Long Term Preservation of Digital Information

Raymond A. Lorie
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
001-408-9271720
lorie@almaden.ibm.com

ABSTRACT

The preservation of digital data for the long term presents a variety of challenges from technical to social and organizational. The technical challenge is to ensure that the information, generated today, can survive long term changes in storage media, devices and data formats. This paper presents a novel approach to the problem. It distinguishes between archiving of data files and archiving of programs (so that their behavior may be reenacted in the future).

For the archiving of a data file, the proposal consists of specifying the processing that needs to be performed on the data (as physically stored) in order to return the information to a future client (according to a logical view of the data). The process specification and the logical view definition are archived with the data.

For the archiving of a program behavior, the proposal consists of saving the original executable object code together with the specification of the processing that needs to be performed for each machine instruction of the original computer (emulation).

In both cases, the processing specification is based on a Universal Virtual Computer that is general, yet basic enough as to remain relevant in the future.

Categories and Subject Descriptors

E.7 [**Electronic Publishing**]: Long-term preservation of digital information.

General Terms

Standardization, Languages.

Keywords

Digital library, preservation, archival, digital information, digital documents, emulation.

1. INTRODUCTION

The problem that libraries are facing today is well known [1]. For centuries, paper has been used as the medium of choice for storing text and images. Today more than ever, some of the archived objects (books, newspapers, scientific papers, government and corporate documents, etc.) are in danger of becoming unreadable. The preferred solution is to digitize the documents and store the binary files in a digital library (DL). As a result, an object can be copied repeatedly without degradation and its content can be sent remotely and accessed at will. Also, the physical space needed to store the object becomes smaller and smaller as storage density increases.

Beside digitization, a high percentage of the data to be preserved is, today, generated directly in digital form. Spreadsheets, word processor documents, e-mail messages, as well as audio CD's or DVD's are obvious examples.

Whatever the origin of the digital information is, we are left with the same challenge: to ensure that the information can survive long term changes in storage media, devices, and data formats. An excellent introduction to this problem is given in [2].

2. THE PROBLEM

Suppose we use a computer (identified as M2000) to create and manipulate digital information today. For the purpose of archiving the data for preservation, the digital information is catalogued in a DL index; the content of the document may be stored in the DL itself, say in a file F2000. Suppose that, in 2100, a client wants to access the archived data. Assuming the catalog entry is still accessible and still refers to the document, three conditions must be met in order to recover its content:

1. F2000 must be physically intact (bit stream preservation)
2. A device must be available to read the bit stream.
3. The bit stream must be correctly interpreted.

Condition 1: some researchers predict very long lifetimes for certain types of media, but others are much less optimistic. Anyway, if a medium is good for N years, what do we do for N+1 years? Whatever N is, the problem does not go away. The only possible solution consists of rejuvenating the information periodically by copying it from the old medium onto a newer one.

Condition 2: machines that are technologically obsolete are hard to keep in working order for a long time. Actually, this condition is more stringent than the previous one. Here also, rejuvenation is needed: it moves the information onto a new medium that can be read by the latest generation of devices. Thus, conditions 1 and 2 go hand in hand. Note that rejuvenation is not an overhead incurred only for preservation; it is also a means of taking advantage of the latest storage technology.

Condition 3: the two conditions above insure that a bit stream saved today will be readable, as a bit stream, in the future. But one must be able to decode the bit stream to recover the information in all its meaning. This is quite a challenging problem, and this paper sketches a solution.

3. THE TYPES OF DOCUMENTS

We distinguish three cases; by increasing order of complexity:

Case 1: The document is represented as a simple data structure. The rules to decode and understand the document can be explained in natural language, and saved in the DL. In 2100, a program can be written to decode the data[1].

Case 2: When the data structure and the encoding reach a certain level of complexity, it becomes impractical or even impossible to explain them in natural language. The alternative is to save a *program* that decodes the data; this may be the only way to be sure that the decoding is specified completely. The program is written in some language L. In 2100, the M2100 system must be able to interpret it.

Case 3: At the end of the spectrum, we may be interested in archiving a computer program or system for its own sake. In this case, it is the ability to run that program which must be preserved by the archiving mechanism. Not only the bit stream that constitutes the program must be archived, but we must also make sure that the code can be executed at restore time. Therefore, enough information on how to execute an M2000 code on an M2100 machine must also be archived.

The overall case analysis can be simplified by noting that case 1 can be handled by any method that handles case 2. In both cases 2 and 3, a program is being saved. The only – but very significant - difference is that, for case 3, the language in which the program is written must be the native M2000 machine language, while the language L, used for handling case 2, can be arbitrary. We refer to case 2 as *data archiving* and to case 3 as *program (or behavior) archiving.*

4. PREVIOUSLY PROPOSED SCHEMES

Two schemes have been proposed earlier; both have serious drawbacks.

4.1 Conversion

Conversion [1] has been used for decades to preserve operational data in data processing installations. When a new system is installed, it coexists with the old one for some time, and all files are copied from one system to the other. If some file formats are not supported by the new system, the files are converted to new formats and applications are changed accordingly.

Conversion is quite reasonable when the information that is being converted will, most likely, be used in the near future (a bank account record, for example). However, in the type of archiving that interests us here, the information may not be accessed for a long time, potentially much longer than the period between two system changes. Then, the conversion becomes a burden and is not really necessary. Another disadvantage of conversion is that the file is actually changed repeatedly – and it is hard to predict the cumulative effect that such successive conversions may have on the document.

[1] A piece of text encoded in a well known alphabet such as ASCII is a particularly easy instance of a case 2 document; the only requirement is that the DL catalog keep the alphabet definition.

4.2 Emulation

In [2], Jeff. Rothenberg sketched out an overall method based exclusively on *emulation.*

In summary, it consists of saving, together with the data,

- the original program P, also as a bit stream, that was used to create and manipulate the data (this program runs on M2000), including the operating system and other components when necessary,
- the detailed description of the M2000 architecture,
- a mostly textual description of how to use the program P and what its execution returns.

Then, in 2100, an emulator of the M2000 machine can be built, based on the architecture description. Once this is done, the program P can be run and will produce the same results that P used to produce in 2000. Let us point out right away that building an emulator from the description of the M2000 architecture is not a simple endeavor. It can be done only if the description of the M2000 architecture is perfect and complete (a notoriously difficult task in itself). But even then, how do we know the emulator works correctly since no machine M2000 exists for comparison? In [3], the same author suggests the use of an *emulator specification*. Although no particular specification means is proposed, we can imagine that these specifications could be prepared in 2000 from the M2000 architecture, facilitating the actual generation of an emulator in 2100 (by a human or a machine).

Note that the method hinges on the fact that the program P is the original executable bit stream of the application program that created or displayed the document (including the operating system). This is justifiable for behavior archiving but is overkill for data archiving. In order to archive a collection of pictures, it is certainly not necessary to save the full system that enabled the original user to create, modify and retouch pictures. If Lotus Notes is used to send an e-mail message in the year 2000, it is superfluous to save the whole Lotus Notes environment and have to reactivate it in 2100 in order to restore the note content. But there is an even worse drawback: in many cases, the application program may display the data in a certain way (for example, graphically) without giving explicit access to the data itself. In such a case, it is impossible to move the actual data from the old system to the new one.

Other authors have voiced a similar reservation. For example, D. Bearman in [4] notes that “Rothenberg is fundamentally trying to preserve the wrong thing by preserving the information system functionality rather than the record”.

5. OUR PROPOSAL

One important characteristic of the method introduced in [5] and summarized below, is that it differentiates between data archiving which does not require full emulation, and program archiving which does.

For data archival, we propose to save a program P that can extract the data from the bit stream and return it to the caller in an *understandable way*, so that it may be transferred to a new system. The proposal includes a way to specify such a program, based on a Universal Virtual Computer (UVC). To be *understandable,* the data is returned with additional information, according to the metadata (which is also archived with the data).

For the archival of a program behavior, emulation cannot be avoided. But here also, the Universal Virtual computer can be used to write the M2000 emulator in year 2000, without any knowledge of what M2100 will be.

5.1 Data Archival

Consider fig.1. The data contained in the bit stream is stored in 2000, with an arbitrary internal representation, Ri. In 2100, The data is seen by a client as a set of data elements that obey a certain *schema* Sd, in a certain *data model*. A decoding algorithm (*method*) extracts the various data elements from Ri and returns them to the client, tagged according to Sd. A language L is used in 2000 to specify the details of the needed decoding. In addition, a mechanism allows the client to read Sd as if it were data. It relies on a schema Ss, a schema to read schemas. The schema Ss is simple and intuitive; it should endure for a long time to come, and be published in many places so that it remains known. In the following sections, we describe each component in more detail.

5.1.1 The logical data model

The choice of an appropriate data model and a means to describe Sd is based on the following premises:

1) it must be simple in order to minimize the amount of description that must accompany the data and decrease the difficulty of understanding the structure of the data;

2) it is only used to restore the data and not to work with it (actually, once it is restored, the data will generally be stored in the repository of the system used at restoration time, maybe under a different model).

Flat files, or tables similar to those used in the relational model, certainly satisfy the requirement. So do hierarchies. Simple and powerful, inherently easy to linearize, they have been used in many areas, often under a different name: *repeating group*s or *non-first-normal-forms* [6] in databases, *Backus-Naur Form* [7] in syntax specification, and - more recently - *XML* [8]. We choose an XML-like approach.

5.1.2 An example

Consider a file containing a collection of pictures of historical buildings in Mycity, including both formatted data and gray scale pictures. The file is a list of records. Each record consists of a sequence of fields. Each field can itself be a list of records made of fields that can be records, etc. A table showing the populated hierarchical structure is shown, much abridged, in fig. 2.

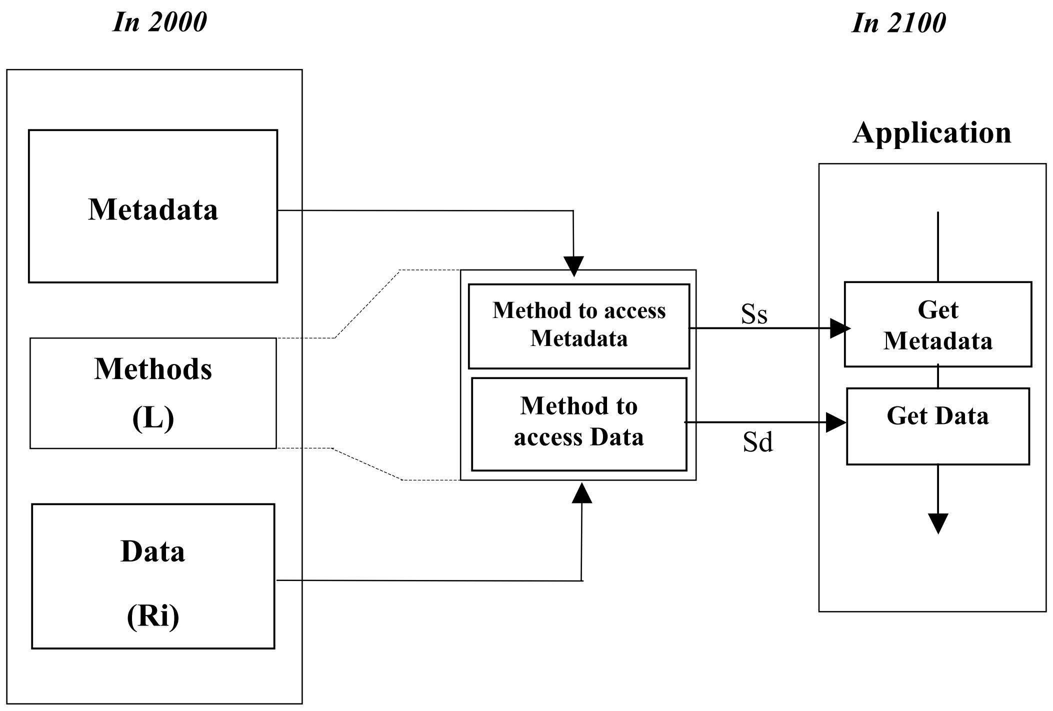

Figure 1: Overall Mechanism for Data Archival

building						
name	address	picture				
		year	nbr_lines	dots-per-line	line	
					No	gray_value
ABC Building	12 Main street	1903	1200	2100	1	102 104 116 ...
					2	211 234 ...
		1924	900	1300	1	125 ...
XYZ Building	9 North street	1917	2180	2700	1	202

Figure 2: Populated table for a Collection of buildings

A reader is able to understand what the various data elements mean because the header, displayed at the top of the table, describes the schema Sd. In addition, the indentation of the data allows the reader to "parse" the data according to the schema. For the digital equivalent of the data in fig. 2, a representation of Sd is also needed. The proposal consists of storing, together with the data, a representation Ri of Sd. That representation is nothing else than the linearized form of a construct similar to a Data Type Definition (DTD) in XML; it defines the application-dependent tags. The DTD for the application is shown in fig. 3.

In plain English, it would read as:

> a collection is a list of buildings; a building is associated with an address, a name, and a list of pictures; a picture is associated with a date (year), the number of dot lines in the picture, the number of dots per line, and a list of lines; a line has a number and a list of gray values.

The * token stands for 0 or more; + means at least 1; ? means optional.

The DTD is the metadata to understand the data (it is clearly application-dependent). At this point, let us assume that the client knows the DTD.

```
DOCTYPE  Building_collection [
  ! This is a collection of gray scale pictures of historical buildings
     in Mycity.
  ! A building has an address, and an (optional) name; it can have
     several pictures (for different years).
  ! The gray value is between 0 (white) and 255 (black).
     ELEMENT Collection (building+)
     ELEMENT building (name?, address, picture+)
     ELEMENT picture (year, nbr_lines, dots-per-line, line+)
     ELEMENT line (nbr, gray_value+)
     ELEMENT name (CHARr)
     ELEMENT address (CHAR)
     ELEMENT year (NUM)
     ELEMENT nbr_lines (NUM)
     ELEMENT dots_per_line (NUM)
     ELEMENT nbr  (NUM)
     ELEMENT gray_value (NUM)
```

Figure 3: The DTD for our application

5.1.3 Invocation and functionality of the methods

The method to access the data supports the retrieval of all values in the tree according to a depth-first traversal. At a logical level, the restore application in 2100 simply executes the following (pseudo-code) statements:

```
open
while (more) {
        get_field (tag, x)
}
```

For each field, the value is returned in variable x, together with a <tag>. The tags, although slightly different from those used in XML, unambiguously indicate the type of each element. In the example, the first repetitive calls to get_field would return the following:

```
<Collection>
    <building>
        <name>              ABC_Building
        <address >          12  Main street
        <picture>
            <year>          1903
            <nbr_lines>     1200
            <dots_per_line> 2100
            <line>          …
```

5.1.4 A schema to read schemas

Contrary to what we assumed earlier, the client may not know the information contained in the DTD. Therefore, we need to provide a way to retrieve that information as well. A simple solution consists of adopting for the schema a method similar to the one proposed for the data: the schema information is stored in an internal representation Ri, and accompanied by a method to decode it. Logically, the Ss looks like this:

```
DOCTYPE  Metadata [
    ELEMENT fields (root_name, comment, field+)
    ELEMENT field  (level, name, description, type?, attribute?)

    ELEMENT root_name (CHAR)
    ELEMENT comment (CHAR)
    ELEMENT level (NUM)
    ELEMENT name (CHAR)
    ELEMENT description (CHAR)
    ELEMENT type (CHAR)
    ELEMENT attribute (CHAR)
]
```

The *level* specifies the depth of a record in the hierarchy. The same code "open... get_field..." can be used to retrieve the metadata. Fig. 4 shows the initial section of the results.

The mechanism presented above accomplishes the following: it defines a simple interface for accessing the archived data. That interface is simple because the decoding rules are all contained in the methods; it will therefore easily survive for a very long time (its definition may have to be stored in more than one place but it certainly does not need to be stored with each archived object). The same is true for the mechanism to read schemas.

It should be noted that the examples used above are oversimplified, but they are sufficient for illustrating the proposed mechanism. It is clear that the DTD's will need identifiers and references such as those available in XML.

5.1.5 Specification of methods

The responsibility of extracting the logical data elements from the data stream lies with the methods, supposedly written in L. But what should L be? Let's try some possibilities:

1. A natural language. The difficulties are well known; and computer scientists have invented all kinds of codes and pseudo-codes to avoid them, leading to the next item:

2. A high level language. High-level languages are designed to facilitate the writing of programs by large communities of programmers. Language developers, then, always try to incorporate the latest features that may facilitate program development. Every five or ten years, something new seems to come along and the current language gets obsolete.

3. The machine language of the computer on which the algorithm runs in 2000. This is the option that requires a full emulation of the M2000 to be written at restore time; we have discussed its difficulties earlier in this paper.

```
<fields>
        <name>                      Collection
        <comment>                   this is a set of pictures of historical buildings
        <field>
                <level>             0
                <name>              building
                <description>       list of building(s) which have pictures
                <attribute>         +
        </field>
        <field>
                <level>             1
                <name>              name
                <description>       name of the building
                <type>              CHAR
                <attribute>         ?
        </field>
        <field>
                <level>             1
                <address>           postal address of building
                ………
        ………
        ………
</fields>
```

Figure 4: Retrieving a schema definition (partial results)

Instead, we propose to describe the methods as programs written in the machine language of a ***Universal Virtual Computer (UVC).*** The UVC is a Computer in its functionality; it is Virtual because it will never have to be built physically; it is Universal because its definition is so basic that it will endure forever.

The UVC program is completely independent of the architecture of the computer on which it runs. It is simply interpreted by a UVC Interpreter. A UVC Interpreter can be written for any target machine.

This approach does not have the drawbacks of the method 3 above. If a UVC program is written in M2000, it can be tested on a UVC interpreter written in 2000 for an M2000 machine. If x years later, in 2000+x, a new machine architecture comes up, a new UVC interpreter can be written. For quality control, a set of UVC programs can be run through both the 2000 and 2000+x interpreter, and should return exactly the same sequence of tagged data elements. Also, a flaw in the interpreter will never damage any archived document; a programmer may simply have to refer to the UVC specifications to fix the problem. Actually, it is safe to assume that the source code used to implement the year 2000 interpreter can be used as the base for developing the 2000+x version. Also, the same source can be compiled for various target computers, still decreasing the size of the task.

In addition, the UVC can be very simple - and at the same time very general, so that writing an interpreter at any time remains a simple task, far from the complexity of writing a full machine emulator.

5.1.6 The UVC Architecture

The details of the UVC specification are not important at this point. Clearly, its architecture may be influenced by the characteristics of existing real computers or virtual machines developed for different purposes, such as Java. What is important is that it does not need to be implemented physically. Therefore there is no actual physical cost. For example, the UVC can have a large number of registers; each register has a variable number of bits plus a sign bit. The UVC has an unlimited sequential bit-oriented memory. Addresses are bit-oriented (so that a 9-bit "byte" computer can be emulated as easily as an 8-bit one). Also, speed is not a real concern since M2100 will be much faster and these programs are run mostly to restore the data and store them in an M2100 system, and a small set of instructions is sufficient. This reduces the amount of work involved in developing an interpreter of the UVC instructions onto a real M2100 machine.

The fact that the instruction set is kept to a minimum may complicate the writing of a program at the machine instruction level. But, as in any RISC machine, it is anticipated that the methods will be coded in some high level language and automatically compiled onto UVC instructions.

5.1.7 The UVC Interface

In 2100, a machine M2100 will come with a restore program that will read the bit stream in a virtual memory and then issue requests to the UVC Interpreter. The interface must be independent of the conventions used in 2000 or 2100. It uses software registers, filled with single values (of elementary types), according to the following list:

- Reg 0: an integer (k) indicating which function is being invoked.
- Reg 1: the completion code returned by the function.
- Reg 2: a pointer *p-data*, pointing to the data bit stream.
- Reg 3: a pointer *p_out* to some memory set aside to receive a data element.
- Reg 4: a pointer *p_tag* to some memory set aside to receive the tag of the data element.
- Reg 5: a pointer *pw* to a working area.

There is a single entry point to the beginning of the UVC code for the methods. That code will branch to the appropriate method depending upon the value k in Reg 0.

5.1.8 Highlights of the approach for data archiving

The use of the UVC gets rid of the need for a full machine emulator. It also eliminates the need for agreeing on standardized formats. Anybody who wants to preserve a file can use any format but must make sure that the UVC method is supplied to interpret the format. The only standards needed are now the UVC and the data model for data and metadata; they are simple enough to endure. Only the UVC interpreter will have to be written (or re-compiled) when a new machine architecture emerges. There is no impact on the archived information.

5.2 Program Archival

When the behavior of a program needs to be archived, the M2000 code must be archived and later emulated. If the program is only a series of native instructions of the M2000, it may not require the saving of any other package or operating system. However, if the object is a full-fledged system with Input/Output interactions, then the operating system must be archived as well.

We have mentioned earlier the difficulties implied in writing emulators in the future. The UVC approach can be naturally extended to support the archiving of programs, providing for a way to essentially write the emulator in the present. Instead of archiving the UVC method to decode the data, the actual M2000 program will be archived, together with UVC code that emulates the instruction set of M2000. This time, in 2100, the UVC interpreter will interpret that UVC code; that interpretation will then yield the same results as the original program on an M2000. This suffices if the program does not have any interaction with the external world (Input/Output operations or interrupts).

Things get more complicate when Input/Output operations are involved. Suppose the program prints a black/white document on an all-point-addressable printer. The program somewhere issues a Start I/O operation with some data. Clearly the execution of that instruction is not part of the M2000. The M2000 only sends the data to an output device processor P which computes an output-oriented data structure S (such as a bit map), and sends it to the last process, the one that actually prints the page. Our proposal for extending the method to support such operations is as follows.

In addition to archiving the UVC program that interprets the M2000 code, another UVC program that mimics the functioning of P must also be archived. It will produce the structure S. It is impossible to anticipate in 2000 the output technology that will exist in 2100. But, if S is simple and well documented, it will be relatively easy to write in 2100 a mapping from S to the actual device. For an all-point-addressable B/W printer, S is simply a bit map. The bit map becomes the interface to an abstract printer, independently of what the new technology will be. This technique, again, ensures that the difficult part (which depends heavily on the details of the device) is written in 2000 when the device exists. It can be fully tested in 2000 by mapping the abstract device into a 2000 device.

Abstract devices must be similarly defined for sequential tapes (with operations such as R, W, Rewind, Skip), for random access storage units (R, W at a particular record address), for sequential character output or input (screen, keyboard), for x/y positioning (mouse, touch-screen, cursor), etc.

6. SUMMARY AND CONCLUSIONS

In this paper, we analyzed the challenges of archiving digital information for the very long term.

We made a distinction between the archiving of data and the archiving of a program behavior.

The same technique is used to solve both problems: both rely on a Universal Virtual Computer. For archiving data, the UVC is used to archive methods which interpret the stored data stream. For archiving a program behavior, the UVC is used to specify the functioning of the original computer.

In summary, for data archiving:

1) In 2000, whoever creates a new data format needs to produce a UVC program to decode the data. For at least one platform, a UVC interpreter must be developed. It can be used to test the correctness of the UVC program.

2) In 2100, every machine manufacturer needs to produce a UVC interpreter.

For program emulation,

1) In 2000, for each platform, the manufacturer needs to provide an emulator of M2000 written as UVC code. Manufacturers of devices in 2000 need to provide the UVC code that emulates the device control unit.

2) In 2100, every machine manufacturer needs to produce a UVC interpreter, and every manufacturer of an I/O device needs to produce an implementation of the abstract device on the real 2100 device.

What the proposed method accomplishes is to provide a reliable framework where preparatory work can be done in 2000 - when the information is well known - rather than in 2100 when the difficulty would be much greater. It also avoids the cumbersome need for defining standards under which the data should be stored. These standards would have to be defined for all types of applications, and would have to remain valid for centuries; this is just unpractical. Instead, the proposed solution replaces the need for a multitude of standards (one for each format) by a single standard on the UVC method. That standard should cover: the UVC functional specifications, the interface to call the methods, the model for the schema and for the schema to read schemas. Each of these components can be kept general and simple enough to remain relevant in the future.

In this paper, we couch the preservation issue in the framework of a digital library. The proposed solution calls for the archiving of the data bit stream, some UVC code, and some metadata describing the data schema and the interface to invoke the methods. A DL system generally contains two databases: the one that contains the metadata and the one that contains the data itself (the archive store). Although it is a matter of choice, we suspect all of the items above will be kept in the archive store. But the meta-database may be used to store some information necessary to bootstrap the restore process (for example, we need to know the alphabet before we start reading any text). The longevity of the meta-database can be ensured by migration and that information will therefore remain available.

Nevertheless, another environment may be worth considering, in which the document is not part of a digital library. Then, the whole information is stored on a removable storage object such as CD-ROM or tape and needs to be restored in a distant future by using only information that *it* contains. The technology presented here remains applicable, with relatively minor additions to solve the bootstrap problem.

It would be naive to think that solving the archiving problem is simply a technical challenge. For example, the success of any effort would hinge on a minimal agreement of all parties involved in generating new technologies or creating new types of data. But this cannot happen before a certain level of technical know how is reached. Thus, it is important for the computer science community to start developing the technology, and the purpose of this paper is to document some initial ideas. Our research project is currently investigating design issues and developing an early prototype to prove the validity of the concepts and evaluate our design decisions. The "real life" aspects of our current work are provided by a joint study with the Koninklijke Bibliotheek, the national library of the Netherlands, The Hague.

7. ACKNOWLEDGMENTS

Robert Morris, then Manager of Data Systems Technology at the IBM Almaden Research Center, brought the importance of the problem to the attention of the author. His encouragements and those of Robin Williams are gratefully acknowledged. Many thanks go to Henry Gladney for many lively discussions on the subject.

8. REFERENCES

1. Waters, D, and Garret, J.: Preserving Digital Information. Report of the Task Force on Archiving of Digital Information, Commission on Preservation and Access and the Research Libraries Group, Inc., May 1996.
2. Rothenberg, J. Ensuring the Longevity of Digital Documents. Scientific American, 272(1), January 1995.
3. Rothenberg, J.: Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation. A report to the Council on Library and Information Resources, January 1999.
4. Bearman, B. Reality and Chimeras in the Preservation of Electronic Records, D-Lib Magazine, Vol. 5, No 4, 1999.
5. Lorie, R.: Long Term Archiving of Digital information, IBM Research Report RJ 10185, July 2000.
6. Dadam, P. & al.: A DBMS Prototype to support Extended NF2 Relations, ACM SIGMOD, May 1986.
7. Aho & al.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986
8. Harold, E.R. XML, Extensible Markup Language. IDG Books Worldwide, 1998.