4 January 1990

Enhancing documents with embedded programs: How Ness extends insets in the Andrew ToolKit[1]

Wilfred J. Hansen
Information Technology Center
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract: *An enhanced document responds to its reader in non-traditional ways: a button press may scroll the document, play music, ... Although such operations may be implemented as objects embedded in text, full generality requires that a programming language be available to the author of a document. This paper sketches the problems of embedding programs in documents and reviews the solutions adopted in the Ness component of the Andrew ToolKit. A key question is the connection from user actions to program functions. Other questions include the appropriate level of programming language, its string processing capabilities, and security.*

Traditionally a computer document is an emulation of a paper one; it sits there for the reader to explore at will. This present paper is no different because even though I am writing it on a computer I expect most readers will view it on paper. But suppose you were reading it at a computer; how much more could it do? Simulations, calculations, interactive examples, waving flags, music, fireworks?

The report below describes a system in which it is possible to write documents that have most of these behaviors. In order to provide the most general environment, the system incorporates a programming language, Ness, the design of which has been kept simple so as to reach a broad range of authors. The underlying system is the Andrew ToolKit (ATK).

Typical applications of such a system include

> enhanced documents - with various animations and simulations to illustrate the points of the document
>
> parameterized letter - after the user fills in a few fields in a form letter other fields are calculated and the full letter is constructed
>
> personal data base - addresses, appointments, course records, bibliography, ...
>
> directory editor - click on a file to see its attributes or select it for moving or deleting
>
> system status monitor - a user builds a personal monitor for work station status by selecting from a library of system measurement tools and a library of ways to view dynamically changing values: dials, strip charts, and so on.
>
> dungeons and dragons - the description of the world is a large text through which the reader can scroll; as the reader solves puzzles, descriptions of new rooms and objects are added to the text.

The remainder of this paper discusses the system as though it were intended for extending document: objects are inserted in the document at various places and their behavior is controlled by a *script* written in the programming language. In all cases the discussion applies equally well to programming an application; such a situation is just an image that cannot be scrolled off the screen as parts of a document can. For documents the substrate is a text; for applications the substrate is a drawing editor or some other tool for laying out the contents of a window.

Hypertext documents can be created easily within a system that supports authorship of enhanced documents. The author need only insert in the text a button extended to respond to a button press by scrolling elsewhere in the same document or another. Nor is the link constrained to always branch to the same place; the author can arrange the script so the destination is chosen among several depending on what the reader has seen so far. Because lengthy linear documents are possible, the Ness/ATK combination may reduce the disorientation readers sometimes encounter in hypertext systems with small nodes. The scroll bar in the view of the document serves as a visual indication of where the current image is within the context of the entire document. See [8] for a good discussion of the advantages of hypertext and references to the growing literature on user disorientation.

Some discussion in the multi-media mail community has focussed on using a language to describe mail documents. With this facility, new varieties of objects can be sent if the receiving system has no more capability than the language interpreter. The work reported below is both more and less general. It is not a language suitable for describing any object, it assumes that a collection of objects will be available in the software of both sender and receiver. It does, however, provide a language tailored for the author to describe a myriad of different forms of interconnection and behavior of objects.

It is important to distinguish the notion of enhancing a document with a script from the various authoring languages for educational tutorials (good examples are cT and Best Course of Action; see [7]). With the latter, the author constructs a program which *generates* a sequence of images; this contrasts with the enhanced document approach of Ness where the program within the images. The crucial difference is in user control: with an enhanced document the reader is in control and can employ ordinary text operations to move through the text. With program generated images control lies with the system; the reader can move only to where the system allows. Experience has shown that it is not easy for authors to always imagine where readers will want to go in reviewing a tutorial, so the reader many be sometimes stymied in trying to get to a desired place.

The best known system with a programming language for enhancing what the reader sees is Hypercard [1]. However, Hypercard operates in a far more limited environment: its images are constrained to a certain small size and only two forms of object may be embedded, buttons and fields. In contrast the system described below is implemented within the far richer environment of the Andrew ToolKit, ATK. [5]

Underlying ATK is an object architecture, complete with inheritance of methods. Two principle forms of object are

> *data object* - this provides for storage and manipulation of information; every data object has at least the methods for writing the object to a data stream and reading it back.
>
> *view* - this provides user access to the contents of a data object by displaying it in a rectangle on the screen; it also provides user interface operations for scrolling or modifying the information.

The ATK architecture is specifically designed so that a view cannot know whether it occupies an entire window or whether it is a subrectangle in another view. If a view has embedded child views, the architecture provides for the parent view to completely control the events seen by the child, thus views can be nested arbitrarily. To describe what the user sees of the combination of view and data object, this combination is called an *inset*. It is worth noting that the architecture proved sufficiently general that little change to the underlying system was required to add Ness.

In starting Ness, it was not clear how script pieces should interface with insets and the user. The many design alternatives are outlined in the next section and the following section describes the choices made for Ness. A sketch of other facets of Ness--especially the string algebra--is in the third section, followed by a discussion of security issues in the last section.

1. How can documents be enhanced?

Define an "extension" as a sequence of code to be executed at some particular time during a user's perusal of a document. The principal semantic questions are

1) What sorts of events can trigger the extension?

2) What effect can the extension have on the containing text and surrounding objects?

It is apparent that in the usual workstation environment the set of events that can trigger an extension must include user actions with the keyboard and the mouse. In ATK, an important subset of mouse actions are selection from the popup menu. Another set of trigger events are those defined by each inset class. For instance text objects may initiate an event when some portion is selected or when the text scrolls, both of which may be caused by any of several user actions.

In principle, inset-defined events would be a sufficient set of triggers. However, user inputs are an important additional form of trigger because they enable the extension language to deal with any object, whether or not it has been constructed with the language in mind.

How is each object to be connected to the portion of the script which specifies the response to events on that object? One answer to this is given by Hypercard, which associates a script with each object. The difficulty with this approach is that scripts and their functionality get scattered all over the place. In trying to understand what happens in response to a mouse click on a button it may be necessary to look at as many as five different scripts (button, card, background, stack, and home stack); and trying to understand the interaction of several buttons requires looking at each in turn. The situation is similar to the difficulties of trying to understand a spreadsheet of even modest complexity [3].

In Ness all the scripts for a collection of objects are gathered together. Each object is named and the extensions for an object appear together in a group identified with the name of the object. Review of the script is thus reduced to looking in a single place, though it is now not as easy to see which object each name refers to, especially if many objects are extended. Ness authors can avoid this by using multiple scripts in the document, one for each section. It is the author's responsibility to clarify as far as possible the relation between script portions and objects.

A few people who have heard about Ness without seeing it have suggested that the script ought to include not only the extensions but also textual descriptions of the objects themselves. Among the claimed advantages are that the script could be editted with a plain ASCII editor and that scripts could be generated by programs. Ness has taken a very different approach. Construction of objects and their assembly into a document or application are handled with the ordinary ATK object oriented editing facilities. If a button is required in the upper left corner, the author puts a button there and it is immediately visible, just as the reader will eventually see it. Ultimately ATK data streams are indeed encoded in ASCII, so they can be editted with ASCII editors by incorrigible hackers. It is also possible for a Ness script to insert an object in the document or application. To select its size and place the script negotiates with the parent inset inset which the object is inserted.

In a fully developed system there is a point to having representations of the objects in the script so the author can click on one to request that it be highlighted in the document. At present this is handled for Ness by an "arbiter" application, part of the mechanism that handles naming of objects.

What can an extension do to its environment? A guiding principal of Ness is that the instructions in an extension ought to have at least the same capabilities as the user sitting at the workstation. Consequently, primitives are provided for simulating the user actions of mouse hits, menu selections, and key strokes. In addition, each object makes available a set of operations that Ness extensions may invoke.

One of the serious problems with allowing scripts to pretend to be the user is that ATK permits users to alter the binding from keystrokes and menu options to function calls. If the author writes a script which invoke the sequence ESC-N it may work for the author but fail for others who have ESC-N bound to a different operation. The correct approach is for the author to instead call the function which he or she has bound to the ESC-N key. Functions behave the same for all users.

How does the extension script refer to objects in the environment? For each execution of an extension there is one unique object, namely the one which triggered the extension; this object can be referred to by a special name. There is also a special name for the text in which the script is embedded; via this name the program can access any portion of the document and any of its other objects. Most commonly, however, objects are referenced by the same name which is used to associate extensions with objects.

2. The Ness ‘Extend’ construct

A Ness script is a sequence of attribute specifications: declarations of global variables, global functions, and *extend* blocks. An extend block associates a set of contained attributes with some named object. An extend block has the syntax:

> **extend** <name>
>   <attributes>
> **end extend**

where the <name> must be a string constant giving the name of an object.

There are situations in which it might make sense that the <name> value be an expression rather than a constant and that the extend construction be executable rather than a declaration. For instance an author may wish to have different extensions for an object at different times. This design was not chosen for Ness because I felt that readers and authors would be best served by a language with relatively "static" semantics; one where the reader could tell what the program refers to without a great deal of poking around the code. There could be confusion for some authors if they were allowed an expression for <name>: they might not know when the expression would be evaluated or whether a change in the value of the expression would cause the extend construct to refer to a different object. Furthermore, the static nature of the extend construct helps suggest to the author that the behavior of documents should be constant rather than dynamic; once a reader has understood a given button, it should not change its behavior without a clear cause. Nonetheless, a button can have differing behavior if an author really so desires; the extension for the button need only test a global flag and behave differently for different values.

Suppose the author wants to dynamically create an object and insert it in the document. How can extensions be specified for it? With Ness this can be accomplished if the name of the future object is pre-known. The <name> in the extend construct can refer to an object which does not yet exist. When it is created, the extensions will be automatically applied to it. For complete generality, however, it will probably be necessary to add to Ness some form of executable **extend** operation.

In additional to declarations and function definitions, the attributes in a extend block may include *event specifications*. Each such specifies a trigger event and a list of statements to be executed when the event occurs. In a manner similar to the extend construct, an event specification is a static declaration with the form

> **on** <event-type> <specifier-string>
>   <statements>
> **end** <event-type>

The <event-type>s are **mouse**, **menu**, **keys**, and **event**, where the first three intercept the various user events and the fourth reacts to named events initiated by objects. The interpretation of the <specifier-string>, which must be a constant, depends on the <event-type>. For **mouse** it says which button is to be intercepted and whether the interception is for the down stroke, movement while down, upstroke, or all of the above. For **menu** and **keys** it specifies which operations are to intercepted or provided for the user. In particular, the <specifier-string> for **menu** can add new options to the menu. The statements for an **on event** are executed whenever the object extended (by the surrounding **extend** construct) initates the event named by the <specifier-string>.

In writing Ness scripts one of the early discoveries is that intercepting a mouse hit means that that hit will not go to the object. In one example, the object was a slider and the final value, when the mouse is let up, is to be transferred to a computation. If the mouse is intercepted, the slider will not change its value in response to the mouse, so the script must perform an additional operation to tell the extended inset of the mouse change. A built-in function in Ness provides for passing the event along to the inset.

Passing an intercepted event along to an inset may seem similar to the "pass" operation in Hypercard, an operation that sends the current input message *up* the hierarchy to the next level (button to card to stack to ...). However, in Ness and ATK events go *down* the hierarchy so the bottommost object will not see the event at all if the surrounding script so dictates. In our estimation, this gives the script and surroundings more control with less special purpose code within the objects.

Naming of insets and interception of mouse, menu, and keystroke events is handled by the *cel* and *arbiter* insets provided by the ADEW component of ATK [4]. These mechanisms are outside the insets themselves, so their facilities can be utilized without the collaboration of the subject inset. Cels and arbiters are both examples of *wrapper* insets, ones which have a single child to which they allocate their entire screen space. Each cel wraps around a simple inset like a button, so it can name the button and intercept events. Each arbiter wraps around a substrate inset, like text, which can have multiple embedded objects; from this vantage the arbiter collects the names from all enclosed cels and makes the names available to Ness.

Since an arbiter wraps a substrate and a substrate may contain arbitrary insets, one subinset of an arbiter may be another arbiter together with its own enclosed substrate, Ness script, and collection of insets. So how does a script refer to objects in an arbiter other than its own. It doesn't. At present, a Ness script can only refer to objects that share the same surrounding arbiter as the script itself. This restriction could be eliminated, but not without some difficulty. Consider the case where an author has defined a simulation with a collection of named insets and a Ness script that refers to them. Suppose then that the author inserts two copies of this simulation in a document. It is only because each is surrounded with its own arbiter that name conflicts are avoided: the Ness script in each simulation still refers to the local insets. As yet we have not enough experience with Ness to suggest that a capability of referring to objects within embedded arbiters is a necessary capability.

The statements within an **on** construct may refer to *currentinset* to refer to the inset whose event has triggered the current execution; they may refer to inset(<string expression>) to refer to the inset whose name is given by the <string expression>. Both constructions yield objects as their values; objects on which it is possible to invoke two classes of function that are defined in C code. First, the script may refer to methods and instance variables of the object. Since this is not well protected and can lead to incorrect behavior, it is discouraged, but there are situations in which it is crucial. More safely, the script can call functions in what is called the *proctable*. Each inset defines a set of procedures in this table, from which they are available to be called from Ness functions or to be bound to keystrokes or menu options as a user customizes his environment.

3. The String Algebra

Other than for string values, the Ness language is as simple and traditional as possible. The seven statement forms currently implemented provide for variables, functions, and flow-of-control, as sketched in Table 1 and illustrated in Appendix 1. Semicolons are optional between statements. Parentheses are used only for function call and expression nesting. Since it is implemented with ATK, programs may be typographically formatted for clarity. There are five types of data in Ness: **integer**, **real**, **boolean**, **object**, and **marker**. The first three of these are as in other common languages. An object value is a pointer to an object; it is principally of use for values that are to be passed to methods and proctable functions written in C.

variables

    declaration: *boolean* p, q   *real* x, y, z   *integer* i, j
    assignment: x := y + z   p := *not* q *or True*

function

    call: f(x, p, i)   sin(z)
    return: *return* x + y   *exit function*

flow-of-control: while-do, exit-while, if-then-elif-else

    while i < j *do*
        x := x * 2.5
        if x > y *then*
            exit while
        end if
        i := i + 1
    end while

Table 1. The seven statement forms in Ness. Other flow-of-control statements are planned for the future.

String values in Ness had to differ from those in other languages for several reasons. Most importantly, strings in any user-level language must be able to include full typography--fonts, indentation, italic, non-ASCII characters, and so on--as well as embedded objects. In a functional language it is also important that substrings be first class objects which can be passed as arguments and returned as values. In other programming languages it is difficult to write parsing and other string processing functions because substring references are clumsy. If substrings are not an integral part of the language, each substring reference needs three components: a reference to the underlying string, an indication of the start of the substring, and an indication of the length of the substring. It is possible to retain such values by keeping three separate variables; it is even reasonable to pass them as arguments to functions. Returning such a triple of values from a function, however, is awkward at best. The situation is so bad that it is difficult to see how to write a satisfactory string package as either a set of functions or a preprocessor for C; the algebra needs to be incorporated as a fundamental part of a language.

In Ness, **marker** values serve as string values. Each such value refers to a substring of some underlying base string. In particular, for documents one base string will be the document itself and a Ness script can refer to and modify an associated text via marker values. A formal algebra underlies marker values, as detailed in [2], where it is proven that the algebra is Turing equivalent. Within this algebra, constants and concatenation serve traditional roles: each returns a marker value for an entire, newly-created string. Five functions provide for all possible manipulations on strings: base(), start(), next(), extent(), and replace(). See table 2.

base(*s*) - returns a marker for the entire base string underlying *s*

start(*s*) - returns a marker for the empty string which starts where *s* does

next(*s*) - returns a marker for the single character which starts where *s* ends

extext(*s*, *r*) - returns a marker for the portion of the base between start(*s*) and start(next(*r*)); if the latter precedes the former, the value is an empty marker at start(next(*r*))

replace(*s*, *r*) - modifies the base string underlying *s* so the portion that *s* originally referred to will contain a copy of the value initially in *r*

Table 2. The five primitive operations in the string algebra.

The string algebra solves other problems in addition to convenience in writing string processing as function calls. The programmer need not be concerned with allocating storage for strings because that is handled by the system. Strings are not restricted to the array model found in some languages and programmers need not resort to integers, pointers, or some other non-string data type in order to refer to substrings. Strings also provide a data structuring form that may be more amenable to non-programmer computation than traditional programming constructs which are designed more for the convenience of hte machine than the human.

For data structuring, Ness markers provide not only string processing, but also all the capabilities of structures and arrays. A string is a structure when it has multiple objects embedded in it. It is an array when the embedded objects are all the same type and integer subscription functions are used to access the object. It should be noted that integer accessing does not reflect the majority of applications of arrays; in many applications an array is accessed sequentially, varying the subscript by one at each step. This corresponds to sequencing through a marker value with the *next*() function. For non-sequential access, however, it is trivial to write a function in the string algebra to access the i'th element of a marker value, see Algorithm 1. In practice, this algorithm is a primitive provided in the Ness system. (When it becomes common for users to store marker value objects within strings, garbage collection will be necessary. At the moment storage for the underlying strings is released when no markers refer to them.)

```
-- subscript(m, i)
--      Returns the i'th element of m.  If the length of m is
--      less than i, the function returns an empty marker
--      at the end of m.  If i is less than zero the function
--      returns an empty marker at start(m).
--
function subscript (m, i)
        marker s
        if i <= 0 then return start(m) end if
        s := next(start(m))  -- first element of m
        while extent(s, m) /= "" do
                -- we are not at the end of m
                if i = 1 then   return s   end if
                s := next(s)   -- next element of m
                i := i - 1
        end while
        -- we are at the end of m
        return start(next(m))
end function
```

Algorithm 1. The i'th element of a marker value. This example also illustrates typographical formatting of Ness code.

The desire for simplicity in Ness has led to a number of decisions to defer or not implement popular semantic tools. The two major such decisions are that Ness is not object oriented and functions are not first class objects. Ness does provide access to objects, but these must be written in C as augmented with the *class* preprocessor [6]. If functions were first-class, the **extend** functionality could be expressed as an assignment to an attribute of the object extended. During the design it was felt that this provision would make it possible to make more complex, less intelligible programs. For both functions and object-orientedness, only experience will suggest whether we need to augment the language.

Some observers have asked: should Ness be Lisp or be more like Lisp? One argument in favor of Lisp is that it has "less" syntax. However, this argument does not hold up when we notice that each "special form" in Lisp has its own unique syntax; and that there are many more special forms in Lisp than language constructs in Ness Another argument for Lisp is that it is a well known language. This is true, but the domain for Ness is so different that a Lisp programmer must learn a new set of functions anyway; this set of function names is a much higher hurdle to Ness than is the syntax of the language. A final argument for Lisp is that lists are a convenient data structure which is simple to learn. Ness counters this with the string algebra, which is just as powerful and may be even more intuitive for non-programmers.

4. Security

Embedding of scripts in documents does not introduce a new level of security problem, but makes more obvious a common security problem. The problem is that in small operating systems when I execute a program written by someone else it may do anything I myself may do; in particular, delete a file, modify a file, or send a copy of a file--say a forth-coming examination--to an interloper, perhaps a student about to take that examination. Since a Ness script is a program, and since it can do anything a user can, its execution is a security loophole.

Hypercard offers a security level scheme of a sort: users may choose to execute at one of five levels of privilege. However, these levels restrict the user from dangerous operations while not restricting scripts; a script may even reset the level itself. One reason this is not more of a problem in the Hypercard environment is that the equipment is less frequently connected to networks. However, stackware is shared and we can expect virus attacks via stackware in the future.

Some mainframe operating systems have implemented "capabilities", permissions that can be granted to limit the operations available to programs. These would ease the security problems, though they will still exist. Consider, for example, the user who offers a brand-new spiffy shell which gives graphical access to files. This shell will have to be given enough capabilities that it could be dangerous.

Ultimately the best and only protection is Trust; the reader must trust the person from whom he or she got a document. In a small closed community, such trust is an important factor in the free and open exchange of software. Unfortunately, the spread of networking is widening our communities and exacerbating the security problems.

The Ness implementation has features that make it more difficult--though by no means impossible--for a villain to damage an unwary user.[2] In particular, no script is ever executed--or even compiled--without permission from the reader. Users may choose among two options for this protection. The default option, automatically invoked for any user who has not chosen otherwise, is that the Ness script is surrounded with a text that describes the dangers of executing a script (see Appendix 2). The tail end of this text has buttons which allow the reader several options, including that of **Empower**ing the script, which compiles the script and activates any extensions it specifies. To be absolutely sure the user wants to empower the script, a click on the Empower button pops up a dialog box asking whether the user *really* intends to empower the script.

---

[2] It may be no surprise that despite considerable early design work actual implementation of Ness security began November 4, 1988, two days after the infamous Morris internet virus.

A villain should also be intimidated by the fact that Ness scripts are stored only in source form. The villain cannot know which readers will take the time to examine the script before empowering it, an examination which might ferret out any suspicious code. (Few readers will read scripts in their entirety, but enough will to provide a deterent.) Such examination of the code is aided by **Scan**, another option among the buttons at the end of the warning text. This option compiles the script, but generates an error message for each operation which might conceivably modify any of the reader's files, whether in memory or on the disk. Without artificial intelligence, this scan is forced to be quite paranoiac; it flags many statements which are completely harmless. Nonetheless, it typically selects less than a fifth of all statements.

More experienced readers may wish a direct approach to empowering scripts. They may specify in their personal preferences that they wish to see a dialog box instead of the warning text. Then whenever a Ness appears for the first time the reader is presented a dialog box which offers the same options as the buttons at the end of the warning text.

The necessity for security adds an unfortunate complexity. It would be preferable if users did not have to know about the script and the notion of empowering it. Worse, the requirement means that an author must position the script in such a way that it will be visible on the screen, because otherwise the reader will never see it to Empower it. This can clutter the design of applications with an unwanted element. In the future Ness and ATK will have mechanisms to reveal the script at the outset and later hide it.

Evaluation

This paper has shown that document extension has considerable potential for bringing the computer revolution to information delivery. It has described the Ness language which permits an author to construct a document with a variety of behaviors.

The first problem in defining the interconnection of a language embedded in a document is to identify those user events which initiate the operations described by the language. With the **extend** construct, Ness associates event handlers with named insets. If the inset signals appropriate events, they may be handled via the **on event** construct; otherwise the script can intercept user events destined for the inset with the **on menu**, **on mouse**, and **on keys** constructs.

Next the design must specify how the language can affect the document. One general tool is to allow the script to perform all possible user operations. In addition, Ness provides a full set of functions for manipulating insets, especially the text inset for which Ness provides a string algebra.

Finally, the design must provide some control so nefarious authors are not as free to produce programs which can damage readers' files. With Ness, the reader has the option to empower a script or not and also the Scan mode which aids in reviewing the script for potentially dangerous statements.

Although apriori it may seem that enhanced documents would be excellent for mail, they turn out not to be used in mail very much. The world of electronic mail is much more a world of short immediate messages than it is a world of carefully crafted communication. Plans for multi-media mail must satisfy the requirement for transmission of a variety of kinds of bulk information--including scripts--but this will not be the majority of the traffic.

A number of other lessons have been learned from this work:

o One can go quite far with static declaration of extends, events, and functions. Simple scripts for enhancing documents do not seem to need a highly dynamic language.

o Ness shows how to do extensions in a more comprehensible manner than scattering scripts behind each individual object. By giving names to objects they can be extended in the script and can serve as the targets of operations.

o The syntax of the extension language is far less a barrier to authors than is the size of the library of functions available.

o Strings can be dealt with functionally with the string algebra. However, the algebra is not as simple for non-programmers as could be hoped. The next step will be to define a pattern matching language to see if this can make clearer the description of string processing algorithms.

Ness is currently in daily use for maintenance of a data base of bugs and a bibliography. Over time, the number of applications will grow; these will serve as the basis for a future report.

References

[1] *Hypercard User’s Manual* Publication 030-3081-A, Apple Computer Inc. (Cupertino, Calif.) 1987.

[2] Wilfred J. Hansen, "The Computational Power of an Algebra for Subsequences", Information Technology Center, Carnegie-Mellon Univ., 1989.

[3] Clayton Lewis and Gary M. Olson, "Can Principles of Cognition Lower the Barriers to Programming?" Report on an informal workshop, University of Colorado, July, 1986.

[4] Thomas P. Neuendorffer, "ADEW: The Andrew Development Environment Workbench: An Overview", presented at the X Conference, Boston, MA, 1989.

[5] Andrew J. Palay, Wilfred J. Hansen, et al., "The Andrew Toolkit - An Overview", presented at the Usenix Conference, Dallas, TX, January, 1988.

[6] Paul G. Crumley, "The Andrew Class System", Information Technology Center, Carnegie Mellon University, file andrew/doc/Class.doc, 1989.

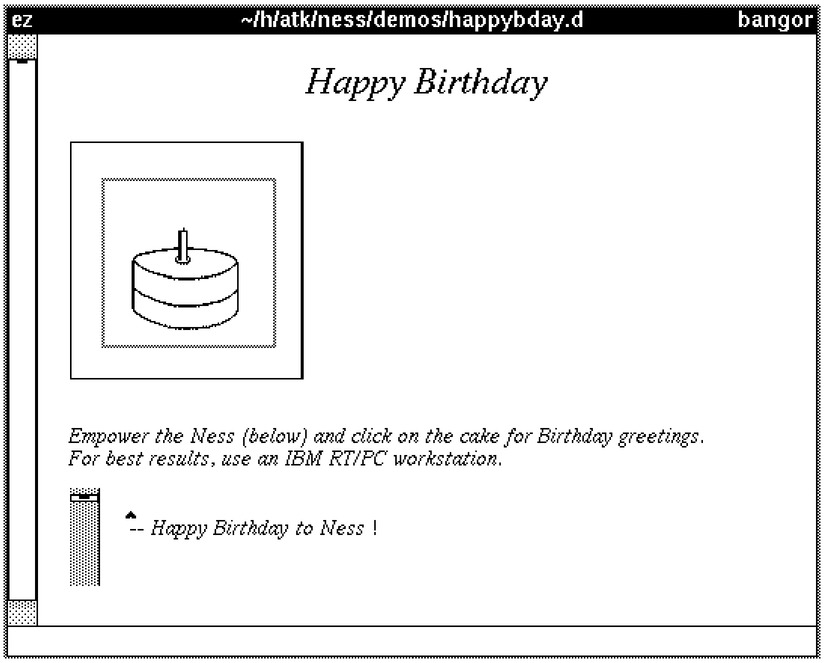
[7] Bruce A. Sherwood and Jill H. Larkin, "New tools for courseware production." *Journal of Computing in Higher Education*, vol. 1, no. 1, pp. 3-20, 1989.

[8] Ben Shneiderman and Greg Kearsley, *Hypertext Hands-On!*, Addison-Wesley, 1989.

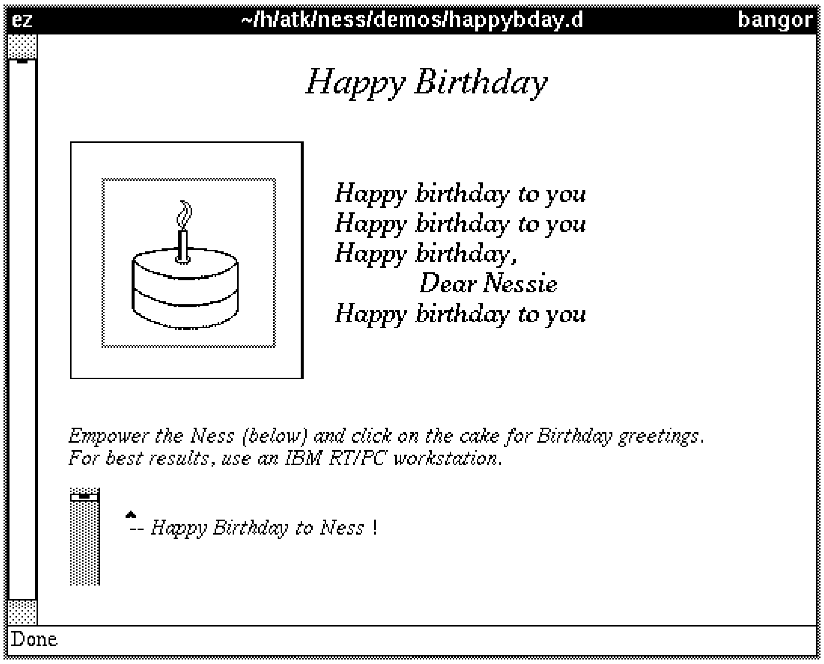
Appendix 1: A Ness extended birthday card

After empowering the Ness in the birthday card below, the reader can click the mouse on the cake; the card plays "Happy Birthday", shows the words, and lights the candle on the cake.

Before:



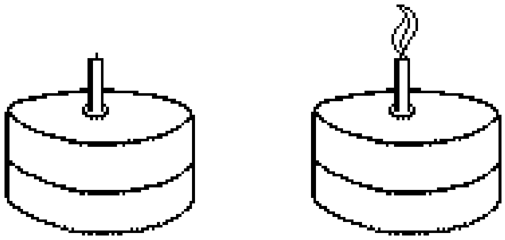
After clicking on the cake:



One the next page is the Ness script for the birthday card. The image area at the top of the card has two named insets: "visible cake" is the raster on the left and to its right is a text inset called "song text". The "visible cake" inset is extended so mouse clicks on it can be intercepted.

```
-- Happy Birthday to Ness !

boolean lit:= False     -- "visible cake" is initially unlit
marker Cakes := "
```



```
"

extend "visible cake"   on mouse "any"
        if mouseaction = mouseleftup then
                lit := not lit
                if lit then
                        showcake(FirstObject(
                                second(Cakes)))
                        sing()
                else
                        showcake(FirstObject(Cakes))
                        replace(base(currentselection
                                (inset("song text"))), " \n")
                end if
        end if
end mouse   end extend

function showcake(object cake)
        raster_copy_subraster(cake)
        raster_select_entire(inset("visible cake"))
        raster_replace_subraster(inset("visible cake"))
        raster_center_image(inset("visible cake"))
end function

function sing()
        marker m
        m := last(base(currentselection(inset("song text"))))
        m := last(replace(m, "\nHappy birthday to you\n"))
        im_ForceUpdate()
        play_notes("L7 CC L4 DCF    E P4")
        m := last(replace(m, "\nHappy birthday to you\n"))
        play_notes("L7 CC L4 DCG   F P4")
        im_ForceUpdate()
        m := last(replace(m,
                "\nHappy birthday,\n\tDear Nessie\n"))
```

```
    play_notes("L7 CC L4 >C <A  FED P4")
    im_ForceUpdate()
    m := last(replace(m, "\nHappy birthday to you\n"))
    play_notes("L7 A#A# L4 AF G  F.")
```
end function

In function showcake, the raster_xxx() functions are proctable entires defined by the raster inset. The copy/replace sequence copies the cake to the cut buffer and then replaces the visible cake with the new image.

The text inset named "song text" initially contains a blank and a newline. The expression currentselection(inset("song text")) returns a marker value for the currently selected portion of the text in this inset; this will usually be the empty string at the beginning. It is immaterial which marker value is returned because the base() function is applied to the value to get a marker for the entire contents of "song text".

im_ForceUpdate() causes the display to update to make visible the preceding change to the text.

The function play_notes() is inserted in the proctable by a package which interprets notes strings and plays them on the keyboard speaker in the IBM RT/PC.

Appendix 2: The Warning to novices

The warning text given below is wrapped around a Ness script when it appears in a document for user perusal. (The user may set a preference option to get a dialog box instead. Such a user is presumed to know what he or she is doing.)

NESS - This inset is a Ness script. If you choose the **empower** option at the end of this inset, the script may alter the behavior of this window. It may respond in new, useful, exciting, or bizarre ways to your mouse clicks, keystrokes, and menu selections.

Warning: Empowering a Ness script is just like running a program. The author of the script or program--if malicious--can write it in such a way that it can destroy your files. If you do not trust the place or person from which you got this script, DO NOT EMPOWER IT.

To learn what this script is supposed to do, you should read the surrounding document for a description. Or, you can read the script itself if you are familiar with Ness. After reading, you have four choices: do nothing, empower the script, "scan" it for potentially dangerous statements, or change it. The last three options appear after the script.

To learn about Ness, give the command ‘help ness‘ or see the files in /usr/andrew/doc/ness.

---- The Ness Script ----

<< the script is nested here >>

---- End of the Ness Script ----

Your Options

If you are uncertain whether to empower this script, the safest choice is to select NONE of the options below.

Also safe is the **Scan** option, which you can choose if you are familiar with the Ness language. The scan highlights each statement in the script which might conceivably change this file or other files. After choosing the Scan option, select the **Next danger** item on the **Ness** menu card to cycle through all the potentially dangerous statements.

If you know Ness and wish to modify the script, you can choose the **Author mode** option. If you do so, this help text surrounding the script will vanish and you will be able to edit the script. You can select the **Add warning** item on the **Ness** menu card to get this help text back.

The final option is to decide to **Empower** this script. To do so means that you trust the author of the script and the person who gave you this file; it also means you are aware that the script may change how the system responds to your actions.

Scan for dangerous statements
Author mode - Let me edit the script
Empower - I trust the source of this script