

HYPertext SUPPORT FOR LEARNING COMPLEX INTELLECTUAL PROCESSES

David Stotts
Dept. of Computer Science, Univ. of North Carolina
Chapel Hill, NC 27599-3175
(919 962-1833, stotts@cs.unc.edu
<http://www.cs.unc.edu/stotts/>

The main characteristic that separates hypermedia structures from information collections such as databases, IR systems, and digital libraries, is that hypermedia integrates *task* with *information*. The "hyper" in hypermedia is a technology by which an author encodes into the browsing process the collective experiences a reader may have with a body of multimedia information.

We report here on a simple comparative study of teaching methods, a study that exemplifies the fundamental hypermedia-as-process characteristic. Using the ubiquitous World Wide Web, we have encoded interactive notes on the complex intellectual activity of program verification. Traditionally a difficult topic to learn as well as present systematically, we found that a presentation based on hypermedia notes made the concepts and activities of program proving easier for students to learn effectively in recent software engineering classes. We briefly outline the dynamic structure and organization of the Web document used, and comment on our informal evaluation of its effectiveness with the students.

1 Integrating task with information

A fundamental characteristic of hypertextual structures is that they *integrate task with information*¹. This necessary fusing conceptually distinguishes hypertextual structures from collections of information such as databases of all flavors, information retrieval systems, and electronic archives. The integration of process with information should be effectively exploitable in many domains of human activity. We report here on a study of one such area: learning complicated intellectual procedures.

A great deal of research is devoted these days to retrieving information from hypertextual networks. While such investigations are important and are yielding useful techniques, they view a hypertextual structure as fundamentally static—a data and information repository. Our work, both in basic research and in application of the resulting ideas, views a hypertextual structure as having true *value added* over a data repository. A hypertext certainly contains stored information, but it also admits various tasks, or processes, that will encompass, use, and interpret that information. In its broadest (and earliest) form, this task was unfettered browsing, following any and all links that existed among the information components. On the other extreme, the Trellis model offers a view of integrating task with information in which the total collection of links in a document is treated as an automaton [6, 5]. The execution rule for the automaton defines the document's *browsing semantics*, that is, the allowed manner in which the information components of the document can be visited, combined, and presented.

¹ Though this concept has been for years a basic premise of the work by R. Furuta and the author in Trellis, I am indebted to Steve Poltrock of the Boeing Corporation for this succinct and eloquent phrasing.

The point of this paper is not to survey approaches to integrating task with information, but to emphasize one advantage that this view of hypertextual information structures brings—namely, that the encoding of complex processes in a hypertext can aid in learning those processes.

COMPLEX INTELLECTUAL PROCESSES: FORMAL SPECS AND VERIFICATION

One of the mainstays of a class in formal methods for software engineering is mastery of various techniques for program verification, also referred to as program proving. In this activity, generally agreed now to be egregiously but historically misnamed, no guarantee is given that a program is error free, but rather a structured argument is developed to show that a program corresponds semantically to formal specifications of its required behavior or properties.

There are several methods in use for verification. One of the earliest, attributed to Hoare [3, 4], is termed *axiomatic verification*. It features specifications written as predicates in a first order logic, asserting relationships among variable values in the program state. An axiomatic proof consists of a demonstration that each statement in a program transforms these predicates in such a way that the assertions holding prior to execution—the *input specifications*, or *preconditions*—imply the truth of the assertions holding after execution—the output specifications, or *postconditions*. The result is a reasoned claim that, if execution begins in a state in which the preconditions are true, and should execution complete normally, then the final state of the program will be one in which the postconditions are true.

A related topic of similar intellectual complexity is the axiomatic specification of an abstract data type (ADT). Formal expressions – the axioms of the type – are written describing its behavior without reference to the details of any specific implementation methods. For example, in specifying a *stack* of integers, we might specify that

$$\text{pop}(\text{push}(S, a)) = S \text{ for any stack } S \text{ and integer } a.$$

This assertion contains a certain “essence of stackness” and must be true whether one chooses to realize stacks in code with arrays or with linked lists. Once specified, the user of an ADT user may wish to prove that the set of axioms exhibits a certain behavior. Such a proof is done via a method called *data type induction*.

In this report we present three examples of encoding complex intellectual processes in hypertext: program verification; construction of axiomatic ADT specifications; and proving ADT properties via data type induction.

SUMMARY OF MAIN RESULTS:

We produced hypertextual notes explaining and exemplifying axiomatic ADT specification and algebraic program verification for recent graduate classed in software engineering. The resulting information structures produced an interactive style of lecture notes; lectures were delivered by projecting the hypertext screen, and notes were distributed to students for their personal study. We found lectures and notes delivered in this hypertextual form to be noticeably more effective than traditional lectures with traditional notes at producing in students a working understanding of formal specifications and program verification.

In the following sections we describe the structure and behavior of the hypertext used, and discuss the class experience in using it.

2 A brief word about Iris hypertext

The Iris system for DOS is very simple to use and the document format is simple to author. Each document is marked-up ASCII text, with the file divided with labels into panels.

Links are indicated on the screen in two ways. Any word in the main body of text with an asterisk after it is an embedded link; this kind is selected by using the tab key to highlight it (tabbing cycles through all embedded links), then pushing carriage return to follow it. The word “PROOF_PLAN” in figure 2 shows an embedded link.

Iris allows creation of a menu of links for each frame. By default, the menu with a single link, shown as a left-arrow in the lower right of the screen (see figure 1). The default takes the reader to the next frame (sequentially in the source file), and is selected automatically with carriage return. The author can explicitly place names links in the menu and override the default CR. In figure, for example, the word “Done?” appears as the single link in the menu. It is programmed to cause a return to the previous frame.

3 Example: Interactive program proof

The Iris hypertext system is a shareware product for DOS machines. It is especially useful for lecture support applications because of its combination of simplicity, flexibility, and ease of authoring. No mouse is required in Iris, and authoring is done with any editor by inserting markup lines in ASCII text. One file is one document in Iris. The overall design of the markup language is nicely intuitive, and effective documents can be built after learning only three or four markup commands. The documents discussed are constructed with about ten markup features, which can be mastered in about an hour of reading and exploration.

The main observation exploited in this use of hypertext is that teaching program verification is largely a demonstrative endeavor. Students easily follow the proof process when a lecturer performs one on a whiteboard. They cannot take effective notes, however, while watching such a demonstration, because the *process of annotating* a program is the skill to be learned. A student’s traditional notes will look like the final annotated product, with no process description recorded. Students of program verification cannot then repeat the demonstration from their notes.

The complexity of the link web in this example is low. It is mostly intended to be used as a presentation, going from one frame to the next in the order of the verification process. However, at any frame the reader may select detours, going off to the side to review the proof plan, for example, or pausing to review the reasoning behind the invariant. It is this combination that is effective, we think. The presentation (i.e., the lecture, and the underlying process) is there for recreation, but the intellectual points to be learned are there when needed without interrupting the overall flow of the process.

We also do not attempt to give the entire example here as screen dumps. Instead, we seek to give only enough of the process to give a good idea of the flow of information on the screen. The full example, with Iris, is available over the Internet (see section 7).

4 Example: Generating abstract data type axioms

A second example of using hypertext notes for interactively illustrating complex processes is the heuristic for generating a consistent and complete set of algebraic axioms for abstract data types, as explained by Guttag [1, 2].

An *abstract data type (ADT)* is an encapsulation formalism for data that provides half the intellectual basis for object-oriented computing (inheritance being the other half). An ADT consists of a set of base types (the *sorts* in algebraic terms), a set of operations over those types, and a set of axioms that describe in an implementation-independent manner the behavior of the operations. A designer of ADTs is concerned that the axioms be both *complete* and *consistent*; that is, they should express all intended behaviors for the operations while not specifying any contradictory actions.

Guttag's ADT construction method is a heuristic for producing axiom sets that are consistent and complete. We have found in teaching this heuristic that students have difficulty understanding that the success of the method depends on following closely a specific procedure; they often will deviate from correct practice without realizing that anything is amiss.

The heuristic is as follows.² A *canonical form* is selected for the ADT; this is a standard way of representing any instance of the type. In a *STACK* type, for example, any specific instance can be built by invoking a *new()* operation followed by some number of *push()* operations. The operations used in the canonical form are identified as *constructors*; all other operations are *non-constructors*. The non-constructors are operations that are not necessary to “directly” build any specific instance of a type; for example, to create a *STACK* with, say, 4 elements in it, we do not need to invoke any *pop()* operations, since we can build it by a *new()* operation followed by the appropriate 4 *push()* operations.

Once the constructors have been identified, the axioms are generated by forming all combinations of applying a non-constructor to a constructor. For an ADT with n constructors and m non-constructors, the method will generate $n * m$ axioms. To emphasize this approach, the screen sequences we built into the hypertext lecture notes first generate all left-hand sides by doing these combinations. After that, we go back and supply right-hand side equivalent expressions to complete each axiom.

5 Example: Proofs of ADT properties

Another application for the process encoding is in generating proofs of properties of ADTs. For example, if you have defined a *STACK* of integers as an ADT, you may wish to prove that every *STACK* has \leq *MAX* elements (for bounded stacks). This can be done with data type induction, and form of structural induction done on the length of the sequence of operations needed to construct an element.

The screens in figures ?? to ?? show how such a proof is presented to the students. The structural features of the presentation are similar to the program proof shown earlier.

²We assume the reader has some familiarity with the basic components of an ADT; if not, the papers by Guttag cited previously contain the necessary descriptions.

6 Observations on effectiveness

We have taught program verification to graduate students in seven different classes over the past six years, at the University of Maryland, the University of Florida, and most recently the University of North Carolina at Chapel Hill. We used the Iris-based notes in the a class at Florida two years ago, and in a class at UNC in fall 1994. The hypertext system was used both as lecture support (projecting the screen of the DOS laptop running Iris) and as personal support, by distributing electronic copies of the notes with browser to the students for their individual study and modification.

All students reported strong enthusiasm for the notes and the method of delivery. In prior classes, no enthusiasm for the subject was reported, and the topic was even disliked compared to other ones in the software engineering class. More importantly, the results of the hypertextual delivery showed up in the work produced by the students. Proofs done by the class that used the Iris notes were very well organized and showed stronger levels of understanding of the basic proof process than in prior classes. In comparison, classes without the use of the hypertextual presentations tended to produce proofs that were incomplete, sketchy, and lacking steps in the basic process. I attribute this to the fact that a student with traditional notes from a traditional lecture must re-invent the actual process from notations that do not contain good indications of the process demonstrated by the lecturer. The dynamics of the technique cannot be recorded in a paper-based notebook, and consequently, during study and learning, the student often does not recreate the dynamics correctly or completely.

No controlled evaluation was performed, so the discussion here should be viewed as a report of collected observations rather than conclusive experimental results. Some controls on the teaching methods were present, however; for example, the proof style, phraseology, and general content used in the hypertextual notes are essentially the same as the content used in the previous classes with traditional lecture. We cannot attribute the differences noted to newer or more refined content. The example programs proven in class and those assigned for practice have been mostly the same in all classes. In past classes, students were given paper copies of extensive lecture notes, with practice problems; this parallels the hypertext notes given in the recent class. The main difference we see is the organization, delivery, and dissemination of the content in hypertext versus the traditional formats of lecture and paper notes.

We believe that the representation of the process—inherent in a hypertextual presentation—was instrumental in obtaining the improvements we noted. Students have examples not only of individual assertion forms, but of the creation of sequences of assertions as well. These examples are interactive and may be explored or replayed in various ways as required for mastery by each individual.

7 Conclusions

We suspected that hypertext notes would assist student to learn the complex intellectual process of program verification more easily and more accurately. While the trials described in this report are in no way a controlled experiment, we feel the results are encouraging and lend credibility to our suspicions.

ACKNOWLEDGEMENTS

The author would like to acknowledge Art Crummer at the University of Florida for helpful comments on my use of Iris during drafting of the software engineering notes.

INTERNET ACCESS SE NOTES AND IRIS

As with many hypertext documents and ideas, the point of this development is much more apparent when demonstrated than when simply discussed. The screen image sequences in our examples are helpful, but not nearly as instructive as the experience of browsing the proofs. The Iris notes for program verification are available via anonymous ftp, along with the Iris browser. Connect to `ftp.cs.unc.edu` and in directory `pub/stotts/iris` find file README that will provide further instructions. They can also be obtained at Web node `ftp://ftp.cs.unc.edu/pub/stotts/`. Though distribution is via Unix, the Iris browser is executable only on DOS systems, so some decoding and transfer to other disks will be necessary. Comments and modifications are welcomed from readers who are interested in continuing this experiment.

References

- [1] J. Guttag. Notes on type abstraction (version 2). *IEEE Transactions on Software Engineering*, pages 13–23, January 1980.
- [2] J. Guttag, E. Horowitz, and D. Musser. Abstract data types and software validation. *Communications of the ACM*, pages 1048–1064, December 1978.
- [3] C.A.R. Hoare. An axiomatic basic for computer programming. *Communications of the ACM*, pages 576–580, 583, October 1969.
- [4] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language pascal. *Acta Informatica*, pages 335–355, 1973.
- [5] P. D. Stotts, R. Furuta, and J. C. Ruiz. Hyperdocuments as automata: Trace-based browsing property verification. In *Proceedings of the 1992 European Conference on Hypertext (ECHT92: November 30–December 4, Milan, Italy)*, pages 272–281. ACM Press, New York, 1992.
- [6] P. David Stotts and Richard Furuta. Petri-net-based hypertext: Document structure with browsing semantics. *ACM Transactions on Information Systems*, 7(1):3–29, January 1989.

```
{ PRE: x >= 0 }                                ( PROOF_PLAN* )
begin

  q := 0

  r := x

  while r >= y do

    r := r - y

    q := q + 1

  endwhile

end
{ POST: 0 <= r < y and x = q * y + r }
```




Figure 1: Opening screen: the program to be proven.

OUTLINE OF PROOF or "PROOF PLAN"

step 1: gen a loop invariant INV, and show that
INV is invariant for the loop

step 2: push the INV back to the beginning
of the program, and show that the
'PRE' spec implies the transformed INV

step 3: push INV forward to the end of the
program and show that the transformed
INV implies the 'POST' spec




Figure 2: After selecting the "PROOF PLAN" link.

```

{ PRE: x >= 0 }                                ( PROOF_PLAN* )
begin

    q := 0

    r := x
    { INV*: 0 <= r and x = q * y + r }
    while r >= y do

        r := r - y

        q := q + 1

    endwhile

end
{ POST: 0 <= r < y and x = q * y + r }

```




Figure 3: Generation of the Invariant.

```

{ PRE: x >= 0 }                                ( PROOF_PLAN* )

INV*: 0 <= r and x = q * y + r

```

GENERATING THE INVARIANT

To get INV, in this simple case, we take the condition we need after the loop (here, POST) and "subtract out" the negative of the loop boolean (here, $\sim(r \geq y)$).

```

    q := q + 1

    endwhile

end
{ POST: 0 <= r < y and x = q * y + r }

```




Figure 4: Explanation of how INV is obtained.

```

{ PRE: x >= 0 }                                     ( PROOF_PLAN* )

  INV*: 0 <= r and x = q * y + r

  Then, we must convince ourselves "by eyeball"
  that our candidate INV is indeed invariant.

  Check its validity just prior to loop execution
  for the first time: here, q = 0, r = x,
  so INV reduces to 0 <= x and x = x, which is true.

  q := q + 1

  endwhile

end
{ POST: 0 <= r < y and x = q * y + r }

```




Figure 5: Continuation of explanation of how INV is obtained.

```

{ PRE: x >= 0 }                                     ( PROOF_PLAN* )

  INV*: 0 <= r and x = q * y + r

  Now check what one loop execution does to the INV.
  r get smaller by y, q goes up by 1, so the effect
  on INV is  $x = (q+1)*y + (r-y) = q*y + r$ 

  So, things look good... let's formalize this
  thinking.

  q := q + 1

  endwhile

end
{ POST: 0 <= r < y and x = q * y + r }

```




Figure 6: Final frame of explanation of how INV is obtained.

```

{ PRE: x >= 0 }                                ( PROOF_PLAN* )
begin

  q := 0

  r := x
  { INV*: 0 <= r and x = q * y + r }
  while r >= y do

    r := r - y
    { P1: 0 <= r and x = (q+1) * y + r }
    q := q + 1
    { INV*: 0 <= r and x = q * y + r }
  endwhile

end
{ POST: 0 <= r < y and x = q * y + r }

```




Figure 7: Pushing INV back through the loop body.

```

{ PRE: x >= 0 }                                ( PROOF_PLAN* )
begin

  q := 0

  r := x
  { INV*: 0 <= r and x = q * y + r }
  while r >= y do
    { P2: 0 <= (r-y) and x = (q+1) * y + (r-y) }
    r := r - y
    { P1: 0 <= r and x = (q+1) * y + r }
    q := q + 1
    { INV*: 0 <= r and x = q * y + r }
  endwhile

end
{ POST: 0 <= r < y and x = q * y + r }

```




Figure 8: Pushing INV back through the loop body.

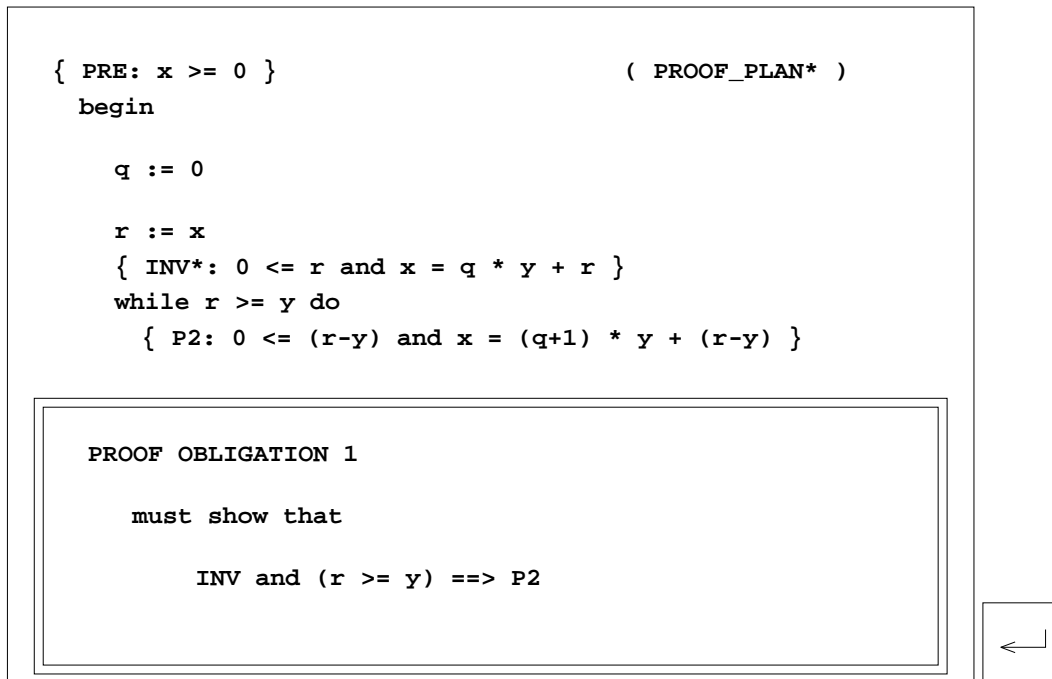


Figure 9: First proof obligation.

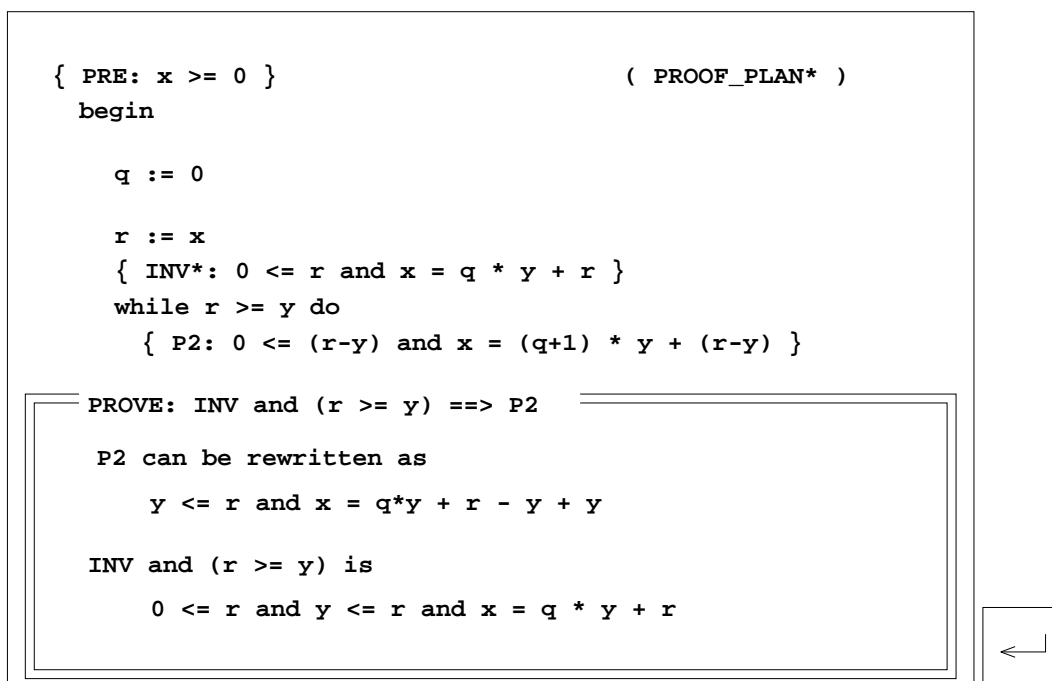


Figure 10: Satisfying the first proof obligation.

```

{ PRE: x >= 0 }                ( PROOF_PLAN* )
begin

  q := 0

  r := x
  { INV*: 0 <= r and x = q * y + r }
  while r >= y do
    { P2: 0 <= (r-y) and x = (q+1) * y + (r-y) }

  PROVE: INV and (r >= y) ==> P2

  So, each clause of P2 is directly a clause
  of INV and (r >= y)

  QED

```




Figure 11: Final frame of first proof obligation.

```

SET of INTEGER

sorts: SET, INTEGER, BOOLEAN

operations:
  empty:                --> SET
  insert: SET x INTEGER --> SET
  delete: SET x INTEGER --> SET
  member: SET x INTEGER --> BOOLEAN

```




Figure 12: Developing operation signatures for an ADT.

```
SET of INTEGER

sorts: SET, INTEGER, BOOLEAN

operations:
  >> empty:                --> SET
  >> insert: SET x INTEGER --> SET
  xx delete: SET x INTEGER --> SET
  xx member: SET x INTEGER --> BOOLEAN
```




Figure 13: Identifying canonical constructor operations.

```
SET of INTEGER

sorts: SET, INTEGER, BOOLEAN

operations:
  >> empty:                --> SET
  >> insert: SET x INTEGER --> SET
  xx delete: SET x INTEGER --> SET
  xx member: SET x INTEGER --> BOOLEAN

axioms:
  member(empty(),j) =
  member(insert(S,j),k) =

  delete(empty(),j) =
  delete(insert(S,j),k) =
```




Figure 14: Constructing axiom left-hand sides.

SET of INTEGER

sorts: SET, INTEGER, BOOLEAN

operations:

```
>> empty:                --> SET
>> insert: SET x INTEGER --> SET
xx delete: SET x INTEGER --> SET
xx member: SET x INTEGER --> BOOLEAN
```

axioms:

```
member(empty(),j) = false
member(insert(S,j),k) =
  if (j=k) then true else member(S,k)

delete(empty(),j) = empty()
delete(insert(S,j),k) =
  if (j=k) then delete(S,j)
  else insert(delete(S,k),j)
```



Figure 15: Full presentation of ADT axioms.