further. Specifically, *clean* has been refined into two parallel operations, *make* and *vacuum*. Instead of operating on *room*, however, *make* and *clean* operate on the child components *bed* and *carpeting*, respectively.

Sequence, Parallel (where the various operations can be executed in parallel), Loop (WHILE..DO and REPEAT..UNTIL), Selection (IF..THEN and CASE) are illustrated in Figure 1 . Notice that each type of refinement is represented graphically in a distinctive manner (U.S. Patent 5,262,761, Scandura, 1993). In particular:

- a) the child operations *pickup* and *vacuum* in the sequence refinement *pickup\_vacuum* must be performed one after the other. It makes no sense to vacuum before picking up what ever might be in the way.
- b) The child operations in the top-level parallel refinement *clean*, however, may be performed independently. (Parallel refinements are distinguished by vertical parallel lines.) Making a *bed* and cleaning *carpeting* are (at least for present purposes) independent operations.
- c) Which child operation (*pickup\_vacuum* and *vacuum*) is to be executed in the CASE refinement of *clean (: current\_rug; )* depends on the abstract value of *current\_rug* (i.e., *messy\_dirty* or *dirty*).
- d) The REPEAT..UNTIL refinement is represented in terms of an *advance\_and\_clean* operation followed by the condition *done* (used to determine whether to repeat the process or to exit).
- e) In the current (and preferred) embodiment, the advance and process operation (in this case *advance\_and\_clean*) is always refined into a *navigation\_sequence*. A *Navigation Sequence* is a sequence in which an initial operation assigns an alias (e.g., *current\_rug*) to a specified member of a

set (e.g., of rugs) and others process whatever (e.g., rug) that alias refers to. The refinement of *advance\_and\_clean* (; *current\_rug*, *carpeting*;) in Figure 1 is illustrative. The first child operation, *current\_rug = next\_element* (*carpeting*, *current\_rug*) serves to identify a particular rug in the set of rugs under *carpeting*. The next operation *clean* (: *current\_rug*; ) operates on it.

*Abstract Operation, Interaction* and *System* refinements in designs require a bit more elaboration.

An *Abstract Operation* refinement (cf. Scandura, 1997, 2001c) may be used when a parameter represents the root of an object hierarchy. For example, we might have an abstract operation *clean* (: *room*), where the *room* parameter is the root of a component hierarchy with the subcategories (i.e., children ) *bedroom*, *kitchen*, etc. In this case, the parameter represents a category refinement (i.e., the corresponding specification refinement is a category refinement). The parameter on which *clean* acts is a structure refined into subcategories (as in a class hierarchy in OO design). In this case, the corresponding operation *clean* is said to be an *abstract operation*.

Abstract operations have (abstract) parameters, which may vary as to structure (i.e., category or type) as well as value). These types do not have to be declared in advance as in OO programming. For example, *bedroom* and *kitchen* types implicitly also define “*efficiency\_apartment*” (which in the USA is a single room combining features of a bedroom and kitchen).

In HLD, each abstract parameter has affiliates, operations that apply to components defining the category in question. For example, *bedroom* may have affiliates for making beds; *kitchen* is likely to have an affiliate for washing dishes. Affiliates, in turn, may have affiliates in subcategories. Thus, the sub-category *eat-in-kitchen* (of *kitchen*) is likely to have an affiliate for setting tables.

Affiliates automatically apply as required by the particular structure (object) assigned to the parameter at any given point in time. Each affiliate knows its domain of applicability and is automatically executed as required during runtime. For example, assume a *room* class hierarchy has subcategories, *bedroom* and *kitchen*. The abstract operation *clean* (which operates on *room*) might have one or more affiliates for cleaning components common to rooms in general. These affiliates, in turn, might have affiliates for cleaning components (found only) in bedrooms or kitchens. When *clean* is invoked, all applicable affiliates (including affiliates of affiliates) are automatically invoked. In effect, the (object) parameters in abstract operations may have any number of types. When the operation is executed all applicable affiliates are automatically executed.

In Object Oriented (OO) design/programming, this type of situation is handled by introducing class hierarchies, where each class has associated methods, with methods associated with lower level classes invoking methods associated with those higher in a hierarchy. As shown in Scandura (1997, 2001c), however, traditional OO leads to such anomalies as “telling cars to paint themselves” or “houses to clean themselves”. It also requires specifying all possible categories in advance. Thus, if a room class hierarchy has subclasses *bedroom* and *kitchen*, then one can only talk about cleaning (generic) rooms, bedrooms and kitchens.

Correspondingly, there are two important differences between abstract operations and OO. One, abstract operations make it possible to talk directly about cleaning houses or driving cars (e.g., rather than telling a house to clean itself). That is, one can talk directly about the behavior of a program rather than in terms of passing messages between objects. Two, the parameters in abstract operations are not limited to any predefined set of subclasses (i.e., types). In addition to cleaning generic rooms, bedrooms and kitchens, for example, abstract operations automatically handle arbitrary hybrids as well. Efficiency apartments would be automatically handled, for example, because they have (some) components in both bedrooms and kitchens. Thus, presented with an efficiency apartment, the applicable affiliates would be applied automatically.

*Interaction refinements* in design hierarchies involve a parent operation in which at least one parameter is itself an operation. (As shown in Table 2 below, interaction refinements correspond to dynamic refinements in specifications.) A simple example would be an operation, *display\_dialog* (: *dialog*, *callback*; ), which displays a dialog box, with one parameter representing a callback operation (i.e., server) that responds to events triggered by the dialog. The critical property is that the two operations are independent but nevertheless interact (communicate) with one another. Events from the dialog trigger actions in the callback, which in turn may trigger changes to the dialog.

As noted above, *system refinements*, involve a pair of more basic refinements. Specification hierarchies consist of a definition (relational) refinement with each component individually further refined into an operation. In a system, these operations typically interact with one another exactly as specified by the relational refinement. Each operation in turn can be defined individually both in terms of its I-O behavior (typically called its specification) and the process (design) responsible for that behavior. B2B (e-business) systems, for example, often consist of buyers and sellers. In instructional technology the components include content (expert models), tutors and learners.

### Relationships Between Specification and Design Refinements

There is a close relationship between the various kinds of specification and design refinements (see TABLE 1). Abstract operations in designs, for example, are closely related to categories in specifications. Abstract operations operate on parameters that represent all sub-types in the associated class hierarchy. That is, parameters of abstract operations may vary as to type as well as value (e.g., Scandura, 1997, 2001c).<sup>14</sup> In interaction refinements one or more of the parameters may be operations.

TABLE 1

Specification Refinement	Design Refinement
Definition [non-hierarchical relation]	Sequence, Selection, Loop w/ operation that determines the relation
Component [is an element of]	Parallel
Prototype (variable # components)	Loop (repeat-until, do-while) [introduces variable alias]
Prototype (fixed # components)	Navigation sequence [introduces fixed aliases]
Category [is a subset of]	Abstract Operation
Category (atomic sub-categories) [is a subset]	Selection (case / if-then, based on sub-categories)
Dynamic (variable refined into operation)	Interaction (e.g., dialog with callback)
Terminal	Case (based on abstract values)

*Relationships Between Kinds of Refinements and Abstract Values.*-- There also is a close relationship between kinds of refinements and abstract values. Abstract values are only needed where actual input values must be treated differentially during execution. For example, if each component in a sequence refinement works as it should then distinguishing equivalence classes (abstract values) with respect to those components serves no useful purpose. Each component is atomic (as defined above) and expected to do what it is supposed to do irrespective of the input.

On the other hand, loop and selection refinements necessarily require abstract values. In particular, loop and IF..THEN refinements require exactly two abstract values, whereas CASE refinements may have any finite number. For heuristic reasons, meaningful names should be assigned to those abstract values.

These requirements should be anticipated in constructing specifications: In our clean room example, the abstract values *presentable* and *unpresentable* are distinguished on the input side -- the main purpose of

<sup>14</sup> Abstracting across data types in behavioral models avoids the need to directly reference (overloaded or polymorphic) functions associated with particular objects.

clean, of course, is to make every input room *presentable*. This distinction is preserved in the next refinement when *room* (using a parallel refinement) is refined into *bed* and *carpeting*. Each of the component variables retains the 'good-bad' distinction. The next (prototype) refinement also retains this distinction. The CASE refinement, however, expands the clean-dirty distinction by further distinguishing between *messy-and-dirty* and just *dirty*. (Although the example does not show it, a complete specification would explicitly include the case where *current\_rug* is *clean*.)

Abstract values associated with abstract operations (see Scandura, 1997, 2001c) are never an issue. The corresponding object parameters (e.g., *room*) may represent any number of different sub-category structures, but only one at a time. At each step in an execution, for example, *room* may be *kitchen* or *bedroom* but not both. Consequently, abstract values associated with the parent object are necessarily shared by all of the child objects. Relationships between structures never arise (as is the case with components).

### Hierarchies, Consistency and Correctness

It is obvious from the above analyses that any given concept or process may be refined repeatedly. The result in each case is a hierarchy of refinements. Hierarchical representation has long been fundamental in a wide range of areas, ranging from structured design and information modeling in software engineering to task analysis in education. What is new in the present formulation is ensuring equivalence between various levels in terms of mappings between abstract values. Successive refinement of a concept or process yields a hierarchy in which each level represents exactly the same concept or process, as defined by those mappings.

Overview.-- A specification hierarchy is consistent when the *abstract* behavior of each parent is equivalent to that of its children. A design hierarchy is abstractly correct with respect to its specification if: a) each refinement is consistent in the sense detailed below and b) each design in the hierarchy generates behavior, which is abstractly equivalent to that specified. Rather than actual values, we use equivalence classes of values (abstract values).<sup>15</sup> Relative correctness can be guaranteed, typically via simple enumeration proofs, by introducing (typically small) numbers of abstract values.<sup>16</sup>

Also, as detailed in U.S. Patent 6,275,976 (Scandura, 2001): When every refinement in a hierarchy is consistent, each design (slice through a hierarchy) generates abstract behavior equivalent to that of every other design. The patent further shows that if a design generates abstract behavior equivalent to that in a specification hierarchy, and each terminal element of the design is implemented in terms of a component that executes correctly, then the design as a whole will execute correctly with respect to the specification.

---

<sup>15</sup> Proving actual correctness of a non-trivial program is very difficult. One must show that the program generates the correct output value for each and every input value. Automatic theorem provers (ATPs), for example, typically use state transition methods and are not practical with large software systems (e.g., Dahn, personal communication). We reduce the complexity of this task by successive refinement, and specifying behavior in terms of abstract values. Practically speaking, this means that overall consistency and correctness can be determined by checking each refinement locally, a much easier task.

<sup>16</sup> For example, a room being presentable may be defined to be equivalent to a bed (component) being made and a carpet (component) clean. Determining parent-child equivalence in this example requires specifying mappings between rooms and their components as well as input-output behaviors. In category refinements, parent and children necessarily share the same abstract values: All *rooms*, as well as the subcategories *bedrooms* and *kitchens*, for example, must be *presentable* or *unpresentable*. Similarly, in mathematics the parent variable might be a rational (floating point) number  $n$  with the child sub-category variables being integer or non-integer. All would share the abstract values  $n > 0$ ,  $n = 0$  and  $n < 0$ .

Design correctness is effectively reduced to correctness of individual components used in an implementation.<sup>17</sup>

The following sections make the concepts of consistency and correctness of specification and design hierarchies more explicit.

### **Specification Hierarchies (a/k/a Behavioral Task Analysis)**

Input-output specifications, now matter how simple or how complex, can always be refined successively into arbitrarily simple elements. Put differently, input-output behavior can be represented at multiple levels of abstraction. There is an INPUT hierarchy and an OUTPUT hierarchy. For each OUTPUT variable, there is a subset of INPUT variables whose values uniquely determine the appropriate value of that OUTPUT variable. Equivalently, some function exists, which maps each combination of input values (of these INPUT variables) into a unique value of the specified OUTPUT variable. In creating a specification, we make no attempt to specify the function in question. All we care about is the I-O behavior, not how that behavior is to be generated. More specifically, we are concerned only with abstract I-O behavior.

*Equivalence in a Specification Hierarchy.*-- Strict equivalence in a specification hierarchy requires that each level of refinement represents exactly the same behavior. Proving (or disproving) equivalence mathematically requires showing that the actual behavior associated with each parent OUTPUT variable is identical to the actual behavior of the associated child variables taken collectively.

Operationally, this means that the top and bottom paths in a refinement specify exactly the same behavior. Starting with values of the children input variables: The parent-child mappings followed by the parent input-output mapping must generate exactly the same values as do the input-output mappings of the children followed by parent-child mappings of the output variables. In short, the top path must generate the same results as the bottom path – for each and every input value.

Proving consistency in any particular case requires representing the operations involved in a precise, mathematically valid way, and then showing that the resulting behavior is identical. Proving exact equivalence in practice is quite impractical. Every refinement would pose its own difficulties making the task next to impossible.

Given our assumptions as to atomicity, however, this is not necessary.

*Abstract Consistency in a Specification Hierarchy.*-- Consistency based on abstract values is much simpler. Instead of proving that parent and children variables in a specification refinement represent exactly the same behavior, it is sufficient to show equivalence with respect to abstract values. Specifically, parent mappings from abstract values of the input to output variables (parent abstract mappings) must be *equivalent* to the children abstract I-O mappings taken collectively. Operationally, this means that the top and bottom paths in a refinement specify equivalent abstract behavior. Since the number of abstract values is necessarily finite (and usually very small in number) the required proofs are much simpler and can even be automated (U.S. Patent 6,275,976). One can simply innumerate the individual cases. Consider, for example, the I-O behavior associated with *room*, where the abstract values, *presentable* and *unpresentable*, are mapped into *presentable*. At the next level, *made* and *unmade* (values of *bed*) are mapped into *made* and *clean* and *dirty* (values of *carpeting*) are mapped into *clean*. It

---

<sup>17</sup> Proving actual correctness is rarely required, but becomes progressively simpler as the design process proceeds because the implementation components become simpler. Our patent guarantees that one can always reduce complex systems to components that are "small enough".

is easy to see that the top and bottom paths give identical results. Starting with each pair of abstract values of the children (e.g., *made*, *dirty*), we get the same value of the parent output variable *room* (in all four cases). Given the children input  $\langle \textit{made}, \textit{dirty} \rangle$ : the top path consists of the P-C mapping  $\langle \textit{made}, \textit{dirty} \rangle \rightarrow \textit{unpresentable}$  followed by the parent I-O mapping  $\textit{unpresentable} \rightarrow \textit{presentable}$  (the parent output). Tracing the bottom path gives the same result:  $\langle \textit{made}, \textit{dirty} \rangle \rightarrow \langle \textit{made}, \textit{clean} \rangle$  (the child output) followed by  $\langle \textit{made}, \textit{clean} \rangle \rightarrow \textit{presentable}$  (the parent output).

When this requirement is met, we say the specification refinement is abstractly consistent (or just consistent). When each refinement in a specification hierarchy is consistent (i.e., assures equivalent abstract behavior at each level), then the specification hierarchy as a whole is said to be consistent. In fact, if abstract values are the only values (e.g., where there are only a small finite number of possible values), then the refinement is absolutely consistent.

In a consistent specification hierarchy, the abstract behavior associated with any subset of output variables is consistent with the behavior associated with any equivalent set of direct descendents.

### Design Hierarchies (a/k/a Cognitive Task Analysis)

Design Refinements and Hierarchies-- Designs deal with how behavior is generated as well as the actual input-output behavior itself. In software engineering the distinction is between specification of what a program is supposed to do and the program responsible for generating the behavior. In the study of cognition, this distinction corresponds to the difference between human (observable) behavior associated with a task (e.g., behavioral objectives) and the cognitive processes required for producing that behavior. Figure 1 above illustrates how design hierarchies may be displayed as HLD Flexforms using Scandura's SoftBuilder and AutoBuilder software.

Every slice through a design hierarchy represents a design -- an assembly of components, which collectively produce some behavior. Assuming needed components have been implemented (are executable), every design through a consistent hierarchy must generate the same behavior. For example, *make (:bed;)* and *clean (:carpeting; )* collectively must generate the same behavior as *clean (: room; )*. Similarly, the CASE refinement including the CASE selector, *current\_rug*, and the operations *pickup\_vacuum (: current\_rug; )* and *vacuum (: current\_rug; )* must generate the same behavior as *clean (: current\_rug; )*. Unfortunately, widely used design (e.g., structured design and OO design) methodologies cannot guarantee such equivalence.

Proving that different designs in a design hierarchy generate the same behavior is even more complex than proving consistency in specification hierarchies. For example, in the above example, one must prove for each and every input value of *room*, that *clean* generates exactly the same result as will the children operations *make* and *vacuum* acting collectively. More generally, the behavior generated by a design composed of the child operations in a design refinement must be equivalent to the behavior of the parent operation. In most cases, proving this to be true is extremely difficult. The kind of proof required would be different for every different operation. Consequently, it has been impractical to guarantee that a given software program will do what it is supposed to do. Even simple cases normally require a skilled logician with unlimited time (not to mention the strong programming skills also required), and this has been impractical with realistic software.

As above, we simplify the task, this time by restricting attention to design consistency and relative correctness. Design consistency means that each refinement in a design hierarchy must satisfy pre-specified constraints. These constraints are designed to insure that the abstract behavior of the parent



Not all refinements, of course, in fact very few, are consistent. For example, the following sequence refinement where a generic operation is referred to by “ $\rightarrow$ ” is not consistent:

PARENT

$A \langle a1, a2 \rangle \rightarrow C \langle c1, c2 \rangle$   
 where  $a1 \rightarrow c1$  and  $a2 \rightarrow c2$

CHILDREN

$A \langle a1, a2 \rangle \rightarrow B \langle b1 \rangle$   
 where  $a1 \ \& \ a2 \rightarrow b1$  and

$B \langle b1, b2 \rangle \rightarrow C \langle c1, c2 \rangle$   
 where  $b1 \rightarrow c1$  and  $b2 \rightarrow c2$ .

According to U.S. Patent 6,275,976 (Scandura, 2001d) a *sequence refinement is minimally consistent* if and only if "Every input variable of a child is: a) an input or input-output variable of the parent, b) a child of an input or input-output variable of a parent variable or c) the output variable of a previous child" plus 6 other conditions. (Reference to a variable in this context includes all of its abstract values.) This sequence consistency rule requires  $b2$  to be the output of a previous operation (i.e.,  $A \rightarrow B$ ) -- which it is not. Hence, this refinement is not consistent.

To summarize, partitioning the values of input and output variables into equivalence classes (i.e., abstract values) and using those abstract values to define (abstract) relationships between parent and child variables (P-C mappings) and (abstract) I-O behavior, makes it possible to impose formal consistency constraints on variables (and their abstract values) associated with various kinds of refinements. These consistency constraints must insure that one gets exactly the same abstract behavior via both the top and bottom paths in a design refinement. If each refinement in a hierarchy of operations generates equivalent behavior, then we say that the design hierarchy is internally consistent.

*Design (Abstract) Correctness.*-- The term "consistency" is used to indicate that a design refinement satisfies the specified constraints. By way of contrast, "(abstract) correctness", which involves abstract behavior, can only be judged relative to some specification. That is, (abstract) correctness requires that the initial and ending abstract values of variables in a design must satisfy the input-output specifications. Correctness of a design (or program) with respect to its specification means that behavior generated by the design is consistent with that in the specification. If a consistent design hierarchy generates the same abstract behavior as a (consistent) specification hierarchy, then we say that the design hierarchy is relatively (or abstractly) correct with respect to its specification. From an instructional design perspective, correctness reduces to guaranteeing that what is taught, if learned, must enable the learner to satisfy whatever behavioral objectives have been set.

Guaranteeing correctness, even abstract correctness, is not a simple task. Each operation in a design has its own specification, and will often involve variables (e.g., intermediate variables generated during the course of execution) that are not referenced in the specification.

Rather than trying to prove that a given design generates specified behavior for each and every combination of inputs, we simplify the problem in two ways:

First, we restrict attention to abstract values. Rather than showing that a design generates a unique specified output for each combination of input values, we show that the design generates a unique abstract output value for each combination of abstract input values (in the specification). Limiting attention to abstract values is not as restrictive as one might think. As detailed below, abstract values are introduced

where and only where needed to distinguish critical boundary conditions (e.g., conditions in selection and loop refinements).

Second, we reduce the problem of proving correctness of a design (with respect to its specification) to proving correctness of individual refinements in the design hierarchy (on which the design depends). We avoid the difficult problem of proving that an elaborated (full) design is correct by breaking the problem into bite-sized pieces.

Proving abstract correctness is very simple when talking about the top level in a design hierarchy. Because the top-level operation and corresponding spec refinements necessarily involve the same input and output variables and abstract values, proving correctness is essentially a tautology. Hence, if the parent and child operations in each refinement in a design hierarchy can be shown to generate abstractly equivalent behavior, then each design in the design hierarchy will necessarily (by a simple inductive proof) be correct with respect to the specification hierarchy.<sup>20</sup>

As further detailed in U.S. Patent 6,275,976 (Scandura, 2001d), the abstract behavior of any design (slice thru a hierarchy) as a whole will match specified abstract behavior as long as: a) the top level in the design hierarchy is consistent with the specification, b) each refinement in the design hierarchy is consistent and c) each initial and ending abstract value of a variable in the design, whose abstract behavior has been specified, matches the specified abstract input and output values. Further, the design will be absolutely correct to the extent that each operation in the design does what it is specified to do (i.e., is atomic, e.g., Scandura, 1971, 1973, or equivalently generates its own specified behavior).

Moreover, the above consistency rules, together with the relationships specified in Table 2 between specification and design refinements, make it possible to automatically construct children in a design refinement from their parent in a way that ensures internal consistency (and hence abstract correctness with respect to specifications). These constraints together with those imposed by the top level in a design hierarchy also make it possible to automatically construct parent abstractions from children with minimal or no input from designers. Although it is beyond the scope of the current article, human input as to designer intent is needed only to the extent that design abstraction calls for parallel data abstraction.

*Role of Abstract Values in Proving Correctness.*—To reiterate, a key assumption in proving correctness is that each and every design component does what it is supposed to do. A design, which has been shown to be abstractly correct, will actually execute correctly with respect to its specification precisely to the extent that each component works without error. The term "relative correctness" emphasizes this fact.

The close relationship between conditional refinements and abstract values dramatically reduces the need to introduce new abstract values. Abstract values are only needed where input values are treated differentially during run time. Assuming that each component does what it is supposed to do, then distinguishing equivalence classes (abstract values) serves no purpose in some refinements (e.g., sequence refinements). On the other hand, abstract values play a critical role in selection and loop refinements.

---

<sup>20</sup> A design hierarchy may include some variables and/or abstract values, which are not present in the corresponding specification hierarchy. Indeed, the requirement that designs actually execute typically requires further elaboration of design hierarchies (as compared to specification hierarchies).

In checking correctness, those design variables and abstract values of a design variable that correspond to input and output values in a specification must be distinguished from those that do not (i.e., which are intermediate and occur only during runtime). Proving correctness of a design with respect to a specification involves only those variables and abstract values that have been explicitly specified. Intermediate variables and/or abstract values, which arise only during execution of a design, are ignored.

See the above section on Relationships Between Kinds of Refinements and Abstract Values for further discussion.

### Process of Structural Analysis

Having detailed core essentials, consider the process of Structural Analysis (SA) more globally, as it would be applied in creating a KR to be used in an ITS (cf. Scandura, 2001a, b).

Observable Constructs.-- Specifying observable behavior might seem so obvious that it hardly needs elaboration. This may be true with computers where inputs are reduced to ASCII characters, keystrokes and other well-defined (e.g., mouse) events. It is not true in talking about human behavior. What serve as effective inputs and outputs depends inextricably on what can safely be assumed to act as inputs and/or outputs in a given context (e.g., for given technology or population of learners). As above, an observable input or output (I-O) element in SA might be any concept, process, thing or idea. *Room, number, a spoken phrase, the SLT* are all possible I-O. The primary requisite is that input or output must be atomic (directly perceived) elements.

Hierarchical Representation of Observables.-- As described above, any I-O element can be represented as a hierarchy of elements at multiple levels of abstraction. That is, the same element can be represented as an atomic whole, or in terms of the components, categories or operations of which it is composed – as in the specification hierarchy below (for the design hierarchy in Figure 1). This hierarchy represents I-O variables and their (abstract) values at various levels of refinement. *Room*, for example, is more fully defined in terms of its components, *bed* and *carpeting*. *Carpeting*, in turn, is further decomposed into one or more scatter *rugs*.

#### INPUT

<i>room</i> < <i>presentable, unpresentable</i> >	- component refinement
<i>bed</i> < <i>made, unmade</i> >	- atomic
<i>carpeting</i> < <i>clean, unclean</i> >	- prototype refinement
<i>current_rug</i> < <i>clean, dirty, messy&amp;dirty</i> >	

#### OUTPUT

<i>room</i> < <i>presentable</i> >	- component refinement
<i>bed</i> < <i>made</i> >	- atomic
<i>carpeting</i> < <i>clean</i> >	- prototype refinement
<i>current_rug</i> < <i>clean</i> >	

In short, I-O behavior can be described at any number of levels of abstraction. An essential constraint is that behavioral equivalence must be preserved at all levels of abstraction. In this example, *room* being *presentable* is precisely equivalent to *bed* being *made* and *carpeting* being *clean*.<sup>21</sup>

Problems.-- Problem have Givens and Goals. Givens are an input structure, where variables in the structure have been assigned values (e.g., *rug* is *dirty*). Goals are data structures to which values are to be assigned (derived by applying solution rules). A Plan is a specification and is similar to a problem except

---

<sup>21</sup> This example also illustrates an important difference between I-O specifications and corresponding designs (see Figure 1). There is exactly one operation in Figure 1 for each specified I-O mapping -- e.g., the operation *clean* (:*carpeting*;) generates the specified abstract value as do the subordinate operations *make* (: *bed*;) , *clean* (: *carpeting*;) and *vacuum* (: *current\_rug*;) . In addition, however, *current\_rug* in *pickup\_vacuum* (: *current\_rug*;) takes on the intermediate abstract value *dirty* in the (final) sequence refinement of *pickup\_vacuum*. Specifically, the initially specified abstract input values of *current\_rug* are <*messy\_dirty, dirty, clean*>. The ending abstract value is <*clean*>. In the sequence refinement, *pick-up* (: *current\_rug*;) followed by *vacuum* (: *current\_rug*;) , *current\_rug* takes on the intermediate (abstract) value *dirty* in addition to the initially specified value *messy&dirty* and the ending value *clean*.

that no input values are assigned. In short, problems (or tasks) may be defined simply as specifications (as in software engineering) in which the input variables have been initialized (i.e., assigned values).

For example, instead of just saying that the input *room* may be *presentable* or *unpresentable*, the input for a given problem would be a particular room. For purposes of SA, this room would be classified as *presentable* or *unpresentable*, and considered equivalent to every other similarly classified room.

Similarly, problems associated with an interacting system would consist of initial states for each of the interacting processes – a train in the crossing, with the gate down and the signal red, where the goal is to specify what happens when the train leaves the crossing. So-called declarative knowledge in this case would include being able to classify each such configuration as *safe* (and operational) or *unsafe*. When a specified input variable is a prototype, the corresponding variable gives in the problem will include a specific number of instances of that prototype. For example, carpeting in the clean room example is a prototype indicating that there may be any number of (e.g., scatter) rugs. Any particular room will necessarily contain some specific number of rugs (including zero).

As noted below, higher order problems, in which the Givens and/or Goals contain solution rules (see below) also play a central role in SA.

*Problem Domain.*-- Problem domains in SA may be any set of problems (or I-O pairs), including outputs without specified inputs, that happen to be of interest to an outside observer (e.g., an ITS Author). Problem domains may be domain-specific and well-defined or otherwise (i.e., ill-defined).

Well-defined I-O domains correspond to behavioral objectives (specifications in software engineering) and are semantically meaningful sets of inputs and outputs in which there is a unique output for each input. Simple examples include:

*column subtraction problems --> differences*  
*verbs --> participles (ing endings)*  
*[real world object(s) + real world action] --> "subject" "verb"*

Ill-defined domains have “fuzzy” borders, meaning that it is not possible to define such domains in terms of any finite number of well-defined and independent (sub)domains. We return to these below.

*Design Hierarchies.*-- As above, design hierarchies represent solution procedures at multiple levels of abstraction. Such designs have been referred to in the literature as SLT rules (e.g., Scandura, 2001a,b) to distinguish them from production systems. The following operations (i.e., designs) correspond, respectively, to the above problems. Each generates a unique output for each input:

*column subtraction*  
*add "ing" and drop the final 'e' (as appropriate)*  
*write the subject (name of object), then the verb with a "matching" ending.*

Like I-O elements, such operations may be represented as hierarchies at multiple levels of abstraction. Each operation/design includes a domain, range (collectively an input-output plan) and procedure. As shown in Figure 1 and the above specification hierarchy (for *room*), there is an inverse relationship between the level of operations in any such hierarchy and the complexity of the data structures associated with the corresponding parameters (in each operation's domain and range). For example, the *clean* operation operates on *room* as a whole, where *room* represents the full structure shown above. The lower level rule *make* operates on *bed*, which in this example is a simple (i.e., atomic) element.

Higher order designs are designs that include other designs in their domain or range (see examples below).

Ill-Defined Domains and Higher Order Problems.-- Content is not restricted to well-defined domains. Ill-defined domains are arbitrary sets of I-O pairs, where the same input (in different contexts) may be paired with different outputs, where no inputs may be specified and/or where no finite set of solution rules is known to exist. Many real world domains are of this sort:(a) chess positions and corresponding moves, (b) poems covering given topics and, (c) mathematical theorems and proofs.

Unlike "knowledge engineering" in AI, which tends to focus on domain specific knowledge in such cases, SA is equally concerned with higher order rules (i.e., designs) that are domain independent. Higher order rules play a central role in representing the knowledge associated with such domains. A higher order rule is a rule in which (some) inputs and/or outputs are themselves rules. One higher order rule, for example, might construct rules for solving different kinds of geometry construction problems from more basic operations (e.g., using a straight edge and compass, see Polya, 1960, Scandura, Durnin & Wulfbeck, 1974). Another (in the mind of great 19th century inventors) converted manual operations (e.g., using brooms for cleaning) into analogous ones using electric power (e.g., vacuum cleaner).

To understand automated (or human) behavior in such situations, it is essential to know what kinds of competence may be required. To better appreciate the approach taken in SA, consider knowledge engineering in Artificial Intelligence (AI), with specific reference to production systems. The basic idea is to identify productions used by domain experts in solving problems associated with the domain in question.

During problem solving, the basic approach is to systematically search through a space of possible next (or previous) productions, trying and/or rejecting them, in turn, until a solution is found or the search space has been exhausted. The problem has been likened to constructing a bridge across a river on a foggy day, not knowing whether the river has made a turn. This strategy cannot succeed in its simplest form because the number of possibilities grows too quickly with the depth of the search space. Heuristics (which are essentially higher order rules) are typically used to increase search efficiency. Methods used to identify such heuristics, however, have been largely ad hoc.

In the next section we see that higher order rules arise naturally during the course of SA.

Structural Analysis: Analyzing Content/Problem Domains.—The process of SA applies to essentially any content or problem domain. For convenience and continuity with previous literature, the terms rule and design are used interchangeably.

1. The first step in SA is to define the domain, in particular the observable inputs and outputs characterizing that domain. As above, these inputs and outputs are represented as hierarchies. Identifying I-O is not always a trivial task. In teaching a non-swimmer to swim, for example, it is not immediately obvious that the inputs and outputs refer to operations. In this case, the task may be defined in terms of converting one (already known) operation -- namely floundering in the water-- to another (i.e., moving through the water while staying afloat by kicking and stroking with one's arms). Viewed in this way it is clear why starting the beginner with a "float" is often more efficient than starting without. There is less to learn! The learner never has to learn the equivalent of what is called the "dead-man's float" because the float serves to keep the person above water. Learning to kick and stroke serves the dual purpose of keeping the person afloat when the aid is later removed.

It is instructive to consider the role of (e.g., spatial) images in this context. The basic question is how to represent continuous reality in digital form. The first thing to observe is that images can also be

represented hierarchically. Ignoring retinal imprints (and their associated physiology) images may be viewed as icons – symbols that share features in common with the reality they represent (e.g., Scandura, 1970). Such icons correspond to the top level in a hierarchy, representing say, a soccer field and the players on it. Lower levels in the hierarchy represent more detail – the players, position of the ball, etc.

Problem domains may even be specified entirely in terms of outputs. Consider situations where the inputs in question can only be determined in terms of the solution procedures used. Domains characterized by creative tasks, such as proving (or disproving) mathematical theorems, provide a case in point. Such proofs typically are based on large numbers of (input) assumptions and other theorems (lemmas, etc.), and often involve inordinately large search spaces. Other examples include chess where the same goal may be achieved from a variety of chess positions. Although outputs may be highly abstract in nature (e.g., writing an "interesting" poem), specifying outputs in a domain is a necessary minimum in SLT for predictive purposes. Otherwise, one is reduced to observing and reporting behavior after the fact.

2. The second step in SA depends on whether or not the problem domain is well-defined. If so, one or more solution rules (represented as hierarchies) may be constructed. If the domain is ill-defined, then one must first extract well-defined sub-domains of problems in the domain to serve as the starting point. Each well-defined sub-domain, in turn, will have at least one solution rule.

3. A consistent design hierarchy is constructed for each well-defined domain. As above, highly systematic methods have been developed for this purpose<sup>22</sup>. Suffice it to say that each design (or "slice") through a given design hierarchy is behaviorally equivalent to every other.

The only difference is in the abstraction level of the operations involved and the complexity of the data structures associated with input and output parameters of the operations. As illustrated in the above clean room example, higher-level rules operate on correspondingly more complex parameters. At the top of the hierarchy, for example, the *clean* operation in *clean (room)* is simple (and highly abstract) while the parameter *room* has a relatively complex structure. The next level operations, *make (bed)* and *clean (carpeting)* provide more detail as to process but correspondingly operate on less complex structures.

Design hierarchies for solving prototype problems in the well-defined sub-domains represent a first level characterization of the competence associated with the given problem domain. Starting with only the prototypes, however, typically leaves many "gaps" in complex domains. Many problems (I-O pairs) are generally not covered by any prototype.

4. SA offers an explicit way to fill those gaps. This is accomplished by first converting each rule into a higher order problem whose goal includes that rule (see U.S. Patent 6,275,976, Scandura, 2001d). Higher order problems, in turn, are generalized to I-O specifications, or Plans, by abstracting the originally given values. Specifically, the higher order plans include previously identified rules in their ranges and/or domains.<sup>23</sup>

5. A higher order rule is then constructed for solving problems specified by each higher order plan. In general, a higher order rule will not only generate the previously identified (lower order) rule, but other rules as well (depending on values assigned to input data elements). In effect, introducing higher order

<sup>22</sup> Key aspects of the process are detailed in U.S. Patent 6,275,976 (Scandura, 2001d).

<sup>23</sup> Higher order problems and plans (specifications) are closely related, and play a central role in the universal control mechanism in SLT. Although discussion is beyond the scope of this article (see U.S. Patent 6,275,976, Scandura, 2001d), it is worth noting that this mechanism handles conflict resolution very differently than control mechanisms in production systems. Rather than being an integral part of the control mechanism itself, conflicts are resolved by simply adding (higher order) SLT selection rules as desired. The control mechanism itself remains unchanged.

rules serves to fill gaps (in ill-defined domains). Furthermore, rules that can be generated from existing rules become redundant and may be eliminated from the set of rules characterizing competence (underlying the given domain).

This process of converting rules into higher order problems and constructing higher order solution rules may be continued indefinitely. At each stage, higher order rules become inputs and/or outputs for still higher order plans and rules. Empirical research (e.g., Scandura, 1984) demonstrates that the introduction of higher order rules, and the elimination of redundant ones, has two beneficial effects: a) Individual rules (including higher order ones) tend to become simpler and b) the collective generating power of the rules becomes qualitatively greater. That is, much greater varieties of problems in the original domains can be accounted for via the identified rules. Repeating the process makes it possible to account for arbitrarily and sufficiently broad ranges of tasks in the problem domain with sufficiently small sets of higher and lower rules.

Discussion. There are indefinitely large numbers of real world domains. Analyzing such domains would correspondingly require an indefinitely large amount of work -- especially if one had to start SA from the beginning with the introduction of each new or modification to an existing domain. In fact, this is *not* necessary. SA is a strictly cumulative process. Many rules, particularly higher order rules, are applicable across multiple domains. Rather than having to start over with each new kind of problem, one can build on the results of previous SA: Simply add the new problems to the old domain, introducing new prototypes as necessary. The process of SA continues as before.

To summarize, competence underlying given I-O domains is represented by a set of higher and lower order rules, where each rule is represented as a hierarchy. Each hierarchy represents an equivalence class of designs, each representing a different level of abstraction. Given any domain characterizing the behavior of interest, SA offers a systematic means of constructing competence for solving “sufficiently broad” sets of problems in the domain.

The same domain may be analyzed from any number of perspectives. That is, any problem in a given domain may be solvable in any number of different ways (using rules associated with the differing perspectives). The fact that European students are commonly taught to solve column subtraction problems by equal additions while American students use borrowing is well documented in the literature (e.g., Durnin & Scandura, 1973). In practice, the number of different perspectives required for instructional purposes is usually quite small. In most domains, small numbers of alternative rule sets are sufficient to account for the behavior of most learners.

In performing SA, it also is important to recognize that what serve as inputs and outputs may vary across apparently similar sub-populations of learners. Different sub-populations, for example, may make differential use of spatial versus non-spatial inputs in direction finding.<sup>24</sup> Similarly in soccer, for example, a valid representation will depend on whether the field is viewed from the perspective of a coach (with all players viewed from a common perspective) or one of the players. In SLT, specifying what are the effective inputs and outputs often goes a long way in defining what is to be learned.

SA has been used in practice for many years, and numerous examples exist in the literature (Scandura, 1982; Scandura, 1977c; Scandura & Scandura, 1980; Scandura, 1997; Scandura & Scandura, 1999). These include both broad-based comprehensive analyses (Scandura, Durnin, Ehrenpries et al, 1971; Scandura & Scandura, 1980) and smaller more intensive analyses (e.g., Scandura, Durnin & Wulfbeck,

---

<sup>24</sup> Such differences may correlate with gender (due to well-documented hemispheric differences in male and female brains). Although such differences would be of interest in defining sub-domains of problems, SLT puts the emphasis on individual behavior as opposed to group averages.

1974; Wulfeck & Scandura, 1977). SA has evolved considerably since its inception in the 1960s through the early 1980s (e.g., Scandura, 1964a,b; Scandura, Durnin & Wulfeck, 1974; Scandura, 1977c; Scandura & Scandura, 1980; Scandura, 1984b). Indeed, the method itself has become increasingly systematic and repeatable over the years, leading today to precise automated methods in software engineering (U.S. Patent 6,275,976, Scandura, 2001d). Other early references of interest include (Roughead & Scandura, 1968; Scandura, 1968, 1969a, b; Scandura, 1971b; Scandura, 1972).

### Domain Independent Knowledge

Domain independent higher order knowledge has been the subject of much research in the past, and frequently arises in complex problem solving and other creative tasks. While considerable effort may be required in such analyses, SA has been successfully applied in a wide range of complex domains to considerable advantage (e.g., Scandura, 1964a,b; Scandura, Durnin & Wulfeck, 1974; Scandura, 1977c; Scandura & Scandura, 1980; Scandura, 1984b).

Although this article focuses on domain specific knowledge, the examples below add perspective on the role of domain independent knowledge.

EXAMPLE 1 – Consider the task of making an arbitrary refinement consistent. Specifically, consider the I-O specification:

IN *refinement* <consistent, ~consistent>      OUT *refinement* <consistent>

It is easy to introduce a design operation *make\_consistent* that satisfies this specification. However, it is very hard to define *make\_consistent* in a way, which guarantees that any given refinement will be consistent.

This type of situation typically is better handled by introducing higher order rules. Thus, one might consider a set of prototype refinement defects, each of which lends itself to some kind of definable “fix”. Having done this, one would look for “common denominators”, ways (higher order rules) for deriving such fixes from more basic operations. Actually doing so is left as a challenging research project.

EXAMPLE 2 – Devising a design for converting given railroad crossings (see above) into good ones raises similar issues. Whereas self-contained systems have no inputs or outputs themselves, they may serve as inputs and/or outputs for other operations. Since systems themselves include designs (when refined), such operations may be called higher order. Whether or not a rule is of higher order, however, is not a characteristic of the rule itself but rather the role it plays in any given analysis.

EXAMPLE 3 – Similarly, one can envision a higher order design for constructing a good crossing system in the first place.

IN empty      OUT crossing <good >

The system in this case may be viewed as having been created from some undeterminable set of inputs (i.e., made available by some oracle). The top-level system is simply its name with one default abstract value (“good” meaning the system is supposed to work).

EXAMPLE 4: -- The final example illustrates that systems may receive input from the outside world (or conversely generate output to / effect the outside world). SoftBuilder fully supports constructing such systems (see [www.scandura.com](http://www.scandura.com)).



preferred implementation. For example, one can clean something by washing it, vacuuming, etc. Which implementation is preferred necessarily depends on what one wants or must do. The essential requirement is to insure consistency at the various levels of refinement, NOT to require that any given term be refined in the same way by all designers for all purposes.

The critical assumption is that any given idea (e.g., represented by a word) can ALWAYS be refined using one of the refinement types specified in this article. One can challenge the assumption by coming up with a concept that can be automated but cannot be refined using one of these refinement types. The key assumption, one that can legitimately be challenged, is that this refinement process can ALWAYS be continued until the terminal elements in the refinement hierarchy are atomic – that is, are unambiguously clear and precise as to meaning. Atomic elements correspond to executable components in some programming language. These components include such things as “add”, ‘find\_the\_length\_of’, ‘equal’, etc.

Emphasis in this article has been put on procedural and more complex forms of content such as interacting systems or models (even higher order knowledge). Although not stressed herein, there is every reason to believe that the same methods may be (indeed have been) applied to simpler forms of content, including factual knowledge, conceptual learning or case-based reasoning -- even declarative knowledge generally, which often takes the form of facts or relationships.

The second claim pertains to future potential. This claim can only be confirmed by constructing a truly general purpose intelligent tutor (GPIT). Given the representation of ANY content (the expert model), which has been analyzed as above, this GPIT must be able to: a) automatically construct suitable test items and instruction, b) maintain an up to date representation of the learner model (showing what the learner does and does not know) and c) ideally to present those test items and instruction so as to optimize learning. Ideally, it should be able to do this better than even the most skilled teacher.

Early attempts in this direction were made in the mid 1970s (see Scandura, Stone & Scandura, 1996, Scandura, 1997, Scandura & Scandura, 1997). Limited processing power, inadequate software tools and an incomplete (non-hierarchical) representation of knowledge, however, limited the results to simple procedural knowledge. Even here, it was not possible to completely separate tutorial expertise from the representation of content, or to generate problems from that representation. Scandura (2001a,b) recently outlined the form a GPIT might take. The general idea is: a) to test on those items in the given knowledge representation hierarchy, which maximize the amount of information gained about the learner model and b) to provide instruction only where all prerequisites have been mastered. The general principles seem about right, but again the real test will be to add the rigor necessary to automate these generalities while at the same time generalizing the results to other kinds of knowledge. We have made concrete progress in this direction and hope to report results in the near term. The key in evaluating success will be the ability to replace one KR with any other without change to the GPIT itself.

In some respects, the kind of GPIT envisioned might be expected to surpass what even gifted teachers are able to do. It would be very difficult, for example, for any human to keep in mind much of the detailed information that would be available to an automated tutor – for example, exactly what the learner does and does not know at each point in time or exactly what will result in the most efficient learning.

Despite our obvious optimism, we conclude with a caution: There is another kind of expertise that is unlikely to be automated in the near future. This has to do with the ability of gifted teachers to extend the domain of applicability dynamically during instruction. For example, a learner may ask a question which lies outside the domain represented and immediately available to a GPIT. While the domain might, through subsequent analysis, be extended to encompass the new content, there would be no way for the GPIT to do this unless the ability to perform SA can be fully automated and incorporated into the GPIT.

## References

- Anderson, J. R. *The architecture of cognition*. Cambridge, MA: Harvard University Press, 1983.
- Anderson, J. R. The expert Model. In M. C Polson & J. J. Richardson, (Eds.) *Intelligent tutoring systems: lessons learned*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1988.
- Anderson, J. R. *Rules of the Mind*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1993.
- Anderson, J. R., Boyle, C. F., Corbett, A. T., and Lewis, M. W. Cognitive modelling and intelligent tutoring. *Artificial Intelligence* 42 (1), Special Issue on Artificial Intelligence. 1990, 7-49.
- Anderson, J. R., Corbett, A. T., Koedinger, K. R., and Pelletier, R. Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences* 1995, 4 (2), 167-207.
- Anderson, J. R. and Lebiere, C. *Atomic Components of Thought*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1998.
- Dijkstra, E. *A discipline of Programming*. Englewood Cliffs, NJ: Prentice Hall, 1976.
- Doignon, J. P. & Falmagne, J. C. *Knowledge Spaces*. Berlin: Springer, 1999.
- DuBoulay, Benedict. Can we learn from ITSs? *International Journal of Artificial Intelligence in Education*, 2000, 11, 1040-1049.
- Düntch, I. and Gediga, G. (1995) Skills and knowledge structures. *British Journal for Mathematical and Statistical Psychology* 48, 9-27.
- Falmagne, J.C., Doignon, J.-P., Koppen, M., Villano, M., & Johannesen, L. (1990) Introduction to knowledge spaces: how to build, test and search them. *Psychological Review*, 97 (2), 201-224.
- Gagne, R. M. *Conditions of Learning*. NY: Holt, Rinehart & Winston, 1965.
- Ganter, B. and Wille, R. *Formal Concept Analysis: Mathematical Foundations*. Berlin: Springer, 1999.
- Gediga, G. and Düntch, I. Knowledge structures and their application in CALL systems. In: S. Jager, J. Nerbonne and A. van Essen (eds.): *Language Teaching and Language Technology*. 1998, pp. 177-186.
- Gruber, T. R. Towards principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies* 1995, 43 (5/6), 907-928.
- Guarino, N. (1998) Formal ontology and information systems. In: N. Guarino (ed.) *Formal Ontology in Information Systems*. Amsterdam: IOS Press, pp. 3-19.
- Linger, R. C., H. D. Mills, & B. J. Witt. *Structured Programming: Theory & Practice*. Reading, MA: Addison-Wesley, 1979.
- Martin, J. *System Design from Provably Correct Constructs*. Englewood Cliffs, NJ: Prentice Hall, 1985.
- Miller, G. A. The magic number seven, plus or minus two. *Psychological Review*, 1956.
- Newell, A. & Simon, H. A. *Problem solving*. Englewood Cliffs, NJ: Prentice Hall, 1972.
- Psootka, J. M., Massey, L.D. & Mutte, S. A.. Introduction. Joseph Psootka, L. Dan Massey and Sharon A. Mutter (Eds.) *Intelligent tutoring systems: lessons learned*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1988.
- Roughead, W.G. & Scandura, J.M. "What is learned" in mathematical discovery. *Journal of Educational Psychology*, 1968, 59, 283-298.
- Shute, V. J., Torreano, L. A. & Willis, R. E.. Exploratory Test of an Automated Knowledge Elicitation and Organization Tool. *International Journal of Artificial Intelligence in Education* (1999), 10, 365-384.
- Scandura, J. M. Abstract card tasks for use in problem solving research. *Journal of Experimental Education*, 1964, 33, 145-148. (a)
- Scandura, J. M. An analysis of exposition and discovery modes of problem solving instruction. *Journal of Experimental Education*, 1964, 33, 149-159. (b)
- Scandura, J.M. New directions for theory and research on rule learning: I. A set-function language. *Acta Psychologica*, 1968, 28, 301-321.
- Scandura, J.M. New directions for theory and research on rule learning: II. Empirical Research. *Acta Psychologica*, 1969, 29, 101-133. (a)
- Scandura, J.M. New directions for theory and research on rule learning: III. Analyses and theoretical direction. *Acta Psychologica*, 1969, 29, 205-227. (b)
- Scandura, J.M. The role of rules in behavior: Toward an operational definition of what (rule) is learned. *Psychological Review*, 1970, 77, 516-533.
- Scandura, J.M. A theory of mathematical knowledge: Can rules account for creative behavior? *Journal of Research in Mathematics Education*, 1971, 2, 183-196.
- Scandura, J.M. Deterministic theorizing in structural learning: Three levels of empiricism. *Journal of Structural Learning*, 1971, 3, 21-53.
- Scandura, J.M., Durmin, J.H., Ehrenpreis, W., et al *An algorithmic approach to mathematics: Concrete behavioral foundations*. New York: Harper & Row, 1971.
- Scandura, J.M. What is a rule? *Journal of Educational Psychology*, 1972, 63, 179-185.
- Scandura, J. M. *Structural learning I: Theory and research* London/New York: Gordon & Breach Science Publishers, 1973.

- Scandura, J. M., Dumin, J. H., & Wulfeck, W. H., II. Higher-order rule characterization of heuristics for compass and straight-edge constructions in geometry. *Artificial Intelligence*, 1974, 5, 149-183.
- Scandura, J. M.. Human Problem Solving. Academic Press, 1977.
- Scandura, J.M. & Scandura, A.B. *Structural Learning and Concrete Operations: An Approach to Piagetian Conservation*. Praeger, N.Y., 1980.
- Scandura, J.M. Structural (cognitive task) analysis: A Method for Analyzing Content. Part I: Background and Empirical Research. *Journal of Structural Learning*, 1982, 7, 101-114.
- Scandura, J. M. Structural Analysis. Part II: Toward precision, objectivity and systematization. *Journal of Structural Learning*, 1984, 8, 1-28. Also in Proceedings, XXIII Congress of Psychology, Acapulco, Mexico, Sept. 2-8, 1984. (a)
- Scandura, J.M. Structural Analysis. Part III: Validity and reliability. *Journal of Structural Learning*, 1984, 8, 173-193. (b)
- Scandura, J.M., Stone, DC, & Scandura, A .B. "An intelligent RuleTutor CBI system for diagnostic testing and instruction". *Journal of Structural Learning*, 1986, 9, 15-61.
- Scandura, J.M.. & Scandura, A.M. The intelligent RuleTutor: a structured approach to intelligent tutoring - Phase II. *Journal of Structural Learning*, 1987, 9, 195-259.
- Scandura, J.M. A structured approach to intelligent tutoring. Chapter 14 in D.H. Jonassen (Ed.) *Instructional design for microcomputer software*. Hillsdale, N.J.: Erlbaum, 1987, 347-379.
- Scandura, J.M. Role of relativistic knowledge in intelligent tutoring. *Computers in Human Behavior*, 1988, 4, 53-64.
- Scandura, J.M. A cognitive approach to software development: The PRODOC environment and associated methodology. In D. Partridge (Ed.): *Artificial Intelligence and Software Engineering*. Norwood, NJ: Ablex Pub., 1991, Chapter 5, pp. 115-138.
- Scandura, J.M. Automating renewal and conversion of legacy code into Ada: A cognitive approach. *IEEE Computer*, 1994, April, 55-61.
- Scandura, J.M. Cognitive analysis, design and programming: next generation OO paradigm. *Journal of Structural Learning and Intelligent Systems*, 1997, 13, 1, 25-52.
- Scandura, J.M. and Scandura, Alice B. Improving RAM in Large Software System Development and Maintenance. *Journal of Structural Learning and Intelligent Systems*, 1999, 13, 227-294.
- Scandura, J. M. Structural Learning Theory: Current Status and Recent Developments. *Instructional Science*, 2001, 29, 4, 311-336. (a)
- Scandura, J. M. Structural Learning Theory in the Year 2000. *Journal of Structural Learning and Intelligent Systems* (A Special Monograph), 2001, 4, 271-306. (b)
- Scandura, J.M. Structural (cognitive task) analysis: an integrated approach to software design and programming. *Journal of Structural Learning and Intelligent Systems*, 2001, 14, 4, 433-458. (c)
- Scandura, J. M. U.S. Patent No. 6,275,976. *Automated Methods for Building and Maintaining Software Based on Intuitive (Cognitive) and Efficient Methods for Verifying that Systems are Internally Consistent and Correct Relative to their Specifications*. August 14, 2001. (d)
- Sleeman, D. & Brown, J. S. (Eds.) *Intelligent tutoring systems*. New York: Academic Press, 1982.
- Sowa, J. F. (1984) *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley.
- Sowa, J. F. (1999) *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Pacific Grove: Brooks/Cole.
- Stume, G. (ed.) (1998) *Distributive concept exploration - a knowledge acquisition tool in formal concept analysis*. LNAI, Vol. 1504, Berlin: Springer.
- Shute, V. & Torreano, L.A. Formative Evaluation of An Automated Knowledge Elicitation and Organization Tool. In T. Murray, S. Blessing, & S. Ainsworth (Eds.), *Authoring Tools for Advanced Technology Learning Environments: Toward Cost-Effective Adaptive, Interactive, and Intelligent Educational Software*. Kluwer. In Press.
- Warren, J. R. The interaction of user and task modules in intelligent tutoring systems, 2002. ([www.acrc.unisa.edu.au/is/its/iutm/](http://www.acrc.unisa.edu.au/is/its/iutm/))
- Wille, R. (1992) Concept lattices and conceptual knowledge systems. *Computers and Mathematics with Applications* 23, 493--522.
- Wulfeck, W. H. & Scandura, J. M. Chapter 14 in J. M. Scandura. *Human Problem Solving*. Academic Press, 1977.