Development of a Universal Virtual Computer (UVC) for long-term preservation of digital objects

J.R. van der Hoeven

Delft University of Technology, Department of Computer Science (EEMCS); IBM Business Consulting Services; and Koninklijke Bibliotheek, National Library of the Netherlands

R.J. van Diessen

IBM Business Consulting Services

K. van der Meer

Delft University of Technology, Department of Computer Science (EEMCS); Antwerp University, Department of Information and Library Science (IBW); and DECIS, Delft, Netherlands

Received 20 December 2004
Revised 6 January 2005

Abstract.

Emulation has been proposed as a preservation strategy for longevity of digital objects. An advantage of emulation is the relative platform independency. Emulation with a high degree of platform independency requires a Universal Virtual Computer (UVC), as described by Lorie. This paper describes the realization of the enhanced UVC developed for the National Library of the Netherlands. The preservation method of the UVC at preservation time and at retrieval time (in the future) is concisely described, the conceptual model of the UVC is given, and the object design and the logical data viewer of the UVC are presented. The UVC has been realized and proven to work for image types using the JPEG and GIF87a formats, although performance of the present UVC can still can be improved. The UVC demonstration tool is freely available at the Alphaworks web site to let the general public experience this approach on long-term access.

Correspondence to: Jeffrey.vanderHoeven@kb.nl

1. Introduction

The advance of digital objects, like digital documents, web pages, databases and program derivates, comes with a challenge. How can digital objects be accessed and rendered in the future, when information processing hardware, software and file formats have changed and today's storage media cannot be read any more? It is a serious challenge: with the current speed of change of ICT, it may be impossible to access and render present information objects within a decade of their creation, and within half a century society could remain without accessible digital memory of our times.

1.1. Projects

Influential projects in this area were Cedars (Curl Exemplars in Digital ARchiveS [1]), its successor CAMiLEON (Creative Archiving at Michigan & Leeds: Emulating the Old on the New [2]) in which special attention was paid to emulation; and NEDLIB (Networked European Deposit Library [3, 4]), the collaborative project of European National Libraries [5], followed by the IBM/KB Long-Term Preservation (LTP) study [6] at the National Library of the Netherlands. The projects led to the insight that durable access to digital objects needs flexible, robust and durable standards and standardization of technical subjects (standards: for example universal formats – PDF [7], SGML [8]/XML [9], and TIFF [10]); and standards and standardization of organizational subjects like procedures for digital archives; and preservation strategies like emulation and (repeated) migration. The projects also led to tangible results: basic infrastructures for networked deposit libraries, including demonstrators of digital archives and operational repositories for digital objects.

At the National Library of the Netherlands the e-Depot (electronic repository) with the DIAS (Digital Information Archiving System [11]) as its core has been built. The DIAS is an ICT structure already containing a repository with 2.6 million digital publications in the autumn of 2004 [12]. This type of result provides a solid test bed for business models and metadata schemes, as well as for the different proposed preservation strategies.

1.2. Emulation

One preservation strategy is emulation [13, 14]. Emulation is strongly advocated by Rothenberg [15]. An emulator imitates a computer platform or application on top of another computer platform or application. Although emulation is a rather complex process, it is not new. It has already been applied for many purposes. Back in 1994 the computer company Apple introduced their new Power Macintosh, based on a PowerPC processor. To continue supporting software programs written for the older Motorola 68000 processor, they emulated this processor on the PowerPC chipset [16]. In a similar way emulators were developed by different manufacturers to run the Macintosh Operating System under Microsoft Windows on Intel-based machines and vice versa.

Gaming is another important field in which emulation plays a prominent role. Emulation forms the key to run computer games written for an obsolete platform on a current machine. Numerous emulators can be found on the Internet [17]. In the open source community several other interesting emulation projects can be found, like SIMH [18], Bochs [19] and QEMU [20], enabling the recreation of historic and current platforms.

In preservation terms, emulation aims at retrieving a digital object in its original form. This is an advantage for emulation over migration, another attractive preservation strategy. Emulation recreates the original environment without changing the authentic digital object, while migration entails making periodic transformations in archived information, the need for new validation and the risk of error propagation [21]. Emulation does not require changes to the original software to interpret the logical form and to view the object. The original program runs in executable form on the emulated environment as it did when running on the original hardware [22].

Ideally, a single universal emulator could serve as a platform to run many original applications, enabling various types of original information objects to be viewed.

This train of thought raises an important question: how can we ensure that an emulator, able to execute preserved digital objects in their (virtual) original environment, will run correctly on future computers, without knowing what a future platform will look like?

1.3. The Universal Virtual Computer (UVC)

An answer to the question how an emulator will run correctly on future computers is found in the concept of the Universal Virtual Computer (UVC) by Raymond A. Lorie (IBM Almaden Research Center) [23, 24]. This concept combines the best aspects of emulation and migration: emulation through the UVC as a platform-independent layer on top of (future) hard- and software, and migration as supported by the conversion of specific formats to universal technology-independent formats based on XML-like specifications. The platform independency promises that programs, developed for the UVC, can run on this platform today as well as in the far future, while the technology-independent formats keep the objects accessible and understandable over time.

Analysis of the initial results of experiments with a prototype conducted at the National Library of the Netherlands (KB) in 2001 looked promising [25, 26]. The KB stated the UVC to be a good candidate to provide long-term access to their static PDF data. As a consequence, IBM and KB worked together to develop an enhanced UVC demonstration tool: the UVC for images project. It had to be able to fit in the DIAS system environment and, essentially, to operate on the DIAS repository.

In this paper, we describe the realization of this UVC, its conceptual model, its object design and its logical data viewer, and we discuss concisely the current state of the UVC, including performance data.

2. The UVC-based preservation method

The UVC-based preservation method, now developed, allows digital objects to be reconstructed in their original appearance any time in the future using a unique combination of emulation and migration.

The central idea of the UVC-based preservation method is based on four different components. These are:

- Universal Virtual Computer;
- UVC program (format decoder);
- Logical Data Schema (LDS) with information type description;
- Logical Data Viewer.

A UVC program decodes the file format of a digital object. This format decoder program runs on the UVC, which is the platform-independent layer, independent of future hard- and software changes. Executing the format decoder delivers element tags, which hold specific information about the content of the data in a technology-independent manner. These elements build the Logical Data View (LDV) of the data, which is quite similar to XML. The LDV is an instantiation of the LDS, describing the structure and meaning of the tags as parts of a specific information type.

All these components are controlled by a Logical Data Viewer simply called viewer (Figure 1). For reconstruction, the viewer starts the UVC and feeds it with the data of the digital object to a format decoder running on top of the UVC. In return it retrieves an LDV and reconstructs a specific representation of the original object's meaning.

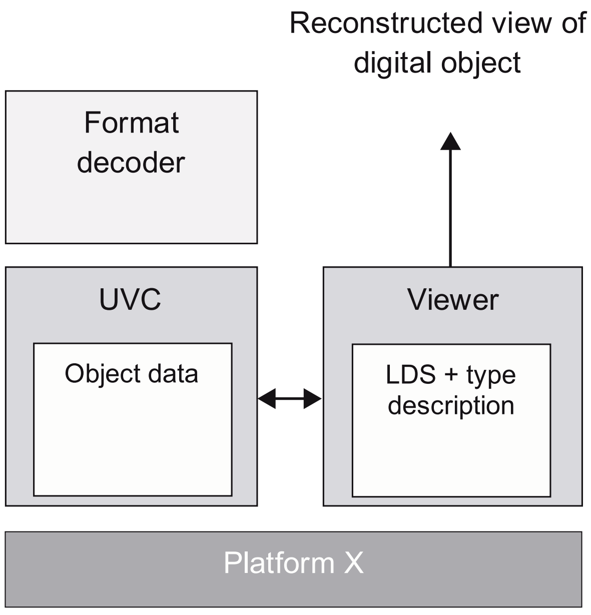


Fig. 1. UVC-based preservation method.

To apply the UVC-based preservation method on preserved digital objects, different steps must be taken in the present as well as in the future. These steps are described in more detail below.

2.1. *At preservation time (the present)*

Step 1. To view a digital object in the future, a detailed description needs to be developed of its structure (the logical form) and meaning independent of any technology. This logical view is returned by the UVC in the future and needs to be interpreted by the viewer; therefore future developers must be able to understand it.

As an example, Figure 2 shows a simplified schematic view of a raster-based image, using the red–green–blue (RGB) colour model as is used in most common image formats. The view consists of different elements, starting with the number of scan lines, representing the horizontal lines of an image. It can contain one or more of these scan lines, which is depicted by a '+' sign. Each scan line contains one or more pixels, while each pixel has a position and a colour.

As mentioned above, executing the UVC will return this information in such elementary tags. All tags and their relationships are defined in a blueprint called the LDS, which explains the tags that are retrieved from the UVC for a particular type of digital object. Here a type is defined as a particular group of files, like an image, sound wave or spreadsheet. The advantage of the LDS is that the same LDS can be used for all formats of a type. A simplified LDS for the image type is schematically depicted in Figure 3.

But knowing the structure of the LDV is not enough. To understand the meaning of the elements in the LDV, the LDS provides a description of the information type, e.g. that the RGB colour scheme is used and how it is related to the natural colour spectrum.

Step 2. The digital object has to be translated (migrated) into the description of its logical view. Therefore, a format decoder running on the UVC is required that can decode the object's format, using the elements defined by the LDS. This format decoder must be written before the format has become obsolete. For each format a decoder has to be written, which requires a lot of effort. But once a decoder is available it can be applied to every digital object of that format.

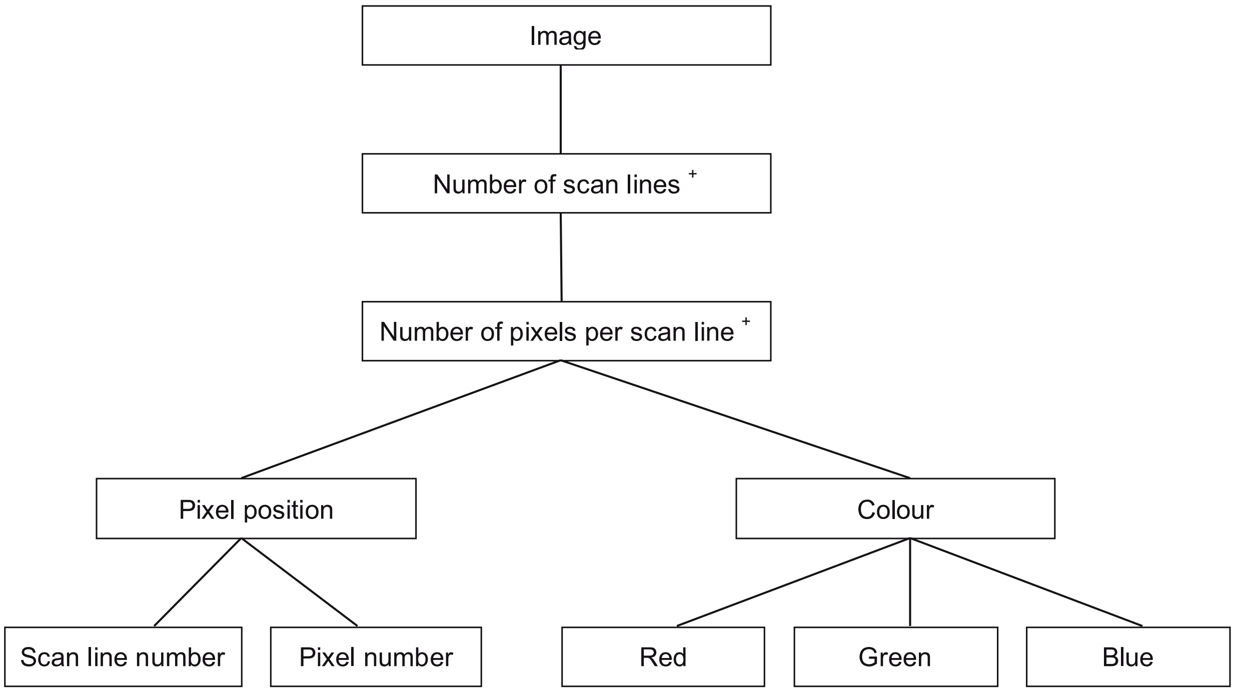


Fig. 2. Simplified schematic view of a raster-based image.

```
ELEMENT 1 [Image] (10, 24+)

ELEMENT 10 [Image Size] (11, 12)
ELEMENT 11 [Number Scan Lines]
ELEMENT 12 [Number Pixels Per Scan Line]

ELEMENT 24 [Pixel] (25, 29)

ELEMENT 25 [Colour] (26, 27, 28)
ELEMENT 26 [Red]
ELEMENT 27 [Green]
ELEMENT 28 [Blue]

ELEMENT 29 [Pixel Position] (30, 31)
ELEMENT 30 [Scan Line]
ELEMENT 31 [Pixel Number]
```

Fig. 3. Simplified Logical Data Schema (LDS) for a raster-based image.

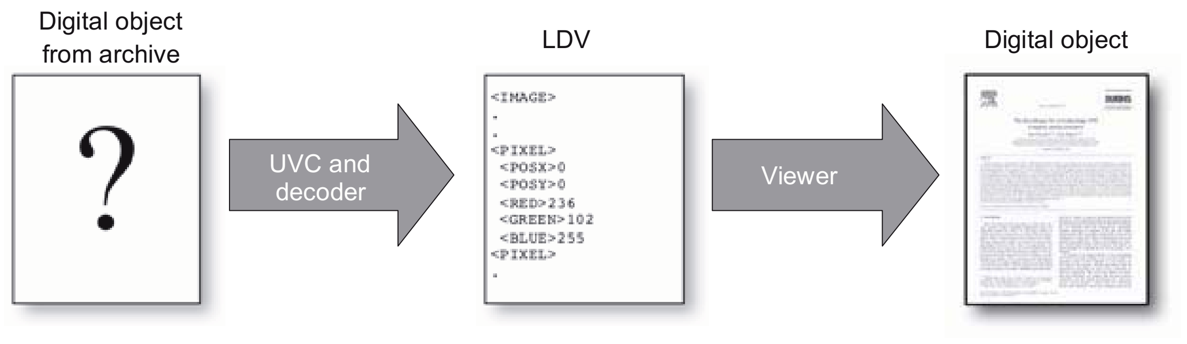
Step 3. Future developers have to know how to construct a UVC, in order to execute the format decoder program for a particular object's format. Software developers in the distant future will need a well-defined specification of the UVC in order to create a new UVC. The UVC is designed to be a general-purpose computer, running on any (future) hardware.

To reproduce a UVC in the future, a description of the present concept must be preserved. This can be done as a document in a digital repository, as a hardcopy on paper and/or on micrographic media.

2.2. *At retrieval time (the future)*

Step 1. If the digital objects, LDS descriptions, format decoders and UVC specification have all been preserved successfully, any object for which the UVC fulfils a decoding process can be reconstructed. First, a UVC has to be created on a current platform. Because of the simplicity of the UVC concept, it is fairly easy for skilled software developers to construct a UVC for a particular platform of the time.

Step 2. Assuming that a proper working UVC exists, a simple application program must be developed. This application program, called a viewer, has to control the UVC and all input/output interaction between it. It needs to run on future hardware and therefore cannot be specified at preservation time. The viewer has to start the UVC, send the encoded data and format



```
<IMAGE>
.
.
<PIXEL>
  <POSX>0
  <POSY>0
  <RED>236
  <GREEN>102
  <BLUE>255
<PIXEL>
.
```

Fig. 4. Reconstruction of a digital object.

decoder to it and receive Logical Data View tags back from the UVC. With these tags and the explanation defined in the LDS, a representation of the original object can be created (Figure 4).

3. The UVC conceptual model

Developing a UVC in the future requires a well-defined specification. This specification will become final and publicly available soon. It defines four primary aspects of the emulator:

- The architecture
- The loading/state specification
- The communication behaviour of the abstract channel
- The supported instruction set

3.1. Architecture

The core element of the UVC is its segment-based memory. Normally, a certain amount of memory is available to all software running on a system. This introduces the problem that applications using parts of memory can overwrite other critical parts in the same memory space. Another restriction of a conventional computer system is the limited available memory space. Instead, the UVC uses segments of memory to store distinct parts of data, which takes away the first problem. Each segment has the ability to use more than 4 Gigabits of memory and the same number of distinct registers. Each register and available memory can grow without limitation (physically register length is bounded by its length specification defined as a 32 bit number equivalent to 4 Giga), essentially taking away the second problem stated.

Segments are logically identified by 32 bit numbers. The architecture makes a distinction between logical and physical segments. Logical segments are the segments referenced by a number within a format decoder. Physical segments are the actual segments as they are internally allocated by the UVC and they are identified with the prefix 'Ph' to distinguish them from logical segments. For example, within a format decoder a logical segment 4 could be internally represented by physical segment Ph32.

Programs written for the UVC can use this segment-based memory. These UVC programs consist of different pieces of code, called sections, which are stored in separate segments. A section consists of a sequence of UVC instructions that form a routine, performing a set of operations. Each section can call on one or more other sections, linking the whole program together. A logical segment used in one section does not have to be unique from other sections, because the UVC internally translates these logical addresses into unique physical ones. Sections have to respect four reserved logical segment numbers, which are:

- *Segment 0* is a shared segment that can be accessed from any other segment in memory by referencing to the logical segment 0. The memory of segment 0 is used to store the data from the file to be decoded. The registers can be used to store global constants used by the decoder program.
- *Segment 1* is the logical reference to a segment associated with every individual section. Each section is only stored once in memory, but can be instantiated multiple times. Each instantiation can use the logical segment 1 as a local segment, which is accessible by every other instantiation.
- *Segment 2* is the logical segment that serves as parameter segment. When a section of the program calls another section, it can send a parameter segment with it. This segment is made accessible in the called section by using segment 2.

- *Segment 3* is the logical segment that should contain the start-up section of the program, i.e. the main routine.

The administration mechanism is realized by three components: *dispatch table*, *activation stack*, and *instruction pointer*. Figure 5 shows an overview of these components, together with the segment-based memory structure.

3.1.1. Dispatch table. To keep track of all sections of program code loaded into the UVC, a dispatch table is introduced. Each section should have a unique ID serving as a key to identify a particular section within the UVC. When the UVC receives a section it reads its section ID and creates two new segments: one to store the code of the section in and the other for defining the local segment 1 associated with this section. After that, it has to record all logical segments used by the section. During this loading process, all the information of the different sections will be stored in the entries of the dispatch table, creating a map that can be used during the execution of the program.

In Figure 6 a dispatch table is depicted after loading two sections with section ID 3 (start-up) and ID 100. The first section (ID = 3) is stored in physical segment Ph1 and has physical segment Ph2 assigned as logical segment 1 (local segment). The second section (ID = 100) is stored in physical segment Ph3 with physical segment Ph4 as associated logical segment 1. The information about the logical segments of each section will be used later on by the activation stack as described next.

3.1.2. Stack architecture. The UVC needs a stack mechanism for the execution of a program. A stack is like a pile of plates, where the one added on top is the first one taken out again according to the Last-In-First-Out (LIFO) principle. Items can be added to the stack via a *push* operation and removed from the stack using a *pop* operation. In UVC terms, the items are formed by creating an instance of a section, called an activation record, and the stack is consequently called an activation stack. Each call to a section will create a new activation record of the called section. As a result, the activation record will be pushed onto the activation stack. Moreover, it is possible that a section calls itself, creating a recursive function. This is a powerful aspect of stack architecture, making programs very flexible. When the code of the current activation record is at the end, it will be popped from the activation stack and the former last activation record on the stack will become active again. The process of pushing and popping activations continues until all activation records are popped from the stack and there is nothing left to do.

An important aspect of the UVC's stack architecture is the way information is passed between two sections on the stack. This is done by defining a parameter segment at the calling section when the call is made. This segment is then linked to logical segment 2 of the called section. In this way, data stored in a particular segment of the caller can be accessed by using segment 2 in the called section.

3.2. State specification and communication behaviour

Within the UVC different states have to be distinguished to mark what the UVC has to do. Each state performs a certain task within the UVC. Four states are defined and will be handled in the sequential order that is presented in Table 1.

If the UVC is in the 'execute program' state it can return from this state if:

- the UVC encounters the STOP instruction in the executing code;
- some fatal error has occurred, like an instruction mismatch or memory problems.

The interaction between the viewer and the UVC runs through a half-duplex, synchronous communication channel, whereas the data packages are formatted in a message-based manner.

3.3. Instruction set

UVC programs should be written using a predefined UVC instruction set. This set consists of 25 instructions, categorized as follows:

- logical bit manipulation – supports *and*, *or* and *not* operations on registers;
- arithmetic operations – add, subtract, multiply or divide register values;
- comparison – compares values on equal and greater than;
- memory operations – load and store data in and from register to memory;
- register sign handling – reads or writes the sign of a register;
- jump and context switching – jump within a section or call another section;
- communication – communicates via *in* and *out* with the viewer;
- control operation: stops the UVC from execution.

With this set of instructions it is possible to write any sort of program.

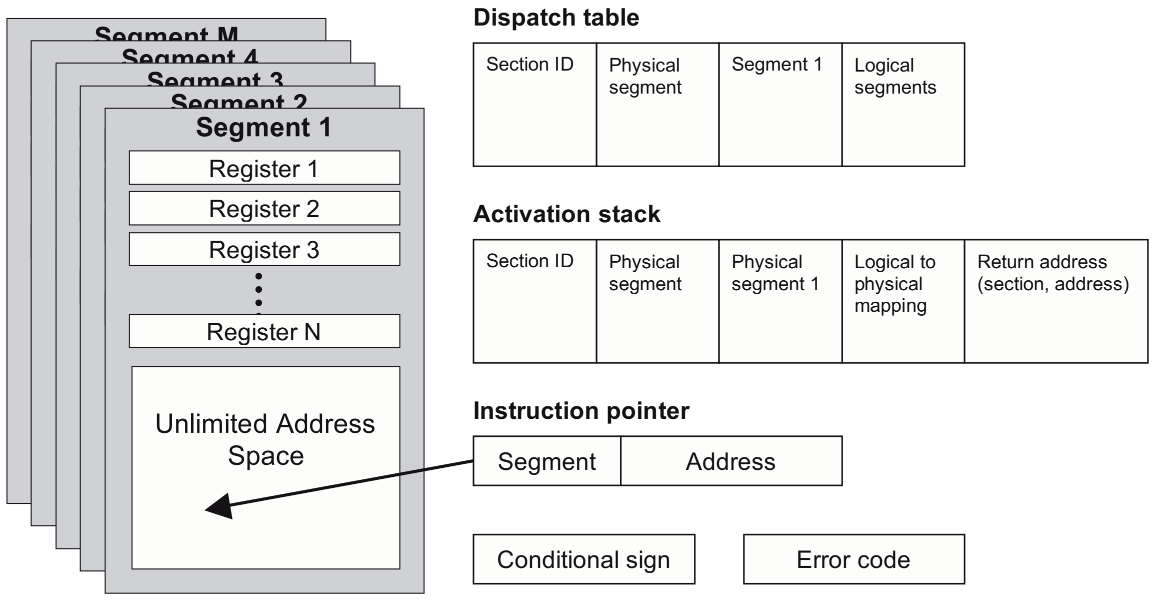


Fig. 5. Segment-based memory and administration components.

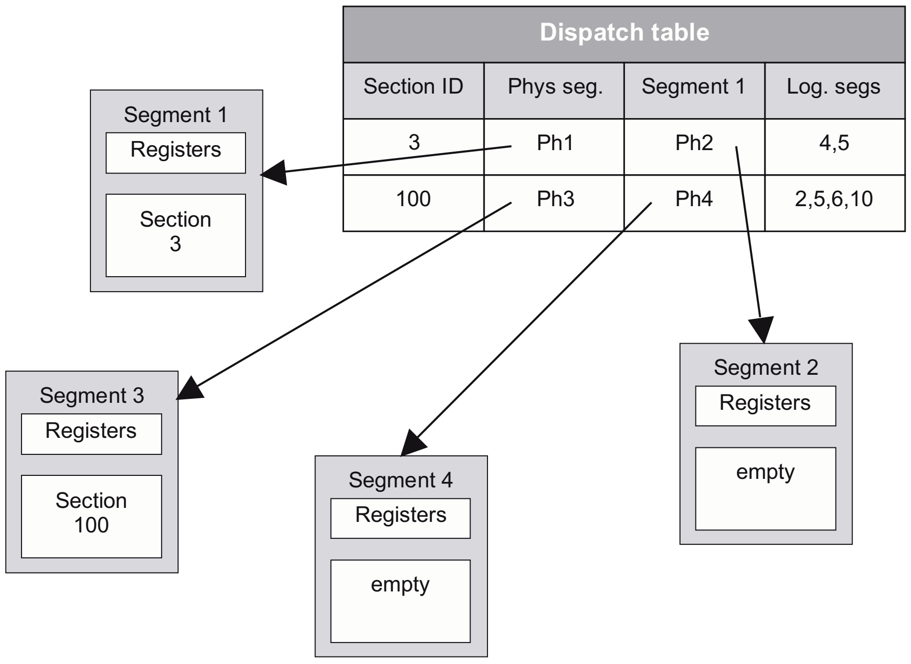


Fig. 6. Dispatch table after loading two sections.

4. Realization

Parallel to the definition of the UVC specification, a UVC demonstration tool has been developed. The purpose of this demonstration tool is to experience the capabilities and limitations of the UVC approach. To prove its usefulness, the KB/IBM project chose to create a secure path for restoring digital still images in

Table 1
States of the UVC

State	Action to go back to initial state
Load constants	*Load constants in global segment 0* Number of registers to be loaded in segment 0 (32 bits) List of registers to be loaded: ○ register number to be loaded (32 bits) ○ sign of the constant, 0 = positive, 1 = negative (1 bit) ○ length of the constant to be loaded (32 bits) ○ the bit string of the constant
Load section	*Load a section of object code with format* Logical section number (32 bits) Number of logical segments used (32 bits) List of local defined segments: ○ logical segment number (32 bits) Length of the section code (32 bits) Section code
Initial data load	*Load the data file in segment 0* Length of the data file to be loaded (32 bits) Data file
Execute program	*Start program* This one has no arguments. The UVC will start at address 0 of the logical section 3, which is presumed to contain the main program.

JPEG format using the UVC-based preservation method. The choice for JPEG as file format is based on its widespread use, its versatile capabilities and the ability to create JPEG images out of PDF files. Reconstructing a PDF using the UVC would take more effort than was planned for the development of a demonstration tool and could be a next step in the project.

The realization of the UVC concept entailed the development of a UVC, an Image LDV Viewer, a JPEG format decoder and an Image LDS. We chose to implement the UVC and viewer in Java, because of its object-oriented structure and portability to different platforms. By means of incremental design, several iteration cycles in the development process have been run, giving feedback concerning the UVC in practice. These results have been used to further refine the specification, design and implementation of the UVC.

4.1. UVC

The UVC is described in an object-oriented structure, which was the design for a Java-based implementation. It consists of different objects, each with its own specific task. Three primary tasks were defined:

- memory management – internal segment-based memory management;
- execution management – execution control of the program running on UVC;
- I/O management – input/output operations between UVC and viewer.

In Figure 7, the system is schematically decomposed into these and other objects together with their multiplicity. The objects in this schema have a one-to-one relation with the classes in the Java implementation.

4.1.1. Memory manager. The memory manager organizes storage and retrieval of data inside the UVC. Within the UVC, memory is organized in different segments. Each section of code is stored in its own segment. A segment on its own can contain multiple memory blocks and registers, represented as different objects in the structure. The memory manager controls the allocation and usage of segments and is bit addressable. Internally these bit addresses are converted into byte addresses, because Java uses bytes as its smallest variable types.

4.1.2. Execution manager. The heart of the UVC is formed by the execution manager. The execution

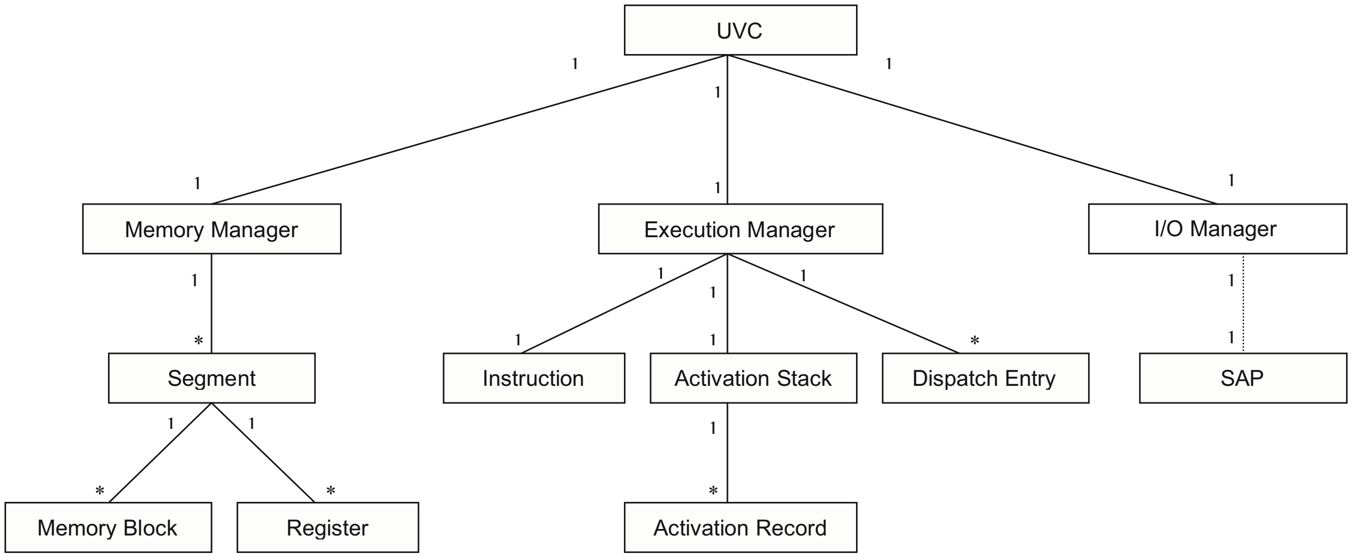


Fig. 7. Class diagram of the UVC.

manager controls the actual flow of the execution, carrying out each program instruction in sequential order and one at a time. In this implementation, pipelining is not considered because the UVC should be as simple as possible.

The execution manager controls three objects: instruction, activation stack and dispatch entry. The instruction object is used to keep track of the next instruction to execute; it works like a pointer to one particular segment and address in memory. The dispatch entry object is created for each loaded section in the UVC, forming the entries of the dispatch table. Finally, the activation stack object implements the stack-based architecture as described by the UVC concept. Each time a section on the stack is activated, the object activation record will be instantiated.

4.1.3. I/O manager. The I/O Manager implements an abstract communication channel between the viewer program and the UVC. The communication channel loads constants, sections and data into the UVC and in turn outputs the processed data as elements of the LDV. The interface between the UVC and viewer is handled by a Service Access Point (SAP). This SAP functions as the communication channel for sending and receiving data to and from both the UVC and the viewer, as depicted in Figure 8.

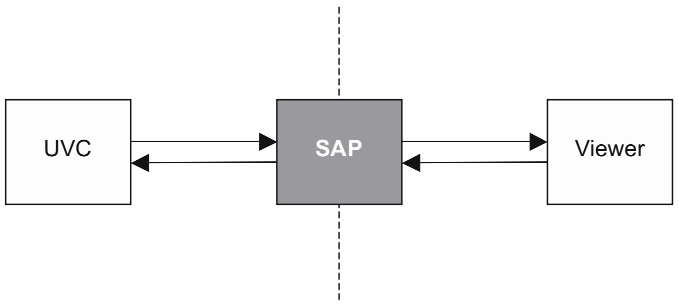


Fig. 8. I/O communication between UVC and viewer via a Service Access Point (SAP).

4.2. Image LDV Viewer

Besides the UVC, a viewer is needed to control the whole process. For this demonstrator an Image LDV Viewer has been created. Because various LDV viewers will be needed, the Image LDV Viewer is designed for reusability. Handling the image information of the LDV is viewer specific, but communication via SAP is identical for all viewers. The implication of the distinction between general and specific viewer components is that new viewers can inherit these general components, making it easier and more efficient to construct new viewers.

In addition to process control, the viewer supports a graphical user interface (GUI). It contains a menu bar, control bar, image field and status bar. The menu bar offers a *print* function and *about* information for this program. The control bar allows the user to select an image and image decoder, as depicted in Figure 9. The reconstruction process can be started and stopped via the appropriate buttons.

The image field shows the image, with scrollbars if necessary. During reconstruction the image is refreshed after every 1000 pixels, showing the progression on screen. Figure 10 demonstrates the reconstruction of a JPEG image with dimensions 500 × 375 pixels. The

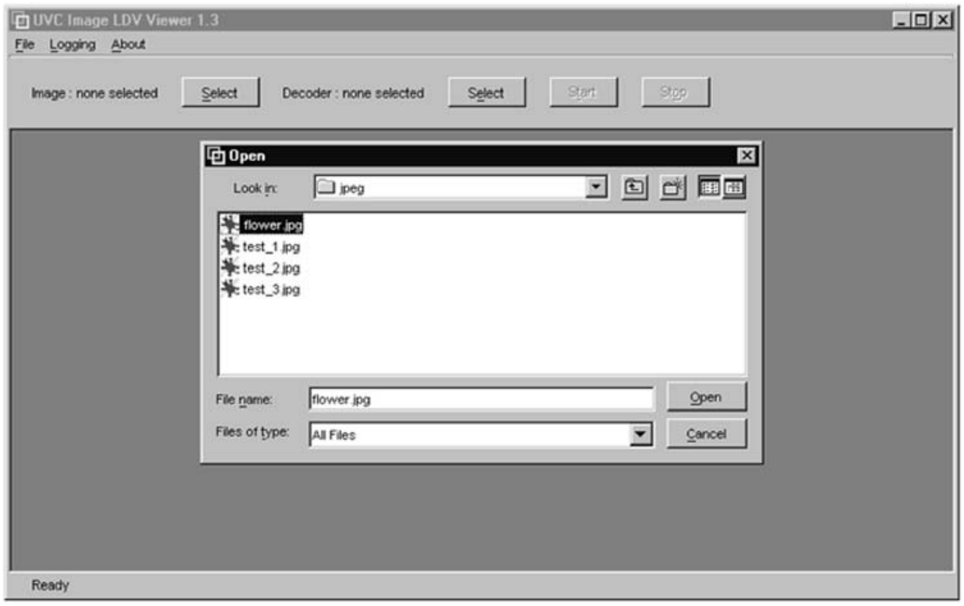


Fig. 9. Screenshot of the Image LDV Viewer – select an image.

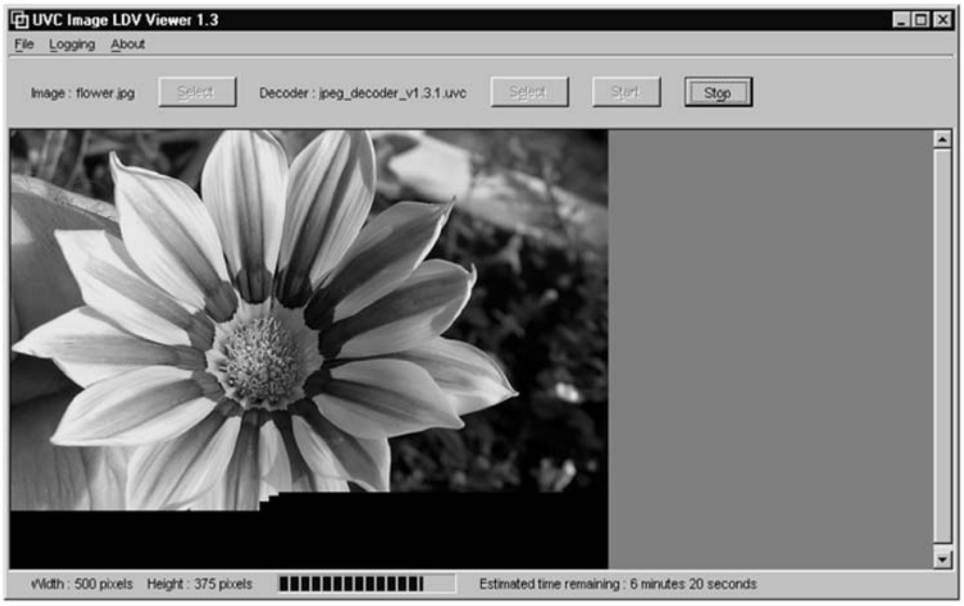


Fig. 10. Screenshot of the Image LDV Viewer – image during reconstruction.

status bar at the bottom shows the image dimensions, progress of the image and estimated time until completion.

Apart from the UVC and Image LDV Viewer, a JPEG format decoder has been developed, together with an LDS description and image test set. The JPEG format decoder is based on the JFIF 1.02 specification by the Independent JPEG Group [27]. At the end of the development of the UVC a second image format decoder has been implemented for GIF87a images, following CompuServe's Graphics Interchange Format specification from 1987 [28]. The ability to use the same UVC and image viewer for different image formats shows the strength of this approach.

5. Experiences

The primary goal of the UVC for images project was to develop a fully functional UVC and viewer to prove that the concept works. In this, the flexibility of the UVC is more important than the speed of execution. The UVC was not designed to be a race monster, due to the emphasis on flexibility and Java's reputation on performance [29].

The first tests showed this prediction was right. Figure 11 shows one of the execution results, obtained during execution of an 8-by-8 pixels JPEG image on a test machine running Windows 2000 on a 500 MHz Intel Pentium III processor and 320 Megabytes of RAM.

The original execution took about 21 seconds! Because the execution time is almost linearly correlated with the image area, extrapolating these results to an A4 paper scanned on 300 dpi (dimensions 3300 × 2500 pixels) would result in a reconstruction time of several days. The execution speed of the UVC is at least a factor 500 slower than an average processor. As a rule of thumb a CPU twice as fast makes the UVC also run two times faster. However, even taking the fastest processor available will not give us the speed we would like.

One of the major speed factors is the bit-oriented nature of the UVC, which had to be mapped and modelled on the byte-oriented architecture of the Java Runtime Environment (JRE). It takes a lot of conversion and shifting operations when manipulating bit strings across the JRE byte boundaries. Another major factor is the increase in data to be transferred over the communication channel by the LDV specification of the digital object. Optimizing the current implementation cannot address both these factors directly.

Despite these factors, other issues could be addressed to gain higher performance rates. To do so, all steps between the native platform and the reconstructed image have been analysed and optimized if possible. In this process, three main issues were distinguished: the JRE, the format decoder and the UVC implementation.

The JRE is based on a Java Virtual Machine (JVM). Although this offers portability between different platforms, it also introduces an extra layer between operating system and Java application causing overhead. To limit the loss of performance by the JRE, tests have been done with different versions of JREs from different manufacturers. The outcomes, as shown in Table 2, indicated that the IBM JRE version 1.3 offers the best performance for the UVC. However, this was the JRE we had already used. Converting the UVC to native code, which converts Java to a platform dependent executable, has also been considered and tested but it takes away the portability advantage of Java and it did not offer the desired performance improvements.

Optimization of the format decoder and UVC is more successful. Analysis of the execution flow of the UVC and its decoder pointed out various delays in function calls. Improving these aspects resulted in an overall performance boost of more than 600% on the same test machine compared to the first tests. At the end the peak performance on the test machine was measured at 109,000 instructions per second.

Other refinements during development were based on improvements of the UVC specification. These include slight changes in the instruction set, exclusion

```
Execution statistics:
---------------------
Total number instructions : 361529 instr.
Total execution time      : 21 sec.
Performance               : 17000 instr./sec.
```

Fig. 11. UVC execution statistics reconstructing an 8 × 8 JPEG image.

Table 2
JRE performance test

JRE performance test	Instructions/s	Index
IBM JRE v1.3	17,000	1.00
Sun JRE v1.3.1	12,240	0.72
Sun JRE v1.4.2	14,280	0.84
Sun JRE v1.5.0	12,580	0.74

of dynamic linking within the UVC to stay close to the principle of keeping the UVC simple and clear, and some extra reserved segments in memory offering more programming freedom for UVC program developers. These changes resulted in the current UVC specification as described above.

6. Conclusions

The UVC demonstration tool has successfully been developed and is operational. It was delivered at the National Library of the Netherlands and it has been brought to the attention of various other institutions. To create a broader interest, the latest version of the UVC demonstration tool is now freely available to the community at large at the IBM Alphaworks web site [30]. This enables the community to get acquainted with the UVC solution and provide valuable feedback for further development.

As a result, the chances are increased that the UVC will be one more option in the variety of preservation strategies that must serve to maintain access to digital objects. A valuable option due to its potential for usability in the very long term, though its performance still must be improved.

At the same time results are under way on the implementation of a Java UVC compiler to enable other organizations to develop their own format decoders. By sharing these format decoders among the different institutions, application of the UVC solution will be extended.

The current UVC solution focuses on a combination of emulation through the UVC and migration through the universal technology-independent formats based on XML-like specifications. This approach works well for static digital object types, like text, image, sound and animation. These objects do not contain dynamic behaviour such as user interaction or executable models used to produce dynamic content during program execution. The next research step we will take is to apply the UVC concept to 'full' emulation where the applications are emulated on the UVC to enable access to digital objects in their original format.

Acknowledgement

We would like to thank Hilde van Wijngaarden (National Library of the Netherlands) for her valuable comments.

References

[1] Cedars, *Curl Exemplars in Digital Archives* (1998). Available at: www.leeds.ac.uk/cedars/ (accessed 15 December 2004).

[2] CAMiLEON, *Creative Archiving at Michigan & Leeds: Emulating the Old on the New* (1999). Available at: www.si.umich.edu/CAMILEON/ (accessed 15 December 2004).

[3] NEDLIB, *Networked European Deposit Library* (1998). Available at: www.kb.nl/coop/nedlib/ (accessed 15 December 2004).

[4] T. van der Werf-Davelaar, Long-term preservation of electronic publications: the NEDLIB project, *D-Lib Magazine* 5(9) (1999). Available at: www.dlib.org/dlib/september99/vanderwerf/09vanderwerf.html (accessed 15 December 2004).

[5] Koninklijke Bibliotheek, *National Library of the Netherlands* (2004). Available at: www.kb.nl (accessed 15 December 2004).

[6] LTP study, *KB/IBM Long-Term Preservation Study* (2002). Available at: www.kb.nl/hrd/dd/dd_onderzoek/dnep_ltp_study.html (accessed 15 December 2004).

[7] *Adobe™ Portable Document Format (PDF)* (2004). Available at: www.adobe.com/products/acrobat/readermain.html (accessed 15 December 2004).

[8] World Wide Web Consortium, *SGML is ISO standard 8879:1991* (1995). Available at: www.w3.org/MarkUp/SGML/ (accessed 15 December 2004).

[9] World Wide Web Consortium, *XML Tutorial* (1999). Available at: www.w3schools.com/xml/default.asp (accessed 15 December 2004).

[10] N. Ritter, *The unofficial TIFF homepage* (1997). Available at: http://home.earthlink.net/~ritter/tiff/ (accessed 15 December 2004).

[11] DIAS, *Digital Information Archiving System* (2003). Available at: www-5.ibm.com/nl/dias/ (accessed 15 December 2004).

[12] J.F. Steenbakkers, Treasuring the digital records of science: archiving e-journals at the Koninklijke Bibliotheek, *RLG Diginews* 8(2) (2004). Available at: www.rlg.org/en/page.php?Page_ID=17068 (accessed 15 December 2004).

[13] C. Bellekom: Building preservation functionality in a digital archive: the National Library of the Netherlands, *Learned Publishing* 17(4) (2004), 275–80.

[14] W. Roberts: Long-term preservation of electronic information, *Bulletin of the Records Management Society* 123 (December 2004) 3–7.

[15] J. Rothenberg, *Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation* (1998). Available at: www.clir.org/pubs/reports/rothenberg/contents.html (accessed 15 December 2004).

[16] *Emulation: context and current status, Digital Preservation Testbed, The Hague, The Netherlands* (2003).

Available at: www.digitaleduurzaamheid.nl/bibliotheek/docs/white_paper_emulatie_EN.pdf (accessed 15 December 2004).

[17] Wikipedia: the Free Encyclopedia, *List of Emulators* (2004). Available at: http://en.wikipedia.org/wiki/List_of_emulators (accessed 15 December 2004).

[18] B. Supnik, Simulators: virtual machines of the past (and future), *Queue* 2(5) (2004), 52–8.

[19] Bochs, *Think inside the Bochs* (2004). Available at: http://bochs.sourceforge.net (accessed 15 December 2004).

[20] *QEMU CPU Emulator* (2004). Available at: http://fabrice.bellard.free.fr/qemu/ (accessed 15 December 2004).

[21] P. Mellor, P. Wheatley and D. Sergeant, *Migration on Request: a Practical Technique for Digital Preservation, ECDL 2002, Rome, Italy* (2003). Available at: www.si.umich.edu/CAMILEON/reports/migreq.pdf (accessed 15 December 2004).

[22] J. Rothenberg, *Using Emulation to Preserve Digital Documents, Koninklijke Bibliotheek, The Hague, The Netherlands* (2000). Available at: www.kb.nl/pr/publ/usingemulation.pdf (accessed 15 December 2004).

[23] R.A. Lorie, A methodology and system for preserving digital data. In: G. Marchionini and W. Hersh (eds), *Proceedings of the 2nd ACM/IEEE Joint Conference on Digital Libraries (JCDL 2002) Portland, Oregon* (ACM, New York, 2002) 312–19.

[24] R.A. Lorie, *The UVC: A Method for Preserving Digital Documents: Proof of Concept. The Hague, IBM and Koninklijke Bibliotheek* (2002). Available at: www.kb.nl/hrd/dd/dd_onderzoek/reports/4-uvc.pdf (accessed 15 December 2004).

[25] J.F. Steenbakkers, Preserving electronic publications, *Information Services and Use* 22(2–3) (2002) 89–96.

[26] E. Oltmans and H. van Wijngaarden, Digital preservation in practice: the e-Depot at the Koninklijke Bibliotheek, *VINE* 34(1) (2004) 21–6.

[27] *JPEG JFIF image format specification* (2003). Available at: www.w3.org/Graphics/JPEG/ (accessed 15 December 2004).

[28] Compuserve, *Graphics Interchange Format (GIF) specification* (1987). Available at: www.w3.org/Graphics/GIF/spec.gif89a.txt (accessed 21 January 2005).

[29] *The Java Performance Report* (1999?). Available at: www.geocities.com/ResearchTriangle/Node/2005/jpr/ (accessed 15 December 2004).

[30] IBM, *Alphaworks Emerging Technologies, Digital Asset Preservation Tool* (2004). Available at: www.alphaworks.ibm.com/tech/uvc (accessed 15 December 2004).