Flowcharts as a Basis for Procedural Training and Task Support

Eric A. Domeshek Elias Holman James Ong John Mohammed

Stottler Henke Associates, Inc.
280 Broadway / 1st Fl
Arlington, MA 02474

Abstract

In the work reported here, we have set ourselves the goal of creating cost-effective high-quality automated on-line learning environments for procedural tasks. The result is an initial implementation of a system called the Flowchart Task Tutoring and Tracing Toolkit (FT^4). The procedure language developed for FT^4 is significantly richer than that used in most previous scenario-based procedural Intelligent Tutoring Systems (ITSs). On the other hand, it is less expressive than the plan languages used in previous model-based procedural ITSs. Our aim is to share some of the authorability advantages of scenario-based systems, while achieving much of the generative power of model-based systems. To date, we have designed and implemented the language with an ASCII encoding, built and packaged an interpreter that can interact with external simulations and prototyped a first application, and begun to explore the interaction of the interpreter with a student modeling component.

1. Goals and Definitions

In the work reported here, we have set ourselves the goal of creating cost-effective high-quality automated on-line learning environments for procedural tasks. The result is an initial implementation of a system called the Flowchart Task Tutoring and Tracing Toolkit (FT^4). We start by elaborating on what, exactly, our goal description is intended to mean.

By *procedural tasks*, we mean any process to be carried out (or supervised) by a person, where that process calls for a reasonably well-specified (though by no means rigidly fixed) sequence of operations. Examples might include:

- controlling complex technological systems (e.g. industrial plants, satellite clusters),
- dealing with narrow social or business interactions (e.g. staffing a help-desk or a check-out line),
- or even mastering the cognitive aspects of complex constellations of physical skills (e.g learning to perform cardiac life support techniques, or machine maintenance activities).

By *high-quality learning environments*, we mean settings that incorporate, and effectively interrelate adaptive instruction, practice, and coaching. In on-line environments, practice is typically provided by interaction with *simulations*. Automated instruction and coaching call for artificial intelligence (AI) components that typically go under the heading of *intelligent tutoring systems* (ITSs).

By *cost-effective*, we mean our goal is to lower the costs of fielding these systems. We believe that, at least for procedural tasks, raw technological capability is no longer the barrier to widespread adoption of effective advanced learning environments. We know how to build interesting and useful systems, but typically it takes too much time, expertise (and therefore money) to build, distribute, and support them. Such systems require a mix of domain, instructional, and computer-science expertise.

Thus, in summary, we are interested in building a particular class of simulation-based intelligent tutoring systems, and doing so in a way that makes these systems more affordable in both the short and long run. We believe this is the best way to make truly engaging and effective instruction available to the widest range of students. In addition, the architecture we propose extends neatly to provide on-line task support in those cases where the procedural task is naturally performed with computer-support.

2. Comparing Procedure-Oriented ITSs

There have been a number of systems developed to support intelligent tutoring on procedural tasks, taking both scenario-based and model-based approaches. Prior to the work described here, we at SHAI had developed a library module called the Task Tutor Toolkit (T^3) (Ong and Noneman, 2000) that emphasized authorability of relatively simple scenarios. Likewise, Guralnik (1996) describes an authoring tool, called MOPed-II, that applies a content theory of task knowledge to enable the tutoring system to generate replies to common procedure-oriented questions. The RIDES system (Munro and Pizzini, 1995) and its successors are probably the best-developed systems extant for authoring graphical procedure-oriented training simulations (frequently, of devices) integrated with the intelligent tutor; it too takes primarily a scenario-based approach. In contrast, we could cite more ambitious and elaborate model-based (and dialog-oriented) systems such as the BE&E tutor (Core, Moore, and Zinn, 2000), or the sequence of systems (i.e. TOTS, STEVE, PAT, and PACO) described by Rickel, et. al, (2000).

These systems vary on several dimensions:

- Type of task environment simulation and/or degree of integration with a particular simulator;
- Complexity of procedures that can be specified/trained;
- Relationship between procedure specifications and "problems" or "scenarios";
- Relationship between procedure specifications and student models;
- Attention to and support for authoring;
- Support for lessons/tasks other than procedure practice.

There are two major stances on the issue of how a procedural ITS package might relate to a simulation environment. Either the simulator is integrated into the ITS package as with RIDES, or the tutoring component is explicitly packaged as a module designed to be interfaced with a variety of external simulators as with T^3 and the TOTS et. al. line of work. The first approach allows for a highly integrated system that can fully support all aspects of ITS development. The second approach makes more sense in domains where simulators are either already extant, or are likely to be more complicated than (or inappropriate for) the simulation and authoring tools in the integrated environment. Our current work on FT^4 follows in the tradition of T^3 in being separate from any particular simulator.

The issue of procedure complexity is currently the strongest point differentiating FT^4 from prior work on procedure-oriented ITSs. Scenario-based ITSs such as RIDES and T^3 adopt something like a simple tree representation for procedures, where internal tree nodes can represent either ordered sequences of children or unordered collections of children, and external leaf nodes represent primitive actions. T^3 adds the capability of generalizing the parameters to primitive actions to ranges or sets rather than fixed values. FT^4 for its part, allows a much more general procedure representation that includes not only ordered and unordered sets of steps, but also (1) named sub-procedures with parameters and local variables to supplement global state; (2) conditional branches, and thus potentially loops; (3) exception conditions; (4) alternative steps; and (5) optional steps. Section 3 presents the full procedure language in detail. While a significant generalization over other scenario-based tutors, FT^4 *procedures* remains less expressive than the *plan* language used in most model-based tutors.

Procedure specifications in scenario-based systems are created as part of defining a particular lesson or problem. Each procedure specification essentially represents a particular scenario a student may be asked to work through. Supported by the more complex procedure language, FT^4 takes the stance that a procedure specification may represent a particular scenario, or a large chunk of the curriculum (from which many different scenarios might be generated). Thus FT^4 aims to bridge the gap between scenario-based and model-based procedural ITSs with a representation of intermediate complexity. The basic idea is that mastery of the full procedure requires demonstrated ability to execute all the right steps at the right time (essentially, to run the student through a set of scenarios that exhaustively cover all the branches and transitions in the procedure).

The current FT^4 prototype does not yet fully implement this approach, but many of the underlying facilities are in place. In particular the system maintains student models that reflect the transition structure of the encoded procedures. The system scores a student on three dimensions: (1) how often they

correctly execute an action indicating they followed a valid transition in the procedure graph; (2) how often they fail to execute an action that is required by a forced transition in the procedure graph; (3) how often they incorrectly execute an action that represents an incorrect transition out of a conditional test. This strongly procedure-centered view of student model is significantly different from the learning objectives or principles centered view taken by other systems.

One of the nice features of a simpler procedure language is that it is more easily authorable by subject matter experts who are not computer science experts. Both RIDES and T^3 aim to ease scenario authoring by supporting a *scripting by demonstration* capability (in both cases complemented by a post-editing phase allowing somewhat different degrees of procedure generalization, as described earlier).[1] Currently FT^4 offers no authoring support for its procedures, and this is a recognized gap in its aim to address the problem of ITS cost. However, we have plans to integrate FT^4 both with the enacted sequence recording capabilities of T^3 and with a graphical flowchart editor that enables much more significant generalization of enacted sequences. We note that the design of the FT^4 procedure language was shaped by the expectation that a simple flowchart interface would be created; it relies on a primitive branch and link representation rather than more modern structured programming language constructs.

We note also, that flowcharts have been exploited in instructional and task aiding system in other contexts and for other purposes. For instance, FAST (Thompson, Ockerman, Najjar, & Rogers, 1997) is a wearable electronic performance support system intended to offer factory personnel just-in-time advise. One form of advice available is explicit flowchart representations of tasks to be accomplished presented to workers in response to questions such as "How do I do this?" Slightly further afield, Puerta (1993) describes how flowcharts can be used to support authoring of task-specific support tools. In his examples, physicians design protocol flowcharts, that then serve to help structure system behavior and interfaces.

3. The FT^4 Procedure Flowchart Language

In this section we provide a detailed specification of the FT^4 procedure flowchart language. Each flowchart represents a named procedure. There can be any number of procedures defined for an ITS, and steps in one procedure can invoke other procedures (including themselves recursively, either directly or indirectly).

Each procedure contains a set of steps, linked to one another through "next" pointers. Basic steps have just a single "next" pointer, while test steps can have two (or more) "nexts". Steps (or branches out of test steps) without an explicitly defined "next" are taken to constitute the end of a path. A path may be a route through a whole procedure, or may be the end of one chain in a parallel-step (which will be discussed at greater length below).

In addition, a procedure may specify exception conditions. When an exception condition becomes true, all other steps become unavailable, and control jumps to the step specified by the exception handler. When the chain of steps beginning with this initial step of the exception handler is completed, control can either return to the point where the exception occurred (indicated by a special-purpose return-step), or it can jump to a statically specified point in the flowchart.

Ignoring exception return-steps, procedure start-steps and other such special-purpose step types, there are five main kinds of steps:

- **Primitive-Step:** A primitive-step corresponds to a real-world action. In order to move past such a step, some action must be taken.
- **If-Step:** An if-step is a kind of test-step that is a simple if-then-else two-way branch. If the test expression evaluates to true, then the first path should be taken, otherwise, the second path should be taken. Thus an if-step has two next steps, but only one or the other is ever available.
- **Cond-Step:** A cond-step is a more flexible kind of test-step than the if-step; it resembles the "cond" form of the Lisp or Scheme programming languages in that it can have any number of distinct tests, each of which controls a possible branch. The tests are evaluated in order, and the

[1] An interesting feature of T^3 is that, contrary to the approach in RIDES, it does *not* require tight integration with its own simulator system to achieve this scenario scripting by demonstration capability.

branch paired with the first test that evaluates to true is considered active. A cond-step has as many next steps as it has tests, but again, only one is ever available.

- **Procedure-Step:** A procedure-step represents an invocation of a nested procedure. When a procedure-step is reached, control jumps to the nested procedure. When that nested procedure completes, control resumes with the procedure-step's designated next step.
- **Parallel-Step:** A parallel step consists of several possible chains of execution, some of which may be specially designated as *optional*. A parallel step specifies a minimum and a maximum number of the main (non-optional) chains that must be completed. A chain is completed by reaching a step in the chain that has no next step. The user may jump back and forth between steps in different chains, and can start new main chains, up to the designated maximum number. Once the minimum number of main chains are completed, and so long as no other chains in the parallel-step are in progress, the parallel-step's own next step may be executed.

Parallel-steps are the most complicated type of step. They were designed to support unordered or parallel actions (i.e. a set of N main chains, with the minimum and maximum chain counts both set to N), alternative actions (i.e. a set of N main chains, with the minimum and maximum both set to 1), and optional actions (i.e. allowance for a set of optional chains). But the minimum/maximum capability also allows for the somewhat odder possibility of unordered subsets of the main chains (i.e. a set of N main chains, with the minimum set to J and the maximum set to K, where 1 <= J <= K <= N).

The FT[4] procedure flowchart language can be cast in several different forms. Currently, we have defined an ASCII file format (based on Lisp-style S-expressions), and an in-memory Java data structure format (suitable for tracing through procedures and determining, at each point, what are the current allowable set of actions a user might take). As suggested by the title of the system, the language was also designed to have a direct mapping to graphical flowcharts. Such a graphical language has been designed, but not yet implemented. Figure 1 shows a sample procedure for making cappuccino, primarily intended to illustrate the majority of features of the language (and to highlight possible problems as well).

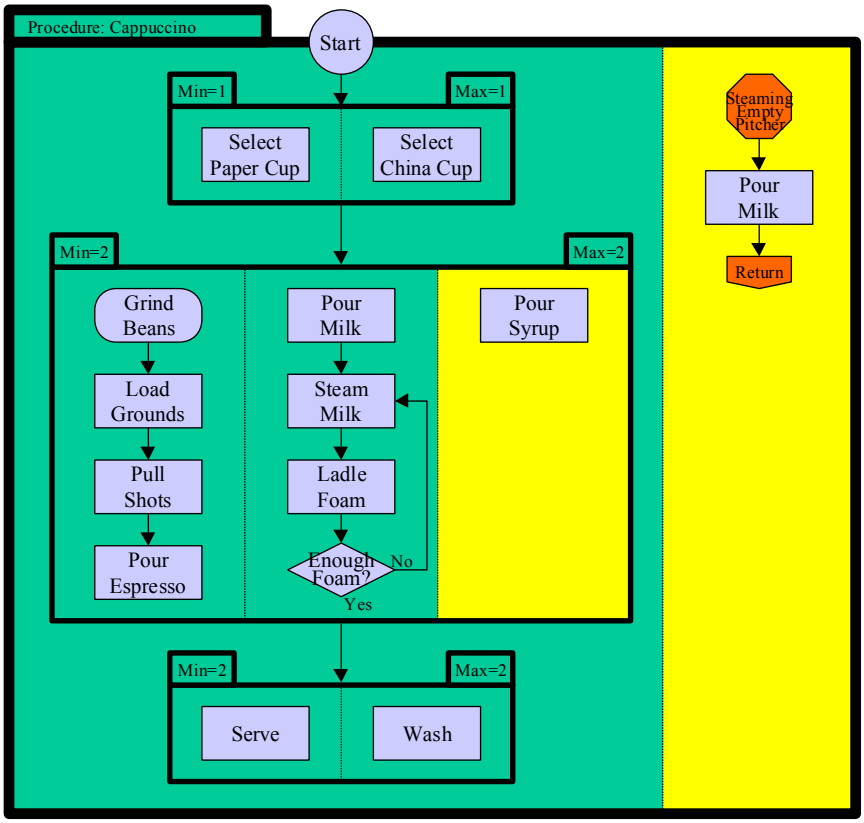

Figure 1. A Flowchart View of the Sample "Cappuccino" Procedure.

The outermost heavy-lined box in Figure 1 represents the entire "Cappuccino" procedure. It is divided into two parts: (1) the mainstream procedure flow (the larger area with the darker background), and (2) the exception conditions (the smaller area to the right with a lighter background). The main procedure is composed of three major sets of steps, where each set is grouped into a complex parallel-step. The parallel-steps are the boxes with medium-weight lines, and with Min/Max counts shown attached. Again, the parallel-steps are potentially broken up into two parts: (1) the set of main chains (shown with a darker background), and (2) the optional chains (present only in the middle parallel-step, and shown with a lighter background). The thin-lined rectangles represent primitive-steps. The thin-lined rounded rectangle "Grind Beans" represents a procedure-step (the invocation of some other named procedure, whose detailed structure is not shown in this figure). The thin-lined diamond "Enough Foam?" represents an if-step that has the effect of creating a loop.

From the start state of this procedure, the only reachable step is the first parallel-step that packages the two "Select Cup" primitive-steps as the starting (and ending) points of two alternate chains. These are *alternate* chains because the min and max constraints are both set to 1. That means that as soon as one of the "Select Cup" steps is executed, a first chain has been started, and since the maximum number of allowed chains is 1, no other chain can be executed. Whichever "Select Cup" primitive-step is executed also completes the first parallel-step.

From that point, it is legal to start any of the three chains in the second parallel step. The chain consisting solely of "Pour Syrup" is an optional chain, and so its execution does not count towards the min/max counts of the parallel-step. With min and max both set to 2, the two main chains must both be executed to fully satisfy the parallel-step. The two main chains are a bit more complex than those in the first parallel-step. The chain that starts with "Grind Beans" is a straight sequence, except that the "Grind Beans" step is shown as a procedure-step—a reference to another procedure—and therefore could require some arbitrarily complicated set of actions before it is possible to advance to the "Load Grounds" step. In the second chain, we illustrate the use of an if-step, and the creation of a loop; the point here is simply to show that tests of environmental conditions are allowed and can be used to determine what steps ought to be performed next. Finally, as an aside, we note that the chains of this parallel-step adhere to the structural rule that each chain is self-contained: there can be no crossing over from the steps of one chain to those of another.

The third parallel-step is similar to the first, but since its min and max are both 2, it amounts to an unordered "and" rather than an "or"—it is necessary to both serve the cappuccino and wash up to complete the parallel-step, and thereby to complete the entire "Cappuccino" procedure.

The only aspect of the procedure we have not yet discussed is the exception condition set to watch for attempts to steam milk when the pitcher is empty. Structurally, the interesting point is the return-step at the end of the exception-handler; this should cause control to return to "Steam Milk" step (which is the only place the exception could have been noticed). Of course an exception that can occur in only one place might be defined as an explicit test and branch at the point in question. Alternately, this exception might better be defined in the scope of the second parallel-step rather than in the scope of the procedure; however, our current syntax does not allow such distinctions in exception scope.

As noted earlier, this procedure language is less expressive than a plan-oriented language; this leads to procedures that are less flexible than they logically could be. For instance, there is no real reason to force the cup selection to be performed before starting in on preparing the components of the drink. Logically, the cup only needs to be selected before the first pour or ladle step (to ensure there is something to pour our ladle into). On the other hand, in procedural training applications, it is quite common for the "by the book" way to accomplish a task to be defined with limited flexibility (compared to all the possible sequences that might accomplish the goal in question). At least for the applications we have considered, we are not very concerned that the FT[4] procedure language is too restrictive.

4. The SatCon Project

The work reported here was carried out in the context of a project called SatCon (for Satellite Control). The point of this project has been to demonstrate how AI technologies can be applied to build a system for US Air Force satellite controllers that provides both intelligent simulation-based training and

job task-support during actual operations. The project ran for six months, resulting in a first prototype that links the current FT^4 with a simulator and a GUI through an agent community. We expect to start development of a more complete operational system in the summer of 2002, as part of a two-year follow-on project. Section 5 on future work lays out some of our goals for the ongoing effort.

Given the dual application requirement, the multiplicity of potentially relevant legacy systems, and the expected overall system complexity, an agent architecture was early identified as particularly appropriate. Given the nature of the satellite control task—complex and somewhat variable, with a wide range of contingencies, and with tremendously valuable resources at stake—operators in current (and projected) practice are largely restricted to acting within the bounds of pre-specified scripts or procedures. Those procedures are far too complex to be expressible in the T^3 task language, thus our invention of the significantly more expressive FT^4 language.

To date, we have (1) designed the FT^4 procedure language (including internal data structures and external representations in both ASCII and graphical form), (2) built a parser to transform ASCII specifications into data structures, and implemented a tracing algorithm that tracks allowable next actions, (3) designed a complementary student modeling formalism, and mechanisms to use such models to drive selection and/or generation of training scenarios, (4) implemented student model creation and update during the procedure tracing process, (5) packaged the existing FT^4 components so as to separate the training and task-support capabilities from any simulation or operational environment, and (6) constructed a first demonstration application by embedding FT^4 in a distributed agent environment.

5. Future Work

Future work will focus on authorability. This will include porting the existing T^3 capability for creating straight-line procedure specifications, and combining it with the interactive flowchart editor. In addition, we will complete implementation of the mechanism for automatically generating individual scenarios based on general procedure specifications combined with student models. With these additions, we expect to fulfill our goal of cost-effective high-quality automated on-line learning environment for procedural tasks.

6. References

Core, M., Moore, J., and Zinn, C. (2000). Supporting constructive learning with a feedback planner. AAAI Fall Symposium on Building Dialogue Systems for Tutorial Applications, Technical Report FS-00-01, November 2000.

Guralnik, D., An Authoring Tool for Procedural-Task Training, PhD Dissertation - Technical Report #71, The Institute for the Learning Sciences at Northwestern University, 1996.

Munro, A. and Q. A. Pizzini (1995), RIDES Reference Manual, Los Angeles: Behavioral Technology Laboratories, University of Southern California.

Ong, J., Noneman, S. (2000). Intelligent Tutoring Systems for Procedural Task Training of Remote Payload Operations at NASA. In Proceedings of the Interservice/Industry Training, Simulation, and Education Conference. Orlando, FL.

Puerta, A.R. (1993). The study of models of intelligent interfaces. Intelligent User Interfaces 1993: 71-78

Rickel, J. Ganeshan, R., Rich, C., Sidner, C., Lesh, N. (2000). Task-Oriented Tutorial Dialogue: Issues and Agents. AAAI Fall Symposium on Building Dialogue Systems for Tutorial Applications, Technical Report FS-00-01, November 2000, pages 52-57.

Thompson, C., Ockerman, J., Najjar, L., Rogers, E., (1997). Factory Automation Support Technology (FAST): A New Paradigm of Continuous Learning and Support Using a Wearable. ISWC: 31-38