LONG-TERM ARCHIVING OF DIGITAL INFORMATION

Raymond A. Lorie

IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, California 95120-6099

ABSTRACT

The preservation of digital data for the long term presents a variety of challenges. First and foremost, there is the technical challenge. How to avoid loosing data because of changes in the storage medium, devices, and data formats? How to be able to run a viewer in the future for an archived multimedia file, or a computer-aided design system, or even a popular game? Then, there are social and behavioral aspects. Which information needs to be archived, who decides? What about intellectual property, proof of authenticity, etc.? Clearly, all aspects will require much work before full solutions may be proposed and implemented. This paper focuses solely on the technical aspects of the problem, and more precisely, on how to interpret a bit stream that has been successfully archived and later retrieved.

A distinction is made between archiving a data file and archiving a program (so that its behavior may be reenacted in the future), and a research direction is proposed for both problems, based on the same basic mechanism.

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 need to be 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.

Both processing specifications are based on the use of a Universal Virtual Computer that is general yet basic enough as to remain relevant in the future.

1. Introduction

The problem that libraries are facing today is well known. 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. What should be done to protect their contents? They could essentially be copied (on paper or microfilm) or digitized. Digitization through a digital camera or a scanner replaces the image by a bit stream. This offers many advantages. First, the object can be copied repeatedly without degradation (even if some physical degradation occurs, redundancy can be used to correct the problem); its contents can be sent remotely and can be accessed at will. Finally, 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. Spread sheets, word processor documents, e-mail messages, as well as musical CD's or DVD movies are obvious examples. The same is true of many engineering designs who were described as blueprints in the past and now exist as digital information in a Computer Aided Design system with multimedia, relational database, and virtual reality.

In the future, the volume of digital information will increase substantially and dwarf the volume of the existing paper information. Thus, it makes sense to digitize what needs to be saved of the past, and concentrate on the single problem of preserving digital information for posterity. An excellent introduction to this problem is given in [0].

2. The problem to be solved

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 stored on a removable medium, say D2000 (most probably some kind of disk). Suppose that, in 2100, somebody (the client) wants to access the data saved today. What mechanism should exist to be able to satisfy the request?

Four conditions must be met:

1. The particular D2000 disk must be found.
2. D2000 must be physically intact (or at least intact enough - depending upon the redundancy).
3. A machine must be available to read the raw contents (bit stream) of D2000.
4. The bit stream must be correctly interpreted.

Condition 1: In digital libraries [2] [3], a digital object is registered under a certain name, the document itself (the file) is stored in a safe place, and its bibliographical data are saved in a catalog (the database). The database also contains a "pointer" that points to the file. This paradigm mimics the organization of current libraries. But, exactly as it is possible to find a book outside a library and still understand it, there is also a requirement for being able to "understand" a diskette or CD-ROM using exclusively data that are contained in the object itself.

Condition 2: 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 about preservation for N+1 years? Whatever N is, the problem does not go away. There really seems to be only one solution to this problem: to copy the information periodically to rejuvenate the medium.

Condition 3: 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 2 and 3 go hand in hand. It must be noted that rejuvenation is not simply an overhead for preservation; it also takes advantage of the latest storage technology.

The three conditions above ensures that a bit stream saved today will be readable, as a bit stream, in the future. But there is still the...

Condition 4: 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 possible approach.

3. The contents of a digital object

A digital object generally corresponds to what we designate today as a file. Objects can vary greatly in complexity, forming a wide spectrum.

Case 0: It is the simplest case. The data object is readily understandable by a human reader. The only practical example is a piece of text encoded in a well known alphabet such as ASCII. This is clearly easy to archive - we only need to know the alphabet.

Case 1: The data has to be decoded in some way by the reader or by a machine (assuming one knows the decoding rules). In the latter case, a program must be written in 2100 to decode the data, based on the stored description; the program must also be able to convey to the (human) reader information on what the data (or individual portions of the data) mean (the metadata). An image, a digital video clip, a table with ASCII fields, are all examples of such data objects. Essentially, the algorithm is simple and amenable to a description in plain English. Writing the program in 2100 should not be very difficult, whatever the programming language is at that time.

Case 2: When the algorithm complexity reaches a certain level, it may become impractical to explain it 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; M2100 must be able to understand that language and interpret it.

Case 3: At the extreme 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 that 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. If you want to preserve the look and feel of Window 95 or MAC, or the user interface of a Computer Aided Design system, the only solution is to archive the whole body of code used during the execution, and enough information on how to run the code at restore time. Programs with complex interactions between the program and a user fall in this category.

The overall case analysis can be simplified by noting that case 1 can be handled by any method used to handle case 2 (the program will only be simpler). The same is true for case 0 (the program will be practically empty).

In both cases 2 and 3, a program is being saved. The only difference is that, for case 3, the language in which the program is written must be the native M2000 machine language. This is because what is being archived is not only the data but the behavior of the program as well. We call this approach *behavior archiving*. The approach for Case 2 is called *data archiving*.

4. A previous proposal

In [0], J. Rothenberg sketched out an overall system organization based on encapsulating everything needed to decode the information when needed.

In summary, he proposes to store in an encapsulated object:

A. a description of the alphabet used to store text;
B. a mostly textual description of the metadata (the semantic of the stored data);
C. the data as a bit stream
D. the program, also as a bit stream, that was used to store and manipulate the data (this program runs on M2000), including the operating system and other necessary components.
E. The detailed description of the M2000 architecture.

In [4], it is suggested that an *emulator specification* be saved in E as an alternative, to facilitate the creation of an emulator in the future.

In any case, in 2100, the client would have to construct an emulator based on the information in E. Then, by reading B, he would learn how to run the program D on the data C, and how to understand the information returned by its execution.

Note that the method hinges on the fact that what is saved in D 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, is it necessary to save the full system that enables the original user to create, modify, retouch pictures when only the final result is of interest for posterity. If Lotus Notes is used to send an e-mail message in the year 2000, is it necessary to save the whole Lotus Notes environment and reactivate it in 2100 in order to restore the note contents? But there is an even worse drawback: in many cases the application program may display the data in a certain way (for example, a graphical representation) without giving explicit access to the data themselves. In such a case, it is impossible to get the basic data out of the old system and into the new one.

Other authors have voiced a similar reservation. For example, D. Bearman in [5] notes that "Rothenberg is fundamentally trying to preserve the wrong thing by preserving the information system functionality rather than the record".

In the remainder of this paper, we propose a method that handles differently the archiving of a program behavior and the archiving of data files.

For data archiving, what is stored in D is simply a program that can extract the data out of the encapsulation and returns them to the caller in an *understandable way*, so that they may be transferred to the new system. The proposal includes a way to specify such a program, based on a Universal Virtual Computer interface (UVC). To be *understandable*, the data is returned with additional information, according to the metadata also stored in the encapsulation.

For the archiving of a program behavior, emulation cannot be avoided. 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 [4], what is saved is an *emulator specification*. Although no particular specification means is proposed, it suggests 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). Our proposal is to write the emulator in 2000 using the UVC. Only a UVC interpreter needs to be written in 2100.

5. The proposal

We first present and evaluate the proposal for the archiving of data, then for the archiving of a program behavior.

5. 1. Data Archiving

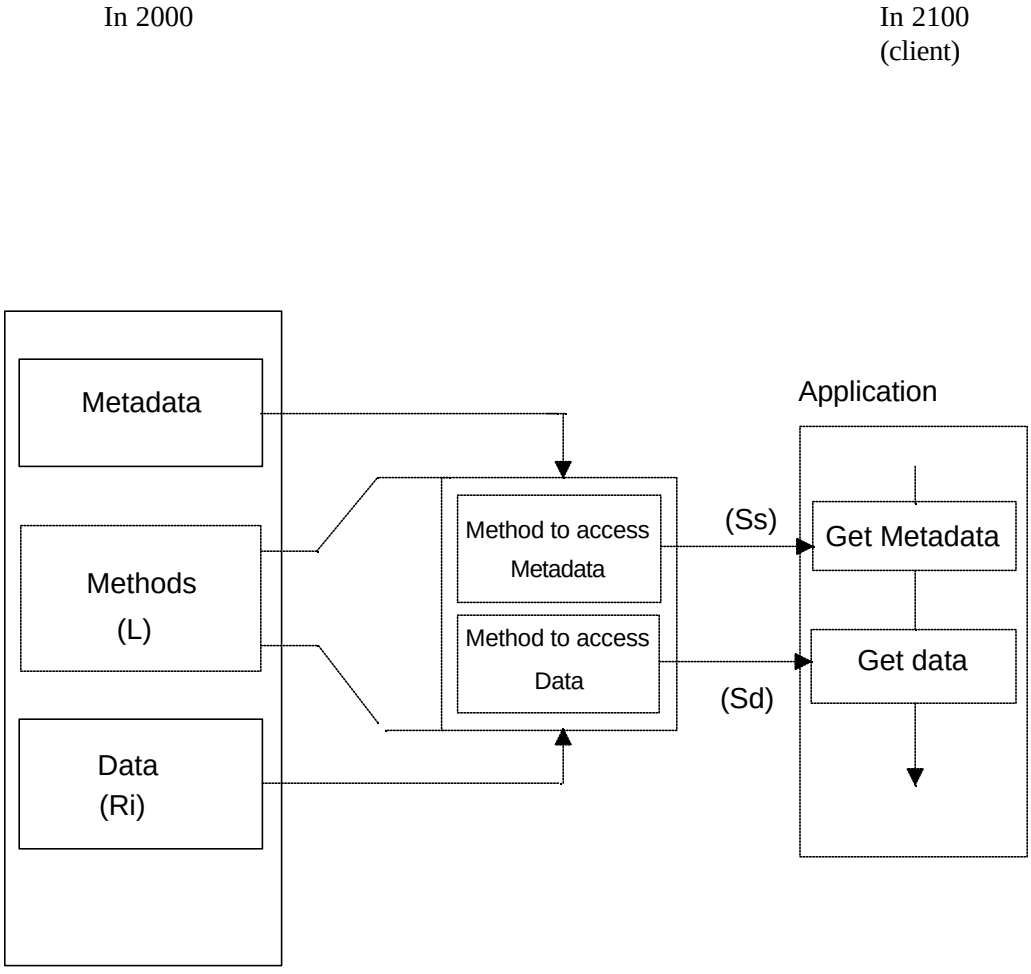


Figure 1: Overall mechanism for data archiving

Consider fig.1. The data contained in the bit stream are stored according to an arbitrary internal representation, Ri. The data are 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 to specify the details of the algorithm. In addition, a mechanism allows the client to read Sd as if it were data. It relies on a schema Ss. Ss is a schema to read schema's; it should be very basic and endure for a long time to come.

The encapsulated object now contains the components A and C identical to the ones described in section 4. B contains the schema and a description of the interface to the methods. D contains the methods, written in L. E is not needed anymore.

5.1.1. The logical data model

The choice of an appropriate data model is based on the following premises:

1) it must be simple in order to minimize the amount of description that must accompany the data as well as 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).

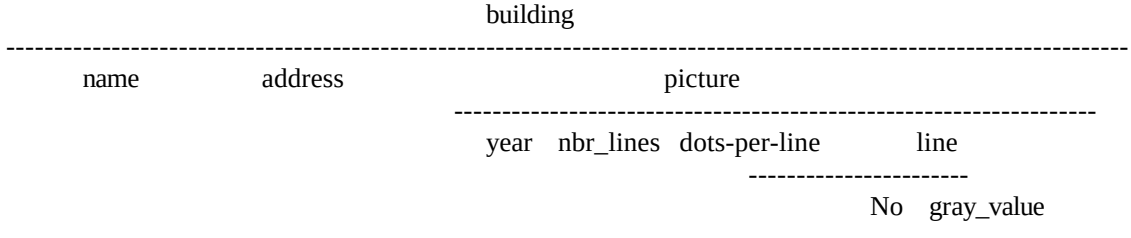
Since the bit stream is a linear structure, the data model should also be linear so that its mapping onto the bit stream remains simple. Flat files, or tables similar to those used in the relational model, certainly satisfy that requirement. But so do hierarchies - at least along one single hierarchical path. Hierarchical models with their inherently easy linearization 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].

Consider a file containing a collection of pictures of historical buildings in Mycity. 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 in fig. 2.

Definition of the schema

Why is a reader able to understand what the various data elements in the table mean? The header displayed at the top of the table plays an important role; it describes the organization of the data, the *schema*. In addition, the indentation of the data allows the reader to “parse” the information according to the format specified by the header. In the stored data, the separation of the data elements can be implemented in many ways. On the other hand, the schema needs to be stored as part of the metadata. The proposal consists of linearizing the schema - and adding some valuable information - by using a construct similar to a Data Type Definition (DTD) used in connection with XML. The schema is shown in fig. 3.



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

```
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 (#PCDATA)
        ELEMENT address (#PCDATA)

        ELEMENT year (#PCDATA)
        ELEMENT nbr_lines (#PCDATA)
        ELEMENT dots_per_line (#PCDATA)

        ELEMENT nbr  (#PCDATA)
        ELEMENT gray_value (#PCDATA)
]
```

Figure 3 : DTD for a Collection

In plain English, it would read as:

> A collection is a list of buildings; a building is associated an address, a name, and a list of pictures; a picture is associated 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. At this point, let us assume that the client knows the DTD.

The method interface

The interface supports accessing all values in the tree according to a depth-first traversal. The client simply executes the following (pseudo-code) statements:

```
open object
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 the XML conventions, unambiguously indicate the type of each element. In the example, the 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>
                                <nbr>           1
                                <gray_value>    102
                                <gray_value>    104
                                <gray_value>    116
                                ...
                        </line>
                        <line>
                                <nbr>           2
                                <gray value>    211
                                <gray value>    234
                                ...
                        </line>
                        ...
                </picture>
                ...
        </building>
        ...
</Collection>
```

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 from the system. A simple solution consists of defining a schema for the metadata. :

```
DOCTYPE  Metadata [
        ELEMENT fields (root_name, comment, field+)
```

```
        ELEMENT field  (level, name, description, type?, attribute?)

        ELEMENT root_name (#PCDATA)
        ELEMENT comment (#PCDATA)
        ELEMENT level (#PCDATA)
        ELEMENT name (#PCDATA)
        ELEMENT description (#PCDATA)
        ELEMENT type (#PCDATA)
        ELEMENT attribute (#PCDATA)
]
```

The *level* specifies the depth of a record in the hierarchy The same code "open... get_field..." can be used to query the metadata, yielding the following results, also in depth-first order (note that the absence of the *attribute* clause means that a single value is required for the field):

```
<fields>
        <name>                  Collection
        <comment>               This is a set of  building pictures
        <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>          #PCDATA
                <attribute>     ?
        </field>
        <field>
                <level>         1
                <name>          address
                <description>   postal address of the building
                <type>          #PCDATA
        </field>
        <field>
                <level>         1
                <name>          picture
                <description>   list of pictures for the same building
                <attribute>     +
        </field>
        <field>
                <level>         2
                <name>          year
                <description>   year the picture was taken
                <type>          #PCDATA
        </field>
        <field>
                <level>         2
```

```
                <name>          nbr_lines
                <description>   number of horizontal dot lines in the picture
                <type>          #PCDATA
        </field>
        <field>
                <level>         2
                <name>          dots_per_line
                <description>   number of dots per horizontal line
                <type>          #PCDATA
        </field>
        <field>
                <level>         2
                <name>          line
                <description>   set of  lines
                <attribute>     +
        </field>
        <field>
                <level>         3
                <name>          nbr
                <description>   sequential line number
                <type>          #PCDATA
        </field>
        <field>
                <level>         3
                <name>          gray_value
                <description>   gray level (white: 0 black: 255)
                <type>          #PCDATA
                <attr>          +
        </field>
</fields>
```

The introduction of a data model like the one presented above accomplishes the following: it defines a universal interface for accessing the archived data. Since it is universal, 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.
It also provides a similar universal interface for retrieving the information about the data (the schema).

5.1.2. Specification of methods

The responsibility of extracting the logical data elements from the data stream lies with the methods. These methods must be expressed in some language. Which one, how? Let's try some possibilities:

1. describe the algorithm in 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. use a high level language; however, high level languages are designed to facilitate the writing of a program. They 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. use 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 the difficulties above.

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 for ever.

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. Actually, there is nothing to be saved in E since the architecture of M2000 becomes irrelevant.

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. It can be checked by running the same UVC program through both the 2000 and 2000+x interpreter. In other words, any UVC interpreter can be checked by comparison with the interpreter of the previous generation.
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.

5.1.3. 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 only 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 is not a problem neither. It is anticipated that the methods will be coded in some high level language anyway, and compiled onto UVC instructions.

5.1.4. The UVC Interface

In 2100, a machine M2100 will come with a restart program that will read the contents of the encapsulated object in a virtual memory and then issue requests to the UVC Interpreter.

A UVC has a set of registers simulated by the interpreter. A request will put some values into some specific registers before giving control to the interpreter which will execute each instruction in the UVC program, sequentially. The registers used at the interface level are:

- 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.

The main functions are:

k = 0: Request for alphabet
The UVC Interpreter reads a displacement at *Reg 2 to branch to the UVC code. It interprets the code that computes the location of the code for function 0, and jumps to that location. Function 0 returns in *reg 3 an ASCII name and/or a description of the alphabet used to encode the characters. There is a bootstrapping problem here, since the alphabet used to describe the alphabet must be known. Depending upon the circumstances, various solutions are possible; they are not critical to the rest of the paper.

k = 1: Open for metadata, k = 2: Get next metadata
Together, these calls return a series of data elements with their related tags. These data elements describe the type of the data elements that will be returned by the next type of call (k=3 or 4).

k = 3: Open for data, k = 4: Get next data
Together, these calls return the actual data elements with their related tags. Suppose the physical representation of the data is the following: 32 bits to indicate the number of buildings, seven bits to indicate the length of the first building name, the name itself, seven bits for the length of the address, the address itself, 15 bits for the year, 12 bits for the number of lines, the number of dots, 12 bits for the line number, etc... The data may start as

(2345) (11) ABC_Building (14) 12_Main_street (1903) (1200) (2100) etc.

The pseudo-code for the UVC program to open the file (k=3) looks like this:

1. Initialize work area (*Reg 5) to point to the beginning of the file
2. Initialize State to 1; store in work area.
2. Return with normal return code

(Note: *R5 means "at address specified in R5".)

The pseudo code for Get next (k=4) starts like this:

1. Get current pointer to the data (from *R5)
2. If (State is not 1) goto 12
3. Get 32 bits to get the number of buildings
4. Store in work area
5. Increase current pointer by 32
4. Get 7 bits for length
5. Copy empty string at *R3
6. If (length is zero) goto 8
7. Get k bytes and store at address *R3
8, Store <name> at *R4
9. Increase current pointer by 7
10. State=2; save current state in work area
11. Return...
12. ...

5.1.5. Highlights of the approach for data archiving

- The existence of UVC suppresses the need for a machine emulator.
- It 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 methods are supplied, able to interpret the format. The only standards needed are now the UVC and the data model, both of which 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.
- The UVC can be made so general, and at the same time so basic, that its definition will remain relevant in the future.

5.1.6. Organization of the bit stream

As a result of the universal interface and the fact that the actual extraction of the data is performed by the - also archived - methods, the bit stream organization becomes very simple to explain. It consists of:

Tag La A Lt T S

where the meanings of the various fields are:

Tag: 8 bits - see below
La: 32 bits that contain (right justified) the binary representation of the length (in bits) of the field A
A: the name of a well known alphabet (such as ASCII, EBCDIC, etc...), expressed as ASCII characters. This is the encoding under which the client will be able to read the field T.
Lt: the length (in bits) of the following field, T.
T: the description of the alphabet encoding for character string data.
Note that Lt can be 0 and T empty if the alphabet is the same as the one specified in A.
S: The data and methods

Clearly, in 2100, the stream can be decomposed easily in its various components. The interpretation program simply skips 8 bits, interprets the following 32 bits as a length La, reads the La following bits and decomposed them into ASCII characters. This yields the name of the alphabet encoding for T. The program then considers the next 32 Lt bits, interpreting them as an integer specifying the length of T, reads the next Lt bits as T and passes the address of S to the executor, which takes control.

Note that, in order to restore the information, we did rely on two assumptions: that the content of A is stored in ASCII, and that the lengths are stored as 32-bit integers. These assumptions can be made today, well publicized, and identified as assumptions 0. The value zero is stored as a binary value in the tag. So, a restore program will always first look at the tag and start the decoding using the appropriate assumptions. In the future, if there is ever a reason for changing these assumptions, a tag 1, then 2, etc. can be used.

5.2. Archiving programs

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 data in D and the UVC method to decode the data in C, the M2000 program will be stored in D and a UVC program that emulates every instruction of M2000 will be stored in C. This time, in 2100, the UVC Interpreter will interpret the M2000 machine instructions. 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 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 conclusion

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 2000,
1) whoever creates a new data format needs to produce a UVC program to decode the data.
2) (for program archiving only,) for each existing or new machine, the manufacturer needs to provide an emulator of the M2000 written as UVC code. Manufacturers of devices in 2000 need to provide UVC code that emulates the device control unit.

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

2) (for program archiving only,) each device manufacturer needs to produce an implementation of the abstract devices on particular 2100 devices.

What the proposed method accomplishes is to avoid 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 unpracticable. Instead, the proposed solution replaces the need for a multitude of standards (for each format) by a single standard on a UVC, which is actually independent of any application, and is so basic that it will remain relevant in the future. The method also tends to do some preparatory work in 2000 - when the information is well known - rather than postponing it to 2100 when the difficulty would be much greater.

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 can not 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. The goal of this paper is to document some initial ideas as a way of generating a genuine interest in the technical aspects of a very crucial problem.

Acknowledgments

The importance of the problem was brought to the attention of the author by Robert Morris, then Manager of Data Systems Technology at the IBM Almaden Research Center. His encouragements and those of Robin Williams are gratefully acknowledged. I wish to thank Henry Gladney for many lively discussions on the subject. Also, Henry, Joerg Meyer and Bruce Baumgart provided insightful comments on the draft.

References

[0] J. Rothenberg: *Ensuring the Longevity of Digital Documents. Scientific American*, 272(1), January 1995.

[1] D. Waters and John Garret: *Preserving Digital Information. Report of the Task Force on Archiving of Digital Information.* Commissioned by the Commission on Preservation and Access and the Research Libraries Group, Inc., May 1996.

[2] IBM *DB2 Digital library*, Version 2.4, IBM Corp. 1999

[3] Management Council of the Consultative Committee for Space Data Systems, NASA: *Open Archival Information Systems (OAIS)*, White Book, Issue 5.0, 1998.

[4] J. Rothenberg: *Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation.* A report to the Council on Library and Information Resources, January 1999.

[5] D. Bearman: *Reality and Chimeras in the Preservation of Electronic Records, D-Lib Magazine,* Vol. 5, No 4, 1999.

[6] P. Dadam & al.: *A DBMS Prototype to support Extended NF2 Relations*, ACM Sigmod, May 1986.

[7] A. Aho et al.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[8] E. R. Harold: XML, *Extensible Markup Language*. IDG Books Worldwide, 1998.