This article is Part I in a two part series introducing AuthorIT, an integrated, extensible authoring system for building intelligent (adaptive & configurable) tutoring systems (ITS). AuthorIT is based on an empirically tested theoretical foundation that is both deep and broad; it automates many processes in building ITS, including an elicitation tool, called AutoBuilder, for representing knowledge as Abstract Syntax Trees (ASTs). ASTs serve as input to AuthorIT’s companion, TutorIT. TutorIT delivers learning as prescribed—based exclusively on AST structure, interface and configuration options without reference to content semantics.

Section 1 presents an overview of AuthorIT’s major components: AutoBuilder, Blackboard Editor, Configuration Tool and TutorIT, illustrating their use with examples. AuthorIT has been used to develop prototype ITS involving procedural, declarative (structural) and model-based knowledge in arithmetic and basic mathematical processes. Programmers easily convert AST-based knowledge representations to executables. EZAuthor and TutorIT Customizer, respectively, are limited teacher friendly subsets that support development and customization with no programming required.

Section 2 describes the underlying theory. It details the close relationships between structural and procedural knowledge and shows how each can be
represented at arbitrary levels of detail in terms of a small finite number of refinement types. The section also describes the role learning objects, display, response and evaluation types play in ASTs and the Blackboard Editor, and the diagnostic and instructional logic enabling TutorIT to accommodate any content represented as ASTs.

Section 3 summarizes accomplishments to date, limitations of current work and planned extensions. Examples illustrate how ASTs in conjunction with AuthorIT have been used to develop tutorials not only for procedural knowledge, but also for declarative (structural) and model-based knowledge. Existing technology also is sufficient for supporting well-defined problem solving. Finally, current limitations are identified along with possible extensions showing how they may be addressed.

Part II in this series will detail theory supporting planned extension of AuthorIT to ill-defined problem solving, with emphasis on higher order knowledge, the role of structural analysis in identifying higher order knowledge and a universal control mechanism.

Keywords: Intelligent tutoring systems, authoring systems, structural learning theory, adaptive instruction, abstract syntax trees, knowledge engineering, computer based instruction, cognitive task analysis, structural analysis

1. INTRODUCTION AND OVERVIEW

Introduction: In principle, highly adaptive ITS may mimic or even exceed the behavior of many teachers. Despite decades of work in the area (e.g., Anderson, 1988; Scandura, 1987; Sleeman & Brown, 1982; Wulfeck & Scandura, 1977), however, developing intelligent tutoring systems and other adaptive educational software (ITS) remains expensive and labor intensive. The availability of low-cost “edutainment” and media-rich, but otherwise uninspired CBI authoring systems, make it hard for ITS to compete commercially, where development costs play an essential role.

Limitations in both technology and theory have made it difficult to create ITS authoring systems that are both feature rich and broadly applicable. Despite ongoing advances in processing power, software tools and web technologies, ITS development today is still expensive and requires high levels of expertise (cf. Psotka, 1988; duBoulay, 2002). Although progress has been made on a number of fronts, including automated knowledge elicitation tools (e.g., Shute et al, 1999) and authoring systems (e.g., Brusilovsky et al, 1997; Munro, 2003; Paquette, 2001) the fundamental problem remains of how best to specify tutorial logic independently of content (aka expert model).

Given these complexities, most authors have preferred the flexibility and
relative ease of use of modern media-rich authoring systems (e.g., Macromedia’s Flash & Authorware7). In these systems, content is represented as (often dynamic) “knowledge objects” with scripting languages used to code pedagogical logic. Associated research has focused largely on conceptual frameworks and design principles helping to guide authors in decision-making (e.g., Gagne, 1985; Dikjstra, 2001; Scandura, Durnin & Spector, 2001). Whereas modern authoring systems offer a broad range of powerful media, however, programming and testing instructional strategies can be extremely time consuming and error prone.

A number of ITS authoring systems have been developed in recent years to help realize the enormous potential of ITS (in an education industry estimated by NIST at $46B by 2005). One approach has been to develop authoring shells supporting different categories of content. Murray (2003), for example, summarizes and classifies features of over two dozen ITS authoring systems that show promise. These systems tend to be pedagogy or performance (e.g., simulation) oriented. The former category includes authoring systems like XAIDA and ID2, which offer different instructional strategies for different categories of learning. XAIDA, for example, was directly motivated by the taxonomy of learning categories detailed in Gagne’s Conditions of Learning (1985). Since then, a variety of shell-based authoring systems have been developed (e.g., Merrill’s ID2). Performance oriented systems focus on environments in which learners acquire knowledge through exploration with varying amounts of guidance and feedback (e.g., RIDES, Munro, 2003).

Murray (2003) has aptly described current ITS authoring systems as “bags of tricks” — sets of tools that facilitate the authoring process. Unlike traditional authoring systems (e.g., Scandura’s SURPAS in the late 70s’ & Macromedia’s Authorware7), which have been used to develop thousands of tutorial systems, few contemporary ITS authoring systems are commercially viable at present. Only three according to Murray et al (2003, personal communication) — REDEEM (Ainsworth et al, 2003), XAIDA (Halff et al, 2003) and RIDES — have been used to develop more than a handful of systems. Even here, there are limitations. Interface requirements, for example, limit the use of REDEEM to compatible content (Ainsworth at TICL 2004 Masters Conference in San Diego).

Knowledge representation (KR) has long been recognized as a major

---

*Hundreds of tutorials where developed with SURPAS in the early 1980s. Some are still being sold to schools even though they only run on the old Apple II computer.*
bottleneck in ITS development (e.g., Shute et al, 1999), especially KR suitable for supporting ill-defined problem solving. Systems that emphasize instructional strategies, either minimize the role of KR (e.g., REDEEM) or support different instructional strategies for various kinds of learning (e.g., ID2). Others, like RIDES, Demonstra8 and early stage authoring research at Carnegie Learning, rely heavily on deep infrastructure. Production systems and relational/conceptual networks, including variations such as (e.g., time sensitive) Petri nets and XML, are widely used for this purpose.

OVERVIEW

ASTs provide a third formalism that holds major promise. ASTs have long been used in software applications as diverse as compilers and software engineering design tools. Hierarchical databases are close cousins. ASTs also have replaced directed graphs as the preferred mode of representation in the deterministic, recently updated Structural Learning Theory (SLT) (Scandura, 2001a,b) on which AuthorIT is based. Except for AuthorIT, ASTs have not been used in serious ITS development. Production systems, relational networks, directed graphs and ASTs are all formally equivalent. Each, however, has different semantic properties making it more or less suitable as a foundation for knowledge representation (KR) in ITS. Table 1, first presented at the TICL Conference 2004, San Diego, April 16, 2004, summarizes the advantages and limitations of each.

The way knowledge is represented has a direct impact on ease or difficulty of ITS development: Relational networks represent knowledge intuitively, but they are not executable. Accordingly, they cannot generate correct responses to given problems. Production systems, on the other hand, are executable. However, essential relationships (e.g., order of execution) are “hidden” in the semantics of individual productions (in lists of productions). In short, familiar, deep structure KRs make it difficult, if not impossible, to separate KR from pedagogical strategies. Ideally, a KR should make it possible for experts to define any number of pedagogical strategies. Even better, a general-purpose, highly configurable tutoring system would enable authors to define pedagogical strategies without programming (with only media/learning object development remaining). Until then ITS development — especially deep-infrastructure, adaptive and/or configurable ITS development — will remain costly and heavily dependent on specialized expertise.
<table>
<thead>
<tr>
<th>Representation Type</th>
<th>Ease of Understanding</th>
<th>Automatic Response Generation</th>
<th>Content Representation</th>
<th>Learner Model</th>
<th>Control Mechanism</th>
<th>Tutor Decision Making</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Syntax Trees (AST)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Directed Graphs (DG)</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Production Systems (PS, P)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Relational Networks (RN)</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

**TABLE 1**

Advantages and Limitations of Alternative Modes of Knowledge Representation (N means > 100% Y)
The desirability of separating KR from pedagogy has long been recognized. Pedagogy oriented authoring systems often employ different pedagogical strategies for different kinds of knowledge (e.g., XAIDA). Munro (personal communication) has recently made significant progress in separating knowledge representation and pedagogy. In RIDES, each can be developed and revised independently of the other.

**AuthorIT** takes this separation an important step further: Use of ASTs to represent knowledge has made it possible to create a single, general-purpose tutor (TutorIT) that is configurable (adaptable) without programming. As summarized in Fig. 1, AuthorIT consists of:

- **AutoBuilder** - a knowledge elicitation and representation (KR) system, used to represent to-be-acquired knowledge, both structural (declarative) and procedural, hierarchically as ASTs (Figs. 2A & 2B). AutoBuilder automates many processes involved SA and is used to define knowledge as ASTs and to assign associated learning objects. ASTs make it possible to represent knowledge with whatever degree of precision is needed to meet defined educational goals. AutoBuilder also is used to assign learning objects for instruction, questions, positive and corrective feedback to individual AST nodes (Fig. 3A).

- **Blackboard (BB) Editor** (not to be confused with BB courseware in e-learning or BB architectures in computer science) - a Learner interface GUI modeler used to define problem interfaces through which the tutor and learner interact. BB Editor makes it possible to define problems in a wide variety of formats (including various display, response & evaluation types), to assign semantic attributes and to layout the learner interface (Fig. 3B).

- **Tutor Options Tool** (Fig. 4) - used to define tutor delivery modes for specified needs (e.g., adaptive, diagnostic), further customized to support alternative pedagogical strategies and learner populations.

- **TutorIT** - takes this information as input and delivers the content as prescribed. Options range from simple practice and simulation to highly optimized adaptive instruction with both tutor and learner control.

AuthorIT (e.g., U.S. Patent 6,275,976; Patent pending, April 2, 2003) was built with new software development systems, called SoftBuilder and AutoBuilder. Both are based on the AST-based High Level Design (HLD) language. SoftBuilder supports a ‘plug & play’ approach to distributed systems development and ensures component interoperability and
extensibility (i.e., the ability to add new components as needed). HLD includes a broad range of fully interoperable high and low level components, which are ideally suited for interpreting and manipulating ASTs (For more details see www.scandura.com/AuthorIT.htm).

AutoBuilder is an extension of SoftBuilder, based on U.S. Patent 6,275,976, which makes it possible to insure software consistency at multiple levels of abstraction. Designers familiar with structural (cognitive task) analysis create high-level designs and refine those designs arbitrarily until contact is made with executables in AutoBuilder’s core HLD library. Tutorials have been created to teach the necessary skills. AutoBuilder ensures that structural (declarative) and/or procedural knowledge is represented hierarchically as ASTs in a way that is internally consistent, captures essential semantics in an intuitive manner and is executable at all

*AutoBuilder is designed for professionals. A limited version, called EZauthor, eliminates the need for AutoBuilder programming and is easily used by teachers.
levels of abstraction. AutoBuilder built ASTs are like relational networks in the sense that the ASTs represent knowledge in an intuitive manner that reflects essential real world semantics. Unlike relational networks, however, and like production systems, they are fully executable. These dual qualities play an essential role in the research.

HLD components have a trivial syntax easily understood by programmers and analytically inclined mathematics and science educators. Operations and parameters may be arbitrarily refined.*

\[
\text{operation\_name (INPUTparam1, \ldots; IN-OUTparam1, \ldots; OUTparam1, \ldots)}
\]

The main prerequisite is familiarity with structural (cognitive task) analysis and the hierarchical structure of procedural (Flexform) ASTs (see Fig. 2A) and data (declarative) ASTs (see Fig. 2B).

AutoBuilder requires authors only to specify expertise directly associated with to-be-learned content. Authors systematically refine content step-by-step ensuring that the children in each refinement produce behavior that is “abstractly” equivalent to parent behavior (Scandura, 2003). Availability of needed components is assumed. Each node at each level of abstraction represents a distinct set of tasks and their solution. Each terminal element in an AST is atomic (see definition below) in so far as the population of learners is concerned (e.g., subtraction facts in the case of learning column subtraction). Terminals are “black boxes” whose internals are assumed to have no interest for instructional purposes. If they did, the author would simply continue the analysis. In short, the process continues as shown in Fig. 2A only until the content has been specified to the degree necessary to accurately reflect what is to be learned. Terminal nodes in AutoBuilder are assumed to be atomic (known/+ or unknown/-). Where desired, + and - can be distinguished further to accommodate misconceptions.

Once an AST has been constructed, authors can turn remaining implementation over to an HLD programmer. All HLD components (nodes) are fully interoperable, and can quickly be converted into executables. Nonetheless, learning to assemble preexisting components to produce desired results (i.e., programming) takes practice, even with an easy to learn high-level language like HLD. Implementation of terminal nodes in AutoBuilder can also be done via further refinement in procedural

*HLD programming eliminates the need for data types, etc. Unlike simple trees (e.g., XML), AST elements may have any number of parents.
FIGURE 2A
Flexforms representing two procedural ASTs and the AutoBuilder Tool bar showing available operations for manipulating same. The main Flexform shows operations in column subtraction at various levels of abstraction. For example, the highlighted node “Borrow_from_the_first_nonzero_borrow_column(…”) describes borrowing across zeros at a high level of abstraction, including its parameters detailed in Fig. 2B. Nodes below the highlight show greater detail. The Flexform insert shows a procedural Flexform for quadratic equations. The subtraction Flexform has more levels of abstraction showing that complexity of any content depends not just on the subject matter, but also on what can be assumed of the learner population.

FIGURE 2B
A tree hierarchy representing INPUT and OUTPUT data AST structures associated with column subtraction. The “prototype” node structure (subtree) CurrentColumn under Prob is shared by all columns (e.g., ones, tens, hundreds) in any given subtraction problem (see Fig. 3B).
FIGURE 3A
Instruction, questions, feedback, etc. (e.g., text, voice, video, Flash files) attached to nodes in procedural ASTs.

FIGURE 3B
Left panel in Blackboard Editor (BB) Editor, used to define individual problems. Center panel, to layout the interface. Right panel, to, assign attributes to individual nodes (elements) in the problem – latter includes display types (e.g., Text, Flash, Animation, Sound, Picture, OLE), response types (Edit Box, Click, Combo Box, Construction) and Evaluation types (Match_text, Within_region, Structure, Debug).
FIGURE 4
Tutor Options Tool (dialog) used by authors to define/configure alternative learning modes. Currently set to ADAPTIVE mode, one might also check DIAGNOSTIC, etc. (e.g., to meet state standards). TutorIT can be customized by teachers to use one mode only or to give the learner a choice. TutorIT delivers instruction accordingly.
Flexforms using SoftBuilder (a more general, less restrictive version of AutoBuilder). Nodes used to implement AutoBuilder terminals are displayed in green to distinguish them from blue AutoBuilder nodes referenced by TutorIT.

Fig. 3A shows how learning objects representing instruction, questions, positive and corrective feedback are attached to individual nodes in AutoBuilder Flexform ASTs. Each node represents a procedure at the corresponding level of abstraction.

**Blackboard Editor** (Fig. 3B) is a GUI used to define the learner interface. The hierarchy in the left panel shows the problem structure. The center panel represents the problem layout. The right panel is used to assign attributes to selected nodes in the problem. Whereas AST data structures have prototype columns, notice that actual problems have specific ones, tens, hundreds columns.

**Options Tool** (Fig. 4) in AuthorIT is used to configure delivery options. Authors are given wide latitude, and can define delivery modes ranging from full learner control (as constructivists prefer) to highly adaptive intelligent tutoring, along with variations thereof. Basic delivery modes include learner controlled, adaptive, instructional, diagnostic, practice and simulation. The author can fine-tune selected modes by choosing a range of options (e.g., making instruction on higher level nodes dependent on lower level performance). Assumed entry levels and alternative tutoring and diagnostic strategies also can be prescribed. Setting options requires understanding learning and instructional processes, but NO programming.

**TutorIT** takes this information as input and automatically delivers instruction as prescribed. AuthorIT and TutorIT both come with an integrated HLD simulator/interpreter. A visual interpreter/debugger is used by AuthorIT to test AST knowledge representations. TutorIT uses the interpreter to automatically generate solutions. Learner responses are evaluated using SLT’s deterministic model or otherwise (e.g., Bayesian, # items for mastery), and the Learner Model is updated accordingly (Fig. 5A). TutorIT makes all instructional decisions based entirely on AST knowledge structures, the current state of the Learner Model and author specified tutor options — all independent of content semantics. For example, if a learner demonstrates mastery of knowledge associated with a given node, the Options Tool may be configured so TutorIT will also mark as known the prerequisite, lower level nodes. Another setting in the Options Tool (to Learner Control mode) allows learners to select arbitrary nodes in the Learner Model (i.e., the knowledge representation AST in Fig. 5A) allowing
FIGURE 5A
Tree view representing the Learner model. Each node in the tree is marked with a symbol designating learner’s knowledge with respect to that node.

FIGURE 5B
TutorIT asking a question corresponding to highlighted non-terminal "borrow" node in Fig. 5A. The learner must answer all steps associated with this higher level node.
TutorIT screens illustrate place value & simple algebraic expressions. A. The top screen asks a question using Selection Objects for place value icons to facilitate learner (L) response. B. The middle screen illustrates instruction. C. The bottom screen asks the learner to create a formula machine using Selection Objects for digits (0, 1, 2, ...), letter variables (n, y & r) and arithmetic operations (-, +, x & / [division sign]). The Selection Objects in both Fig. 6A & 6C are used to facilitate answering the questions. In A, the learner must delete one of the tens in the tens column of the numerator and drag over the ones selection icon 10 times. Similarly, Selection Objects in C must be moved to the proper locations in the Formula machine. These examples also make clear that choice of media and hence physical appearance are strictly arbitrary.
learners to freely navigate through the content.

**Content:** Over the past year AuthorIT has been used successfully to build a sampling of procedural, structural (declarative) and model-based content in elementary and middle school mathematics. (Models include both structural and sets of interacting procedural knowledge in a domain, Scandura, 2003). Structures and procedures can be of arbitrary complexity, ranging from simple facts to building complex structures and/or complex problem solving. To date, AuthorIT has been used to build ITS systems for learning basic arithmetic skills, and their meaning. Individual tutors illustrate the role of place value in basic arithmetic operations and continue through fractions. Relationships between numerical operations and their concrete (actually iconic) embodiments (e.g., as Dienes blocks) are explicitly included and illustrate AuthorIT’s use in building domain-specific models. AuthorIT also has been used to build tutors for higher order, domain independent mathematical processes (cf. Scandura, 1971a): a) detecting regularities b) its opposite, constructing examples, c) constructing mathematical (iconic & symbolic) representations (e.g., Selection Objects in Fig. 6A show iconic representations of ones, tens and hundreds place values), d) interpreting representations, e) deduction and f) its opposite, axiomatization (i.e., identifying basic assumptions).

This work was accomplished under the author’s supervision by one part time undergraduate major in communications engineering with no prior experience in media and few artistic skills, much less the HLD language. It would appear, therefore, that AuthorIT holds considerable promise as a way to dramatically reduce the cost of authoring quality ITS, while making them easier to evaluate and use. Nonetheless, major questions remain as to whether AuthorIT provides capabilities needed to convert these tutorials into commercially viable systems. AuthorIT is advanced and quite usable but still in a formative stage. Demonstrating commercial feasibility is an exciting possibility.

It is hard to overestimate AuthorIT’s enormous potential. To date, AuthorIT has been used primarily with elementary and middle school mathematical content, although AutoBuilder itself has been used to represent a broad range of content, qua software designs. The approach, however, is completely general and assorted samples demonstrate broad applicability to essentially any content where the learner is able to use the key board and/or and mouse. Cost reductions make it commercially viable to build even specialized ITS at a fraction of normal costs. For researchers, an incidental benefit is being able to compare alternative learning and
instructional strategies without incurring prohibitive development costs. Simpler, limited versions of AuthorIT, called EZauthor and TutorIT Customizer, can easily be used by teachers. EZauthor is used to quickly construct lessons, where detailing procedural knowledge is not important. TutorIT Customizer is used to customize existing tutors built with AuthorIT (e.g., allowing teachers to add their own voice).

AuthorIT builds on two major advances. First, recent theoretical advances have been made in Structural Learning Theory (SLT) (Scandura, 2001a, b)*, most notably in the recognition that all levels of expertise for essentially all knowledge may be represented as ASTs. Theoretical foundations have been detailed in Scandura (2003), and are available at http://www.oldcitypublishing.com/TICL/TICL%201.1%20contents.htm**. This work details essentials of knowledge representation (as ASTs) and shows how any idea (knowledge) may be arbitrarily refined. This realization makes it possible to represent knowledge at whatever level of detail may be desired and/or relevant for instruction with any given population of learners. It provides a deep foundation for ITS systems.

Second, technical advances in software engineering provide the necessary software infrastructure (see descriptions of SoftBuilder, Flexsys and AutoBuilder at www.scandura.com ). In particular, the High Level Design (HLD) language is both based on and ideally suited for manipulating ASTs. HLD is easy to understand, learn and use and has a very simple highly familiar syntax,

\[
\text{operation (input\_parameter1, \ldots: input\_output\_parameter1; output\_parameter1,\ldots)}
\]

* Scandura (2001a,b) provides an updated overview of the SLT. The only significant difference between these articles is that the latter includes a history of major developments leading to the theory in its current state. An on-line version of Scandura (2001b) is available at http://www.scandura.com/TICL_On\_1\_ine/disc/00000002.htm

**For the reader’s convenience, a few key characteristics of SLT follow: a) distinctions between lower and higher order knowledge (used to distinguish between domain specific and domain independent knowledge), b) the representation of knowledge at different levels of abstraction (distinguishing levels of expertise makes testing and instruction more efficient), c) a universal control mechanism (which plays a central role in problem solving, and is implementable in a way that is totally independent of higher as well as lower order knowledge), and d) assumed innate learner characteristics (processing capacity & speed). SLT was founded on basic research in cognition and problem solving (e.g., Scandura, 1971c, 1973, 1977). SLT covers similar ground from a cognitive perspective, but differs in detail from cognitive theories widely used in ITS (e.g., Newell & Simon, 1972; Anderson, 1988). Critically important is the preferred mode of knowledge representation: production systems (condition-action pairs) vs. ASTs (which are executable & support basic software engineering constructs - e.g., sequence, selection and iteration). Another major difference is SLT’s focus on behavior as observed by and influenced by an external observer -- as opposed to establishing connections with underlying brain mechanisms.
HLD’s ability to represent knowledge in a direct intuitive manner makes it ideal for designing, implementing and maintaining both content representations and tutorial logic. As shown in the next section, content is represented using AutoBuilder, which imposes constraints on refinement ensuring that all levels of abstraction represent equivalent knowledge. Tutorial logic in TutorIT is built with SoftBuilder, allowing programmers greater flexibility.

2. THEORETICAL FOUNDATIONS

As shown in Scandura (2001a,b) AuthorIT builds directly on Structural Learning Theory (SLT). This section details only those portions of SLT on which AuthorIT currently builds most heavily. Particular attention is given to: a) the process of Structural Analysis (SA), as implemented in AutoBuilder to support construction of ASTs, b) the role of learning objects play in ASTs, c) the role display, response and evaluation types play in the Blackboard Editor and d) the diagnostic and instructional logic enabling TutorIT to accommodate any content represented as ASTs. Planned extensions of AuthorIT to ill-defined problem solving are the topic of Part 2 in this series.

AutoBuilder and Abstract Syntax Trees (ASTs)

Individual rules in SLT initially were represented as directed graphs (formally equivalent to flow charts). As shown in Table 1, directed graphs provide a sound basis for representing individual behavior potential (e.g., via simple overlays). They represent only a single level of knowledge, however, and methods used to construct directed graphs left undue discretion to analysts (knowledge engineers).

Recently, Scandura (2001a,b, 2003) has shown that problems, behavior and knowledge can all be represented at multiple levels of abstraction in terms of Abstract Syntax Trees (ASTs). Specifically, any problem can be represented as input-output ASTs, \( A_i \rightarrow ?B \), where A and B are ASTs in which in which individual nodes (variables) have specific meanings. A precisely defines an allowable set of input structures. B defines structure to be generated. The subscript in \( A_i \) indicates that input nodes in A have been initialized (i.e., have specific values). The ‘?’ before B indicates that the value of nodes in AST B are to be determined (e.g., 3-feet \(-?\)inches and 1+3+5 \(-?\)sum). Fig. 3B displays a sample subtraction problem constructed in AuthorIT’s Blackboard Editor.
Solution rules define knowledge, used to explain how humans perform. SLT rules consist of domain, range and procedural ASTs, and may be denoted \( A \rightarrow B \), where \( A \) is an AST representing the domain structure (defining inputs to which a rule can be applied), \( B \), the range structure (defining the structure of expected outputs) and \( \rightarrow \) the procedural AST (the process by which inputs are converted into outputs). The domain structure precisely defines the domain of applicability of the procedure. One solution rule for the second problem above is \( 1+3+5 \rightarrow 3^2 \) \( \text{sum} \). This solution rule applies only to the sequence 1+3+5. Figures 2A & B show examples of domains and procedures, respectively, as they are represented in the AutoBuilder component of AuthorIT.

Higher order rules are represented as higher order ASTs that include other AST-based rules as elements (i.e., nodes). For example, it is easy to envision a higher order rule that takes rules of the form \( 1+3+5 \rightarrow 3^2 \) \( \text{sum} \), as input and generalizes them to rules that apply to series involving any number of successive odd numbers beginning with 1 (i.e., to \( 1+3+\ldots+[2n-1] \)), where \( n \) is the number of terms in the series. A possible output of such a higher order rule is the more general solution rule \( 1+3+\ldots+[2n-1] \rightarrow n^2 \) \( \text{sum} \). The higher order procedure itself could range from simply replacing 3 terms (i.e., 3) with any number of terms (i.e., \( n \)) to specific steps in a formal derivation (e.g., proof by induction). One might further generalize such a proof to handle any arithmetic sequence of the form \( a+(a+d)+(a+2d)+(a+3d)\ldots \) In short, both problems in a given problem domain and the structural and procedural knowledge associated with that domain can be represented hierarchically in terms of ASTs. The only difference between lower or higher order problems or knowledge (e.g., meta-knowledge, solution strategies) resides in the nature of the nodes in the defining ASTs.

Arbitrary Refinement.— While the above notation is simple, ASTs themselves may be arbitrarily complex. A major requirement for instructional purposes is that ANY idea (data) or process can be represented at whatever level of detail may be necessary – too high a level and the information is too advanced for the learner, too detailed and instruction becomes too inefficient.

Top-down refinement of specifications and designs is not new – hierarchical representation has long played a role in knowledge engineering. Many relationships do not lend themselves to purely hierarchical analysis. Some ideas and design processes can only be represented in terms of potentially complex relations (e.g., Psotka et al, 1988), which themselves become even harder to decompose into more elementary elements. (An
analogous situation exists in software engineering, where there invariably is a significant gap between software designs and executable code.)

What is new is the realization that corresponding structures and processes can each be refined indefinitely in a uniform manner based on a small set of refinement types (see below). The key is to represent knowledge so that each level of abstraction is behaviorally equivalent to every other level (Scandura, U.S. Patent 6,275,976, 2001, 2003).

Relationships Between Declarative and Procedural Knowledge.—Arbitrary refinement of ASTs ensures that terminal levels in an AST can always be made atomic (i.e., sufficiently simple that they act in all-or-none fashion for learners in the given target population). While further refinement might be required for automation (i.e., to make them executable on a computer), the next section shows that further refinement is irrelevant from the perspective of explaining or predicting the (human) behavior.

Each node in a procedural AST defines a sub-hierarchy [subtree/subAST] of processes for generating corresponding behavior, with its own domain and range (AST) structures. Accordingly, each node represents a distinct level of expertise. Higher-level nodes in a procedural AST, where input-output structures are more complex, represent higher levels of expertise. In effect, there is a close complementary relationship between structural and procedural knowledge. AST-based rules (rules) represent arbitrary combinations of structural/declarative and procedural knowledge in a single integrated representation.

ASTs serve equally well with higher order domain specific (situated) and domain independent knowledge. The only difference is that domain and range ASTs associated with higher order rules include other AST-based rules. Domain ASTs rigorously specify domains of applicability. Key processes in SA have been formalized and patented (U.S. Patent 6,275,976, Scandura, 2001), and updated in Scandura (2003).

Learning can be guaranteed to the extent that atomicity assumptions are compatible with prerequisite capabilities of the population of learners in question. Another way of looking at atomicity is that terminal elements do not differentiate internal processes used to generate responses. (Accordingly, distinguishing correct from incorrect behavior — e.g., learner preconceptions, Bransford et al, 2000 — necessarily requires attention to higher-level analysis — AST structures defined by higher level nodes.)

Analysis does require time and effort. It is important to recognize, however, that analysis may be stopped at any point (prior to atomicity). Accordingly, in those cases learning will be less than optimal. Testing will
be less precise (e.g., probabilistic in nature and/or require multiple test items), and instruction will be closer to the norm—proportionately less than the guaranteed learning that is theoretically possible (see the section on diagnostic and instructional logic below).

Declarative and procedural knowledge are commonly distinguished in instructional systems. In SLT, all knowledge AST-based rules include both. Declarative (or structural) knowledge refers to data ASTs on which procedural knowledge, also represented as ASTs, operate. ASTs may be simple or complex depending directly on complexity of the subject matter and inversely on the sophistication of the intended population of learners. As we shall see, AST structures provide a sound and sufficient basis for specifying tutorial logic completely independently of content semantics. Tutorial logic depends solely on the structure of the data and procedural AST hierarchies.

An Example—The Many Facets of Column Subtraction.—Ordinary column subtraction provides a convenient example of the relationship between structural and procedural knowledge—precisely because it is simple, well known and commonly mis-thought of as being purely procedural in nature. Figure 2B shows domain and range data structures representing multiple levels of Structural (Declarative) Knowledge in Column Subtraction (aka AST data structure). Notice that these AST data structures serve as INPUT and OUTPUT templates for any number of concrete problems. For example, the problem shown in Fig. 3B has three columns: hundreds, tens and ones. The bottom digits displayed are “3”, “2” and “5”, respectively. The gray squares represent OUTPUT (or GOAL) variables whose values are to be determined by the learner.

Every subtraction problem has some fixed number of columns. Because the number of columns may vary from problem to problem, CurrentColumn in Figure 2 serves as a prototype variable (indicated in the Fig. 2B template by the word “prototype” to the right of CurrentColumn’s parent Prob). Every idea or concept (data) can be represented as the top-level node in a data AST. A basic premise in SLT-based knowledge representation is that nodes can always be further refined. Two kinds of refinement have long been part of the lore: Being an element or component of a set (bed and carpet being elements in room; Top, Bottom being elements of CurrentColumn) and being a subset of a set (dirty and clean rooms; dogs and cats being subsets of animals, or Top >= Bottom and Top < Bottom being subsets of the set of Top and Bottom number pairs).

Similarly, any operation (action defined on input and output structures)
FIGURE 7
Successive levels of procedural refinement in Column Subtraction. The Flexform on the top shows the path at the highest two levels of abstraction, "subtract (problem)", and the REPEAT UNTIL loop at the next level. The Flexform in the middle shows the refinement of the body of the loop. The Flexform at the bottom shows all levels refinement, taken to the point where terminal operations represent atomic operations for all learners who have mastered the presumed prerequisites (e.g., basic subtraction facts, ability to compare numbers as to size, etc.)
can be refined into corresponding child operations. The three procedure
ASTs in Fig. 6 show corresponding procedural refinements (represented as
Flexforms, see www.scandura.com). Loops (e.g., REPEAT .. UNTIL)
correspond directly to prototype refinements. Just as there can be any
number of instances of a prototype, one can go through a loop any number
of times. Components in a Component refinement (e.g., room into bed and
carpet) typically involve different operations, which can be carried out in
parallel. Notice also that relationships (e.g., \( Top \geq Bottom \)) correspond to
IF. THEN selection refinements.

A Finite Number of Refinement Types.— Unfortunately, the above two
basic kinds of refinement are not sufficient. Some ideas cannot
be refined into either components or categories, or variants thereof (e.g., prototype
components). Non-Hierarchical (relational) refinement requires
relationships between more than one component (e.g., room being tastefully
arranged or not tastefully arranged involves a relationship between
elements; \( top > bottom \) is similar). Further refinement depends inherently
on semantics of the relationship involved. No single relationship (i.e.,
structure) or refinement type will accommodate all cases.

This fact imposes important limitations on how far any given idea can be
refined hierarchically, a fact that has important implications for both
knowledge engineering and top-down approaches to software engineering.
In knowledge engineering, refinement involves making contact with
prerequisites available to all learners in a given population. In software
engineering, this involves making a design executable.

Scandura (2003) has proposed a solution by reformulating the problem.
Instead of attempting to refine relations as such, one can always define (i.e.,
进一步 refine) a parent data node in terms of an operation with its own inputs
and outputs. That is, every relation can be redefined as a mathematically
equivalent function (or operation). The resulting operation, in turn, can be
refined as with any other operation. In effect, every idea (node) can be refined
indefinitely via a small finite number of refinement types.

Higher Order Rules in Column Subtraction.— Column subtraction is
instructive in this case also. The numerals on which subtraction operates might
appear to be inherently irreducible. Deeper analysis shows they can (be
refined). Numerals can themselves be further refined in terms of the operations
involved in their construction. As every elementary school teacher knows,
children do not automatically know how to write numerals. Indeed, they also
know intuitively that until children can write numerals with considerable ease,
any attempt at teaching arithmetic computations will be an exercise in futility.
Reading and writing numeral operations, respectively, serve as inputs and outputs to operations in the column subtraction (and other arithmetic) algorithm. Although the algorithms themselves remain unchanged, they effectively serve as higher order rules (operating on other rules). While superficially quite different, the requisite learning is of the same genre as transitions between Piaget’s stages of development (cf. Scandura & Scandura, 1980). In each case, young children must master certain skills before moving on. For example, being able to determine equivalence of mass, volume or quantity is prerequisite to achieving the stage of concrete operations. Only after learning relevant equivalences will a child come to recognize that certain properties remain invariant under corresponding operations (e.g., pouring liquid from one container to another or rearranging objects). Higher order rules are elaborated in Part 2 of this series.

*Relationships Between Structural and Procedural Refinements.*— A small finite number of refinement types also are sufficient for refining operations/processes. There is a close relationship between the various kinds of specification and design refinements. Component refinements correspond to parallel refinements in which the child operations may be executed independently of one another. Consider the operation `clean (bedroom)`, where `bedroom` is a component refinement, with the components, `bed` and `carpet`. In this case, child clean operations on `bed` and `carpet` (e.g., `make` and `vacuum`) can be carried out independently of one another. Hence, the corresponding process (design) refinement is a parallel refinement.

Loop refinements in procedures, where a child operation is repeated a specified number of times, correspond to Prototype refinements. In Selection (IF—THEN, CASE) refinements, the child operations perform equivalent operations on different data structures. More generally, Category refinements correspond to what Scandura (2001c) called abstract operations.

Abstract operations are the *behavioral equivalent* of Object Oriented design, and 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, 2001c). Accordingly, Abstract Operations can be refined into “affiliate” child operations on different structures (corresponding to polymorphism in OO programming). For example, `clean (room)` can be refined into `clean (kitchen)` and `clean (bedroom)`, each of which involves cleaning, although of a specialized type.

In interaction refinements one or more of the parameters may be operations. Defining (i.e., refining) data elements in terms of operations implies operations
on operations. Consider a simple dialog box that changes dynamically. In this case the parent is an operation for displaying the dynamically changing dialog box parameter. This parameter, in turn, can be refined into a child operation corresponding to what Microsoft calls a callback. Since the callback operates orthogonally (independently) of the parent operation (displaying the dialog), the operation refinement is called an Interaction refinement. Our familiar column subtraction example also is illustrative. The numerals correspond to the dialog box and the callback to writing the numerals. In both cases the child operation acts independently of the parent.

These relationships are summarized in Table 2. See Scandura (2003 and U.S. Patent 6,275,976) for a more complete analysis and details on the rationale showing sufficiency of these refinement types.

**TABLE 2**

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

**Domain of Applicability.**— The above examples are highly structured, which implies that one can pinpoint the domain to which procedures apply. AST structures make it possible to precisely define domain of applicability of the corresponding procedures, something that is critical in automation. While details are beyond the scope of this article, it is important to emphasize that AST structures and procedures in incomplete or inadequate analyses may be out of sync. Given a problem in the domain, the corresponding procedure need not necessarily generate a response that
satisfies the goal AST. As a simple example, suppose the domain consists of arithmetic series of the form \(a + (a+d) + (a+2d) + \cdots + (a+(n-1)d)\), where \(a\) is the first term, \(d\) is common increment and \(n\) is the number of terms, the procedure. The procedure, \(n^2\), generates the correct sum, but only for series of the form \(1+3+5+ \cdots + (2n-1)\), where \(a = 1\) and \(d = 2\).

As demonstrated in Part II of this series on problem solving, misapplication is particularly common with higher order knowledge. Polya (1962) used the term “heuristics” to informally convey this reality (cf. Scandura et al, 1974), which in more recent years has been relabeled “situativity”. Whatever term is used, AST structures make it possible to define domain of applicability with the precision necessary for automation.

**Blackboard Editor**

Learning objects, display, response and evaluation types all play a direct role in defining the learner interface (i.e., the Blackboard).

*Learning Objects.*— Learning objects are assigned directly to nodes in knowledge ASTs. As shown in Figure 3A, learning objects define the instruction, questions, positive and corrective feedback for each node. AuthorIT provides native support for basic text, simple geometric figures (e.g., rectangles, circles, etc.) and OLE objects. In conjunction with widely available utilities (e.g., Microsoft Sound Recorder, Macromedia Flash), AuthorIT also supports arbitrarily sophisticated audio, video, animations, etc. via simple file references. These learning objects are completely modular and can easily be changed by simply referencing new files.

*Display, Response and Evaluation Types.*— As shown in Figure 3B, display, response and evaluation types are assigned to nodes in problem ASTs. Again, display types include an extendable set of display types, including text, geometric figures, OLE, and Flash, audio, picture and animation files.

Response types include *Edit boxes* (text), *Clicks* (in specified regions), *Combo-Boxes* (choices) and *Construction*. The construction type is in many ways the most general and allows learners to construct arbitrary response ASTs. Because arbitrary constructions are largely an authoring function, the key to a workable solution here is to eliminate unnecessary complications (for learners) and/or to provide aids, which simplify responding. One way to do this in AuthorIT is to provide the learner with a predefined selection of objects. Another is to allow authors to predefine attributes that effect how a response is displayed (e.g., by pre-selecting node font, color and/or size), but which are not evaluated by TutorIT.
The author also has a choice of evaluation types to be used by TutorIT for each response type. These include Match-text, Within-region and Best-fit. Obviously, evaluation types are not uniformly meaningful. Thus, match text is irrelevant for click responses as is match text for clicking. Two types of evaluation types are being tested for constructed responses, structure and debug. The former compares internal structure, whereas the latter checks alternatives for acceptability. Debug is useful where one wants to provide differential feedback based on misconceptions.

**Diagnostic and Instructional Logic**

This section describes how TutorIT makes its diagnostic and instructional decisions. Fundamental in this decision-making are the Learner Model (LM) and various options chosen by the author and, within limits set by the author, the learner as well. All diagnostic and instructional decisions are based on these options together with the state of the LM.

*Learner Model (LM).—* The LM is defined as an overlay on the ASTs representing the knowledge to be acquired. In the preferred embodiment, each node in an AST has one of three states: known (+), unknown (-) and to-be-determined (?). To make this concrete assume we have a well-defined problem domain in which any given problem can be solved by a single rule (e.g., column subtraction problems solved by the method of equal additions). More accurately, any given problem may be solved by interpreting/executing the corresponding procedural AST on the given problem. In general, only some of the nodes will be traversed in the process. For example, a problem like “357-135” does not involve “borrowing” (aka “regrouping”). Corresponding nodes in traversing the rule AST will simply be ignored. Accordingly, if the learner gets this problem correct, the only nodes marked “+” in the procedural AST (for subtraction) will be those that do not involve borrowing (see the following explanation). Additional problems will be required to promote full mastery.

Moreover, any problem can be broken up into any number of sub-problems (e.g., borrowing, subtracting column, etc.). Each sub-problem corresponds to a unique node in the rule AST, specifically nodes in executable AST procedures. Each node in an AST is the root of and defines a unique sub-tree — that sub-tree whose execution solves the sub-problem. Each such node also defines a corresponding data structure (recall the close relationship between structural and procedural knowledge). Accordingly, these relationships make it possible to automatically generate sub-problems precisely when and as they are needed. The reader may wish to verify these
relationships with the procedure in Fig. 5A.

An AST-based rule containing at most 20 or so procedural nodes appears sufficient for representing most (individual) procedures taught in most schools – at a level of detail sufficient to ensure that terminal nodes are atomic for the target learner population. “Atomic” in this context means that the corresponding knowledge is so elementary that it can be assumed available to all learners in the target population. One might expect at a minimum to have to treat all 20 or so nodes (and corresponding sub-problems) individually.

Because ASTs represent the same content at multiple levels of abstraction, however, the actual number required for diagnostic and instructional purposes is almost always considerably less. Success on a sub-problem corresponding to a non-terminal node (e.g., borrowing), for example, necessarily implies success on all simpler sub-problems. “Knowing” (i.e., marking ‘+’) a non-terminal node means by definition that all lower level nodes in the AST also must be +. Conversely, all nodes higher in an AST than a node marked ‘-’ also must be marked ‘-’. Other than by chance (mutually compensating errors), it is impossible (using the same rule) to get a harder problem correct unless one can solve all of the simpler sub-problems of which it is composed. These and related ideas dramatically improve tutor efficiency (patent pending).

Accordingly, the LM for a given rule is constructed by assigning “+”, “-” or “?” to each node in the corresponding (e.g., procedural) AST — insuring in the process that nodes in each sub-tree defined by each non-terminal “+” node necessarily are also marked “+” and that all nodes higher than each node marked “-” in the AST necessarily are also marked “-”. All diagnostic and instructional decisions depend directly on the current state of the ASTs defining the learner’s knowledge.

Reconciling the above with other approaches to knowledge representation requires explanation. Relational networks (see Table 1) used to represent school arithmetic (e.g., ALEKS) involve literally hundreds, even thousands, of nodes. One reason for this large discrepancy is obvious. ALEKS-type representations refer to large integrated relational networks of nodes (see Table 1), involving connections not only within specific skills (e.g., facts, subtraction and addition algorithm, etc.) but also between such skills. The above node estimates (e.g. 20) refer to individual procedures as commonly taught. A second reason is subtle, but even more important. The number of nodes in ALEKS-type networks goes up with the square of the number of nodes associated with the various skills (to show relationships
between the skills). This dramatic increase arises precisely because ALEKS requires an indefinitely large number of different kinds of relationships.

This is precisely the problem that is solved by SLT-motivated AST rules — where arbitrary refinement is accomplished via a small finite number of refinement types. Each rule associated with a problem domain, no matter how large or complex, is strictly modular. Relationships between rule “modules” are represented by equally modular higher order rules. Hence, the number of nodes only goes up only linearly with the number of rules. Nothing is lost in the process. Indeed, the introduction of higher order rules actually reduces the number of rules required by eliminating redundant rules. Furthermore, learning higher order rules has been shown to provide a sufficient basis for solving novel problems not explicitly considered when constructing a knowledge representation (e.g., Scandura, 1971, 1973, 1977; Scandura & Scandura, 1980; cf. Polya, 1962 & Scandura et al, 1974). The role of higher order rules and associated theory will be detailed in Part II of this series.

**Diagnosis.**— Given a problem, TutorIT selects nodes in the corresponding AST in accordance with options set by the author (see Fig. 4 and below). Given a sub-problem corresponding to the selected node in the procedural AST, the purpose of diagnosis is to determine the current state of the learner’s knowledge. Hence, nodes that are marked “+” or “−” are ignored for this purpose. Actually, this is not entirely true in TutorIT, where selected nodes may be singled out for higher levels of expertise (e.g., where the author wants the learner to know the material extremely well). These so-called “Automation” nodes are tested further even after they are marked “+” (e.g., to insure that the learner can respond correctly within predefined time limits).

Aside from Automation, diagnosis concentrates on nodes, marked “?”, whose correct state is yet to be determined. Corresponding sub-problems are presented to the learner, the learner responds and TutorIT evaluates that response.

TutorIT supports an extensible number of response and evaluation types in this regard. Matching text, including wildcards, in Edit and Combo boxes, and clicking in pre-set regions are directly supported, as they are in most ITS systems. In addition, a fully adequate tutoring system should allow learners to construct arbitrary ASTs, representing complex structures. While authors currently have this ability, learner support in TutorIT is limited in this regard. While learners theoretically can construct arbitrary data ASTs, for example, this is not necessarily easy to do, especially where learners are to construct procedural ASTs, involving operations. Similarly,
evaluation is limited to experimental work with: a) matching ideal against learner-constructed AST structures (where structural inferences may be made based on levels in data ASTs) and b) determining the best fit between responses and author-determined alternatives (used to identify learner misconceptions).

When ASTs are refined to the atomic level, empirical research demonstrates that a single test item for each operation node tested is sufficient (e.g., Scandura, 1971, 1973, 1977).* In the case of conditions, however, all of which are necessarily terminal in SLT-based ASTs, one test item is required for each distinction defined by the condition. If..then and loop conditions, for example, distinguish between true and false. Case conditions can have any finite number of distinctions.

Instruction.—Ideally, one only wants to provide instruction when and where it is needed. Providing instruction when it is not needed both wastes time and can reduce learner interest. Providing information (e.g., hints) before a learner has mastered necessary prerequisites, on the other hand, can lead to learner frustration and giving up (e.g., Scandura et al, 1969).

Instruction should be provided only when the tutor knows that the learner does NOT know the corresponding material. Accordingly, TutorIT provides instruction only on nodes designated by “-“.

Once instruction corresponding to a “-“ node has been presented to the learner, that node is marked “?” because the tutor cannot know for sure whether or not the learner has mastered the material. This can only be assured after subsequent testing.

Uncertainty—The above analysis is deterministic in nature (cf. Scandura, 1971) and based on the fundamental assumption that defining ASTs are refined to the atomic level. Clearly, there are any number of reasons why not all terminals will be atomic, not the least of which is the time and other resources an author wishes to devote to Structural Analysis (SA). Preliminary analysis (i.e., SA) should be viewed as an interim step in an on-going process, a process that may be picked up and extended at any later time. Only to the extent that this notion of incremental (and hence increasingly better defined instruction) becomes part of the working lore of instructional designers, will instruction realize its scientific potential — of increasingly precise understanding, predication and control of student learning.

* Under laboratory conditions, test-retest reliabilities were 96%, with a 95% confidence interval between 93-99% (Scandura, 1977, Chapter 8). Coefficients of Generalizability (Rajaratnam, Cronbach & Gleser, 1965) based on careful a priori analysis of column subtraction ranged from .71 to .87 (Scandura, 1977, Chapter 9, with John H. Durnin).
Whenever SA falls short of this ideal, neither testing nor instruction can be all-or-none. Just because a learner gets one sub-problem correct, for example, doesn’t imply success on other sub-problems associated with the same node. What to do in this case? One alternative might be to increase the number of states. There is nothing magical about just +, -, and ?. TutorIT itself currently includes Automation as a distinct level of expertise. Indeed, identifying reasons why a learner makes the mistake he or she does has long been an integral part of ITS research (Sleeman & Brown, 1982). This option is inherent in the above discussion of “best fit” evaluation. Another approach might be to introduce probabilities (e.g., assign probabilities to nodes, or to decision making itself, Bayesian being one of the more obvious choices).

Although both options are quite reasonable, subsequent research has not convinced many experts that identifying sources of mistakes is an essential factor in learning (Foshay, personal communication). Similarly, introducing probabilities holds promise; however, decision-making and making a general-purpose tutor that works with ALL content becomes correspondingly more difficult. Accordingly, TutorIT takes a different route, which accomplishes similar goals. One variant would require each node to be tested (on different sub-problems) some finite number of times (greater than one). Because diagnosis as detailed above is so efficient, however, another solution is to simply require the entire diagnostic-tutorial process to take place some finite number of times. Re-confirming diagnosis and remediation is similar to what engineers do in adding a safety factor when over designing a bridge.

Because of the strong interdependencies among nodes in an AST, the number of nodes calling for instruction will decrease significantly with each iteration. Early testing at higher levels in an AST will have the same effect with advanced learners. In general, testing the equivalent of, say, 5 times on an AST as a whole will be far more efficient than it would be if each node were to be tested individually that number of times. A study comparing repeated adaptive instruction with alternative approaches, where uncertainty explicitly enters the picture, would be of considerable interest.

Extension to Collaborative Learning.— Learners can and do learn from one another (as well as learning to cooperate). And, there is significant research on the potential benefits of collaborative learning in ITS (e.g., Arroyo et al, 2004; Mirzarezaeae, 2004). Although convincing evidence is lacking, some investigators might even argue that learners get more from each other than from some omnipotent tutor. Like most similar research in
education, however, the ultimate answer is likely to be “sometimes” — under typically hard to define conditions.

In either case, one obvious way to extend the above model to collaborative learning is to have multiple learner models, in addition to the automated tutor, and to allow for cross-over effects. For example, one learner getting the right answer in a shared environment is much like the tutor presenting instruction to the others. Accordingly, other learners would be assigned unknown “?” status. Just as when the tutor presents instruction one cannot know for sure whether or not the others actually learned. Diagnosis and instruction by an automated tutor could either be eliminated, or could proceed as before. In either case, learners who get an item correct (+) may relieved of having to answer – alternatively, their results might be put in the category of bridge-building “insurance”.

Tutor Options.— As shown in Fig. 4, AuthorIT authors are given considerable discretion in deciding how AST-based content is to be delivered to learners. The author chooses to include or not to include any number of variations on Adaptive, Diagnostic, Instructional, Simulation, Practice and Learner Controlled modes. The above description obviously emphasizes Adaptive mode, with Diagnostic and Instructional modes being special cases thereof. Simulation simply takes the learner through given problems step by step. We say more about this in our conclusions. Practice is simply practice. Learner Control mode allows the learner to select problems and nodes in AST-based rules — along with whether to be tested or to receive instruction.

Each of these modes has certain defaults (options in the Options Tools – see Fig. 4). Certain options do not make sense in some modes, and are grayed out. Among the more interesting options pertain to strategies used by the tutor to sequence nodes and/or to require or ignore prerequisites in deciding on testing and/or instruction. Given any problem, TutorIT begins by choosing a node that can be tested by that problem (not all AST nodes can be tested by any given problem). On what basis are these selections to be made? Perhaps the most natural order is to select nodes in the order they are executed in solving the given problem. This results in sequential instruction (and/or testing) as one might in a classroom. Other reasonable alternatives are to favor selection from the top-down, when the learners already know fair amount about the content, or bottom-up, where the reverse is true.

Other important choices pertain to dependency on prerequisites. For example, should one teach an unknown (‘-’) node before all of its
prerequisites are mastered, or not? The answer depends on whether or not the author believes learners are able to take bigger leaps, and consequently to learn more efficiently. Similarly, should a learner be tested on a node irrespective of prerequisite status. A conservative strategy might require that the learner master prerequisites before being given problems that require them. At present, these options are defined by the author, or determined by the Learner, but it is not hard to envision other strategies that could accomplish similar decision making automatically, based on prior learning history.

However defined, TutorIT gives learners a choice of those modes defined by the author — that is, if the author chooses to provide such a choice.

3. STATUS AND FUTURES

Accomplishments, Limitations and Planned Research

While routinely used in such well-developed fields as compiler theory and software engineering, the recent introduction of ASTs in SLT (Scandura, 2001a,b) offers major advantages over the directed graphs originally used in SLT. In addition to supporting SLT’s traditional benefits (universal control and processing constraints, individual difference measurement), ASTs represent ALL levels of knowledge simultaneously. Specifically, ASTs provide a precise, explicit and general way to close the gap between high-level theory and executable software (Scandura, 2003). ASTs provide:

• a uniform way to represent declarative/structural knowledge (including domain dependence) as well as procedural knowledge — like productions, directed graphs had domains, but there was no simple and heuristically pleasing way to represent data structures — surely data types were not it.

• a simple, uniform and natural way to represent differences between neophyte and expert knowledge. Neophyte knowledge is largely procedural; expert knowledge is more structural — with ASTs representing all gradations in between. The behavioral equivalence between different levels of abstraction in ASTs provides a direct basis for making inferences about learner models.

• An integrated representation of model-based knowledge, including both declarative knowledge about the system and inferential knowledge involving various components of the model (cf., Scandura, 2003).
a uniform AST-based way to represent domain dependent and domain independent knowledge. Naïve problem solving requires higher order knowledge, which is also represented as ASTs (which contain other ASTs in their domains and ranges). Domains of applicability of procedural ASTs, both lower and higher order, are represented as structural ASTs. Structural ASTs easily accommodate all manner of situated variations.

As detailed above (also see U.S. Patent 6,275,976; Scandura, 2003), a systematic method, called structural [cognitive task] analysis, has been developed for creating AST representations. Using ASTs to represent knowledge simultaneously at all levels of abstraction provides the key. Structural (Cognitive Task) Analysis (SA), as defined in Scandura (2003), provides an explicit basis for identifying ASTs in which each level represents equivalent knowledge at different levels of abstraction (cf. single level directed graphs/flow charts used earlier, Scandura, 1971c, 1977, 1984; Scandura, Durnin & Wulfeck, 1974). Different levels of abstraction directly reflect degrees of expertise. TutorIT explicitly attends to these levels in directing instruction. Expert knowledge of column subtraction, for example, might be represented by input/output structures associated with subtraction problems and the procedural knowledge by “subtract”. The expert performs tasks quickly, and TutorIT accordingly does not attend to details of the process the expert uses. Neophyte knowledge involves more detailed specification of processes operating on more elementary data (e.g., digits making up the problem).

Given an AST-based representation of content, TutorIT is able to make all instructional decisions based entirely on the AST structure representing the content, without any custom instructional logic (associated with content semantics). Moreover, AuthorIT’s customization tool can be configured so TutorIT can deliver the instruction (or learning) in any number of ways, ranging from simulation to highly adaptive, including wide variations on self-directed learning, practice, diagnosis and instruction.

Developments to date demonstrate that AuthorIT can be used to develop highly adaptive ITS in essentially any subject matter area at greatly reduced cost. AuthorIT tutorials can easily be configured and/or customized with alternative learning objects to meet a variety of specific educational needs (e.g., state standards). Because Tutorials built with AuthorIT are so easy to configure, AuthorIT also provides a cost-effective basis for definitive research on the benefits of alternative instructional strategies. With well-understood highly structured subject matters, as in science, technology,
engineering and mathematics education, AuthorIT even makes it possible to create ITS that guarantee learning.

The current implementation of Structural Analysis (SA) in AuthorIT’s AutoBuilder component has only been used to construct individual ASTs. Nonetheless, coverage has been broad, and includes both lower and higher order knowledge, the latter involving basic mathematical processes (Scandura, 1971a). Prototype TutorIT tutorials have been built in a variety of areas. Some of these required significant SA using AutoBuilder. Others were simpler, requiring only design of the Blackboard interface. These tutorials range from experimental to near commercial quality, and include the following:

Procedural Tutors.—A wide range of TutorIT tutorials have been built in arithmetic, including facts, algorithms, fractions, decimals and Dienes’ block icons representing place value manipulations. All involve what is commonly thought of as procedural knowledge, but they differ both in the sophistication of the media and nature of the learner responses and evaluation methods required. Media range from simple natively supported text and simple graphics to audio, picture and reasonably sophisticated Flash files.

All media are strictly ‘plug and play’. Each media file is a modular learning object that can be independently developed and/or reused as desired. The common denominator is that these “learning objects” correspond to individual nodes in AST knowledge representations. Media associated with higher-level nodes in an AST tend to (actually should) be both more encompassing and less detailed. For example, instruction associated with higher level nodes provide overviews more suitable for advanced learners, whereas instruction associated with lower level nodes provide more detail.

These tutorials also differ in the kinds of responses the learner is to make and the way in which those answers are evaluated. In many cases, Edit-boxes and/or Combo-boxes are sufficient. Corresponding evaluation methods involve matching text, either exactly or within specified parameters (e.g., using wild cards to specify acceptable answers). Other problems are more naturally solved by clicking on specified screen locations. Each of the above has been used extensively in arithmetic, and has been sufficient for most purposes.

As commonly taught, however, a key step in performing decimal arithmetic involves “lining up” numbers and proper placement of the decimal point. Testing with the above response and evaluation types forces
one to reformulate natural processes. A better solution would be to have the learner directly insert or move a decimal point to the proper location.

Because all knowledge, problems and their solutions can be represented as ASTs, the most general solution would be to allow learners to construct their own (AST) responses. In this case evaluation would involve comparing ASTs constructed by learners with one or more ASTs representing either acceptable responses or different categories of misconceptions. Although not fully developed, AutoBuilder currently supports two kinds of evaluation – based on matching sub-hierarchies within structures and best overall fit allowing for debugging anticipated misconceptions. Further work is needed in this area.

These procedural tutorials can be ‘run’ in any number of modes, with TutorIT serving alternately as a learning environment under learner control, a diagnostic instrument, an instructor and/or a highly adaptive tutor tailoring its every action on what is known about the learner’s current state of knowledge. The ASTs defined using AutoBuilder provide an explicit basis for representing individual differences as AST overlays in Learner Models.

**Higher Order Skills.**— In early work leading to SA, Scandura (1971a) developed a taxonomy of basic mathematical processes. This taxonomy was motivated in part by earlier work by Dienes (e.g., Dienes & Jeeves, 1965) developing concrete models of mathematical ideas, and consisted of three inverse processes: a) Detecting regularities from examples, and the inverse of constructing examples of regularities, b) Understanding symbolic and iconic representations and the inverse of representing understandings symbolically and/or ironically and c) Deduction, making inferences based on given information, and axiomatization, identifying fundamental assumptions from which given information may be deduced.

This work motivated the development of a series of tutorials. Each kind of process involves the generation of new rules, and hence by definition is of a higher order. For example, one regularity in the series 1, 3, 5, 8, … may be described as “next number is the sum of the previous two”. This regularity allows one to extend the series indefinitely (i.e., 13, 21, etc.). There is any number of higher order rules for deriving such regularities (i.e., rules). One such higher order rule, for example, might simply look at successive pairs of elements in the given sequence to see if the next is the sum. Obviously, this would be a very restricted higher order rule of very limited value — it is essentially equivalent to the regularity (i.e., lower order rule) itself. A more general higher order rule might try arbitrary arithmetic operations (e.g.,+, -, x, /) with successive pairs. Such a higher order rule would work with a series like
1, 3, -2, 1 (where subtraction may be used). A still more general higher order rule would look for similar relationships between (single) successive terms, pairs of terms, triples, etc. Such a higher order rule would be sufficient, for example, for deriving regularities for arithmetic series, involving a common difference, and algebraic series with a common multiplier.

None of these higher order rules, however, comes close to universality; they are intrinsically context dependent. Thus, a higher order rule, which allows a knower to derive (some) regularities in number series, may be utterly useless in other contexts. Accordingly, whereas SA allows the theoretical possibility of guaranteed learning, practical limitations often preclude the level of analysis (i.e., SA) required. Nonetheless, it is easy to envision higher order rules of this type that extend quite broadly. Certainly, experts in most areas develop such higher order skills, some to a high level indeed.

Given the documented importance of so-called “situated” learning, it should be emphasized that ALL ASTs have domain and range structures. Domain structures explicitly define (situated) range of applicability, and correspond to automated, perceptual knowledge (cf. Bransford et al, 2000).

**Experimental Tasks.**—To explore AuthorIT’s limits, a sampling of other tutorials also have been built. Two of these requires learners to construct responses – specifically to build a tower from blocks. In one case, learner responses are evaluated against an ideal. In the other it is compared for best fit (see above).

Although problem solving procedures may be arbitrarily complex, TutorIT currently does not support ill-defined problem solving, requiring unanticipated solution methods. TutorIT does, however, currently support model-based learning involving a combination of declarative and procedural knowledge together with a limited form of inference – the equivalent of forward chaining. A simple railroad crossing tutorial, involving a train, a signal and a gate was constructed for this purpose. The learner is required to distinguish crossings that are working and not working, as well as to predict the behavior of the crossing. The TutorIT is only given rules for determining the color of the signal when the train is in the crossing as opposed to when it is not and for how the gate is controlled by the signal. Nonetheless, the built-in equivalent of forward chaining allows TutorIT to automatically determine how the gate behaves given (only) the position of the train. Except for the parallel declarative abilities, this type of inferencing is a simple case of what ITS systems have long been able to do.

**More Elaborate Learning Objects.**—A series of tutorials also has been built to help authors learn to use AuthorIT, and its simpler cousins EZauthor
and TutorIT Customizer. Because few outside our shop are familiar with AuthorIT, these tutorials have been configured to make only instruction and learner control modes available to learners. The main distinguishing characteristic of these tutorials that they involve relatively sophisticated Flash learning objects – again, with the learning objects assigned to individual AST nodes.

**Well-Defined Problem Solving.**— As noted above, AuthorIT can be used to build tutors for well-defined problem solving where solution procedures, no matter how simple or complex, can reasonably be identified. Problem solving is currently accommodated in TutorIT both via (one or more) ASTs representing problem solving procedures and input-output structures at multiple levels of abstraction representing various levels of expertise. Although the work is yet to be done, and without minimizing the effort that would be required, analyzing domains, like verbal problem solving in algebra, would seem to pose no new known problems and should yield to the basic approach. Verbal problems, for example, are often broken down into sub-problems for instructional purposes, Accordingly, the first stage in SA typically involves a sequence refinement: Represent the problem as a set of algebraic equations and then solve the equations. Exploratory work with AuthorIT in this area has been very limited.

To summarize: Although various components can be made more robust and further improved cosmetically, AuthorIT is quite adequate for developing a wide range of highly adaptive, customizable and practical ITS systems. Prototypes developed to date further demonstrate that such development can be accomplished by reasonably trained personnel at a fraction of traditional costs.

**Current Limitations and Planned Research.**— Like any major new system, AuthorIT has not yet achieved its full potential. Planned but still to be realized goals include:

1. allowing learners to create their own problems,
2. extending support to include more powerful response and evaluation types,
3. applying SA to a broader range of more complex knowledge domains, with particular attention to identifying higher order knowledge playing a critical role in ill-defined problem solving,
4. adding support for ill-defined problem solving – the latter requiring implementation and refinement of a patented universal control mechanism and
5. adding support for collaborative learning.
Without minimizing the work required, the former two goals appear routine. Authors and learners, for example, use the same Blackboard machinery — with authors simply having more flexibility (e.g., to construct problems). Making relevant capabilities available to learners will be largely a matter of deciding the degree of flexibility to provide, and how to make the task easier for naïve learners. Similarly, adding new response and evaluation types will be a largely modular exercise and well within the limits of available technology.

The third goal could take more effort. The method of SA has been used manually to identify higher as well as lower order knowledge associated with problem solving in arbitrarily complex domains, including ill-defined domains (e.g., Scandura, Durnin & Wulfeck, 1974; Durnin & Scandura, 1977, Scandura, 1984). The challenge is to extend such analyses, constructing higher order ASTs and reducing terminals to executables. This is certainly possible. Its accomplishment, however, will require dedicating the necessary financial and technical resources. Major decisions will involve selecting sub-domains for analysis and deciding how far to refine individual ASTs. These and related issues in SA will be detailed in Part II of this series.

AutoBuilder makes it possible to refine ASTs indefinitely. For both theoretical and practical reasons, however, one reaches a point – usually sooner than later – beyond which further analysis serves no valid instructional purpose. AuthorIT takes this into account explicitly. Decision making in TutorIT ignores structural information in an AST beyond the point of atomicity, or sooner if one is willing to rely on probabilistic measures. Subject matter experts and/or instructional designers need only refine ASTs to the point of atomicity. Once terminals are judged to be atomic for the intended learner population, programmers take over and complete the implementation. Making terminals executable is frequently a trivial programming task using the HLD language. In the worst case, programmers may need to introduce new HLD components (e.g., written in C/C++), a task which is fully supported in Scandura’s SoftBuilder system (www.scandura.com).

The fourth goal also will be challenging. SLT’s universal control mechanism, for example, must accommodate everything available in expert systems, and then some. What serve as (fixed) control and conflict resolution mechanisms in production systems are completely modular higher order ASTs in the SLT. Assuming universal control can be implemented in a completely general fashion (an explicit mechanism has
been patented but not thoroughly tested), available theory provides an explicit foundation for adding support for ill-defined problem solving. Higher order ASTs make it possible to construct (new) procedures for solving novel problems, procedures not anticipated by the author during analysis. Like any other AST, higher order ASTs can be replaced at will without effecting underlying mechanisms.

The fifth goal, representing knowledge associated with multiple learners (as needed in collaborative learning) also presents challenges (albeit to a lesser degree). What the tutor knows about the learner (model) is currently represented as an overlay on nodes in each AST – where ‘+’, ‘-‘ and ‘?’ represent, respectively, what the learner knows, what he does not know and what is still to be determined. In addition to the AST structure itself, TutorIT uses knowledge states associated with AST nodes to make inferences as to knowledge associated with other knowledge states.

While SLT has only been applied to individuals, the basic theory appears equally applicable to collaborative learning. Extending TutorIT to support collaborating learners will require representing the knowledge associated with multiple learners, and specifying inferences between the various learner models as learning progresses. Representing such knowledge might be accomplished, for example, by introducing arrays or linked lists to represent the knowledge associated with multiple learners. Implementing inferences will require introducing logic specifying how the performance of individual learners effects what the tutor knows or can infer about the others.

For example, one learner solving a problem (that may stump others) serves as instruction effecting inferences TutorIT is willing to make about other learners. For example, one person in a group solving a problem does not imply that others have actually learned as a result. Accordingly, the corresponding state for other learners might reasonably be set to undetermined (i.e., ‘?’). Conceptual issues, of course, are only one part of the equation. Technical issues pertaining to distribution across the web will also need to be considered where learners are scattered at different locations.

Connections to Other Research.— SLT and AuthorIT provide a broad deep-structure foundation, not only for ITS development but for integrating results associated with a variety of research perspectives in teaching and learning, ranging from high level conceptualizations to executable systems. This work does not negate the value of other theories of teaching and learning or technical approaches to ITS development however. High level frameworks and taxonomies, for example, have traditionally been easier to
put into practice, and simulations used to promote problem solving skills have already been used to good effect. What the above shows is that a variety of open theoretical issues may yield to deeper analysis. In this respect, SLT shares some things in common with other deep-structure cognitive theories, such as the SOAR and ACT projects led by Alan Newell and John Anderson, respectively. Unlike these long-term efforts, however, SLT deals exclusively with observable behavior, and makes no attempt to model underlying brain physiology. (It does, however, postulate theoretical constructs, such as universal control and fixed processing constraints, which may turn out to have an identifiable physiological basis.) Cognitive processes are strictly constructs that are operationally defined in terms of observable behavior. Knowledge is inherently relative to what an external agent (e.g., teacher) is observing. In this sense SLT and its realization in AuthorIT directly addresses a broad range of instructional issues, offering a deep, integrated foundation for computer-based learning whose widespread use holds significant promise.

KEY REFERENCES


OTHER REFERENCES


Dijkstra, S. Epistemology, Psychology of Learning and instructional Deign. Special Issue of Instructional Science, 2001, 29, 4-5


**RELEVANT PATENTS**


ADDENDUM

AuthorIT, TutorIT and eLearning Standards

The recent focus on open eLearning standards in CBI (e.g., SCORM, IMS Learning Design) raises questions as to their relevance and where AuthorIT fits. Independent personal communications with Ingo Dahn, Norm Friesen, Hans Hummel and Val Shute suggest that the answers could have a substantial pragmatic impact on AuthorIT’s use, both in the U.S. and overseas.

The goal of such standards is to make eLearning more broadly available and development more cost effective (e.g., via reuse). In the process, standards are supposed to be neutral as regards supporting theory and/or conceptual framework. In practice, the latter can be very difficult to accomplish, especially in a complex, value laden and technically immature area like education.

As Friesen (2005, in press) puts it “Pedagogy … can be said to have the pervasive but elusive character of ‘deeply rooted’ cultural beliefs of Ideology”. … The non-neutrality of e-learning standards is further heightened by the fact that standards in general have been shown to act in the service of practical and ideological positions and goals.” Based on the author’s earlier contact with similar standards groups in software engineering, he too is somewhat skeptical of the benefits of standards, especially in the softer sciences. Putting future developments in a premature straight jacket can impose inappropriate restrictions on future development. In software engineering, hand drawn system diagrams originally used by consultants have been extended in the so-called Unified Modeling Language (UML) to show connections with underlying code — resulting in what many believe is UML overkill. The two-decade long failed attempt in the U.S. to impose Ada on all software development in the military, not to mention the even more egregious I-CASE disaster, is illustrative. Imposing premature standards often results in unproductive schisms between believers and non-believers. Commercial dominance and government mandates, based on mediocre standards, is even worse. They can easily impose insurmountable barriers to true innovation.

The AuthorIT/TutorIT approach appears to be based on a fundamentally different concept. Similar to Adobe reader, TutorIT offers free universal distribution of fully customizable tutorial logic. It was considered but would be difficult, if not impractical, to duplicate TutorIT’s sophisticated logic on a website. Although TutorIT theoretically could run on a server, there is little
downside and the user is far better served by having TutorIT installed locally. AST-based content and supporting (Flash or other) media files are downloaded on demand.

The above issues should be considered carefully before adoption by eLearning standard bodies. This is especially important because some bodies (e.g., UNFOLD project, www.unfold-project.net) have apparently started developing tools (i.e., for an IMS Learning Design spec). Purportedly, the Learning Design spec only assumes that learning episodes are made up of scenes in which actors act in different roles using some resources. No specific restriction is made on what actors, roles or resources are. Dahn (personal communication) suggests that if AuthorIT could produce IMS Learning Design conformant courses, these might be used with Learning Design tools. Like most web-based work these days, IMS is based heavily on XML, which is similar in some respects to SLT-based ASTs. Both have a hierarchical structure and an arbitrary number of fields (e.g., data types, attached media files), although only in ASTs can nodes have multiple parents. Technically, it should be possible to convert one to the other. Indeed, Flexsys (www.scandura.com) already includes the ability to reverse engineer XML into AST-based Flexforms and to generate XML from Flexforms, although modifications would likely have to be made to support IMS standards.

The author has a mixed position on standards. On the one hand, there appears to be little harm, and a lot of potential good, in separating media from authoring and delivery — e.g., making it possible to simply reference and reuse a wide variety of file types (audio, avi, flash, etc.) to support learning and tutoring. On the other hand, he agrees with Friesen (e.g., 2005) that there are very real reasons to be concerned about imposing premature pedagogical standards.

Imposing eLearning standards on delivery (e.g., question formats, content schemas) could be especially detrimental. The potential use of schemas in IMS, for example, would imply different delivery modes for different kinds of knowledge — e.g., Gagne-derived instructional design categories of learning thinking from the 1960s. In addition to leaving open the question of what to do with content involving multiple types (like most real world knowledge), adopting such an approach would confirm the status quo.

As above, it is easy to envision converting any XML format into ASTs, or vice versa. (Flexsys was explicitly built for this kind of thing.) The question is what the XML DTDs actually represent — content or, as suggested above, a combination of content and delivery mode (instruction, feedback, question
formats, etc.)? Are LD standards, unlike LMS, completely neutral as to delivery mode, as suggested by Hummel, or do they presume hidden constraints, as argued by Friesen. Presumably, no harm would be done as long as future development is not forced to adopt unnecessarily and inappropriately restrictive delivery standards (e.g., making new delivery modes difficult to implement or, worse, forcing one to use different delivery modes for different kinds of learning).

Although one could, in principle, move between ASTs and XML in either direction, XML content to AST content would likely be the preferred route. A powerful easy to use delivery system that works with ASTs, TutorIT, is already in place. Nonetheless, there are at least two significant impediments in this direction: a) XML does not ensure consistency between levels as is the case with AutoBuilder. b) Both data and processes in the AST-based HLD language are represented in the same way. One might hook executables onto XML, but XML essentially is data. This implies important limitations in XML representations?

TutorIT finesses the problem of delivery standards by avoiding the need for custom delivery modes. Simply configure TutorIT with AuthorIT’s Options Tool (a dialog box) to deliver the content as desired. One can in principle attach ANY number of kinds of media to nodes in ASTs, which provides almost unlimited freedom.

Does this suggest ASTs as a preferred standard!? Tools already are largely in place. AuthorIT (including AutoBuilder) CAN be learned and used by subject matter experts, instructional designers, even teachers to a limited extent. Where some degree of programming experience comes in is in making tutor relevant AST designs (created with AuthorIT) executable. This is typically a very simple task for individuals with some programming experience — much simpler than learning to program in Java, for example, and probably on a par with advanced XML. Whatever the case, it will be interesting to see if and how AuthorIT might help facilitate implementation of truly open standards.

An even simpler, non-intrusive standard might be to simply require/encourage support for common file types (e.g., .wav, .swf, bmp, etc.). Doing so would allow broad reuse of media, one of the more time and creativity extensive aspects in CBI development. With AuthorIT, one can largely in practice already, and certainly in principle, define essentially ANY delivery mode for ANY given content. Like Flash reader, single-click-install TutorIT once and use it with any content. TutorIT makes it possible to “read” — and deliver as specified — any content whatsoever, as long as that
content is represented as ASTs. In turn, AST nodes reference an extensible assortment of media, and TutorIT automatically presents same as and when appropriate. Existing standards AND associated tools (e.g., Flash reader) already ensure the interoperability and availability of such media.

Capitalizing on such interoperability will require efficient means for identifying and modifying needed media. As in Object Oriented programming, the challenge will be to make it easier to find and adapt existing media than to develop such media from scratch.