Science & Technology Facilities Council
e-Science

JISC

The Significant Properties of Software: A Study

Brian Matthews, Brian McIlwrath, David Giaretta, Esther Conway
STFC
Rutherford Appleton Laboratory
Chilton OX11 0QX
UK

December 2008

Revision History:

Version	Date	Authors	Sections Affected / Comments
0.1	18/02/2008	BMM	Outline
0.5	05/03/2008	BMM	First Draft
0.6	09/03/2008	DG	DG added section on OAIS/CASPAR
0.7	11/03/2008	BMM	Added section on StarLink + revisions.
0.8	23/03/2008	BMcI, EC, BMM	Expanded use cases
1.0	28/03/2008	BMM, BMcI	First Complete release version
1.1	23/12/2008	BMM	Final Revision

Executive Summary

Digital preservation has become pressing concern as more and more of the records of human activity are generated and processed electronically. Software is a class of electronic object which is not only the result of research, but is frequently a vital pre-requisite to the preservation of other electronic objects. However, consideration to the preservation of software as a digital object in its own right has to date has been very limited. Software is seen as complex – forbiddingly so for people who want to maintain access to software but were not involved in its development – and its preservation is often seen as a secondary activity and one with limited ultimate purpose.

Software preservation is thus a relatively new topic of research and there is little practical experience in the field of software preservation per se. This study has become an exploration of the area of software preservation as much as defining the significant properties for preservation. Consequently, we discuss some of the motivations and approaches taken to preserve software.

Software is a very large topic with great range and diversity. In this study we restricted ourselves to mathematical and scientific software used in the academic research community in parts of the UK. We considered the literature and discussed case studies with either practitioners in providing repositories of software, or developers of software packages which have had a very long lifetime.

Although there are many groups who are holding software to support archives, or to support a community, and many others who are maintaining a usable software package for a long time, these groups do not consider themselves to be doing software preservation and have other priorities. Those that do carry out software preservation are often amateur or specialised in science museums or special interest groups; these are ad hoc and small scale and do not tackle the systematic problems of keeping software replayable in a broad context for the long term. Other projects are looking more systematically at digital preservation and tend to rely on the persistence of software, or at least access to software with similar functionality to the original. However, they tend not to concentrate on the problem of how to preserve the software itself.

However, there are good reasons of preserving research effort in software, often as a vital adjunct for preserving other digital objects. Preserving software essentially means that software can be reconstructed and replayed to behave sufficiently closely to the original.

Software is inherently complex with a large number of components related in a dependency graph, and with specification, source and binary components, and a highly sensitive dependency on the operating environment. Handling this complexity is a major barrier to the preservation of software. Different preservation approaches can be adopted which can execute binaries directly, can emulate the software, or carry out software migration by recompiling source code, or even recoding. All can in different circumstances support good preservation.

Adopting the notion of performance from the NAA and InSPECT, we developed a notion of performance of software which is closely related to the adequacy of the performance on the target data. Establishment and preservation of test cases for expected behaviour of end

software on test data is a key feature for assessing the adequacy of performance of software preservation on specific chosen significant properties.

Good software engineering practice to support software version control, software maintenance, migration and especially software testing can also support software preservation. Groups which have successfully maintained software over a long period have developed rigorous software engineering practice and developed techniques to support software migration in particular.

To capture and control the inherent complexity of software, we have developed a conceptual model for software which is more complex than that of InSPECT, Many of the structuring significant properties of software are thus captured in this model. Significant properties of software are then categorised according to this model and also according to their role. As a consequence, it is observed that the InSPECT categorisation of significant properties does not match comfortably with the significant properties of software. This is probably because of the indirect performance model of software, which is tested by the performance of the end data. Contextual significant properties play a key role and software is dependent upon them being satisfied for satisfactory reconstruction and replay, whilst behavioural significant properties determine the performance of the software on end data.

Given the relatively immature state of the art in software preservation, we consider our definition of a conceptual model of software and the associated identification and classification of significant properties to be a proposal, which needs to be evaluated further in practice to judge its value and effectiveness in practice.

The significant properties identified in this study are still relatively general and do not go into the detail of other significant studies. For example, we decided that we would stop at the level of granularity of code represented by the common coding concept represented by a public class or module or subroutine (terminology varies between programming language) and it would not be worthwhile detailing any further. Other significant properties also stop at a high level, and do not for example enumerate the possible values which they could take[1]. Further testing and evaluation is required to see if this is sufficient and whether the significant properties are always appropriate and whether they can be extracted and used in practice.

Tools support should be eventually forthcoming to support the significant properties of software; however, we feel that the above development of the methodology needs to be investigated further before investing comprehensive tool support.

Recommendations

We conclude with a set of recommendations for JISC.

- Raising awareness of the software preservation within the JISC community.
- Further consideration should be given to the justification and role of software preservation within a broader digital preservation strategy.

[1] As an example of how this may be undertaken, see the example of licensing in Appendix C.

- Specific consideration should be given to the role of software preservation in preservation processes which are conformant to OAIS.

- Further studies should be undertaken to test and extend the notion the conceptual model of software and its significant properties. Studies and test cases should be undertaken specifically in areas which were seen as outside the scope of this study, in particular:
    - Database software
    - Commercial software
    - Business and office software
    - Software which supports the performance of other key digital objects (e.g. documents, vector images, science data).
    - Systems and networking software.

- Studies and practice of software preservation should involve experience software engineers to introduce best practice in code development, testing maintenance and reuse.

- Specialist consideration should be given to the problem of preserving the user interaction model of a software package.

- Guidance developed on the relative value of adopting an emulation or a migration strategy to support the preservation of software.

- Reconsideration of the categories of significant properties identified in InSPECT and those appropriate for software.

- Development of methodologies to exploit software testing as a measure of adequacy of performance.

1 Background to the Study

1.1 Introduction

Digital preservation has become pressing concern as more and more of the records of human activity are generated and processed electronically. Unlike traditional paper-based records where preserving the media is usually sufficient, electronic records are highly sensitive to the persistence of the electronic environment in which it is created and used, and highly dependent for its reusability (and thus effective usable persistence) on other computing artefacts also persisting and being usable. With the rapid change in the computing environment over the last 50 years, change which looks set to continue, there has been an increased realisation that electronic records maybe more vulnerable than paper based records, and require different and potentially more complex actions to be undertaken to preserve them in a usable state. Thus in recent years there has been a strong impetus to investigate methods, tools and practices to enable the long-term preservation of digital objects in a reusable state, a process often call *digital curation* to emphasise that the focus is on caring for the artefacts to ensure their future "replayability" and usefulness. This has been particularly strong in the museum and library sector, and also the academic research sector where there is a realisation that unless action is undertaken to preserve the results of research, there is a danger that their long-term use is likely to be compromised.

Much of the effort has gone into preserving the records traditionally preserved by libraries; human readable document written by humans for and intended to be read by humans. This is extended to other records intended for direct human interaction, such as still and moving images, and audio files. The recognition that large amounts of communication are now electronic has also led to the preservation of "electronic ephemera" such as emails, and websites.

More recently, work has gone into the preservation of primary data, both the numerical data which is typically the generated within scientific experiments, and also the records which are kept in databases. Data differs from documents in that it is intended to be primary machine processed, so that in order for the data to remain interpretable a greater effort needs to undertaken to preserve auxiliary and annotation material to preserve the context and meaning of the data so that it can re-processed appropriately.

Software is another class of electronic object which is frequently the result of research and as discussed below, is often a vital pre-requisite to the preservation of other electronic objects. However, the consideration to the preservation of software as a digital object in its own right has to date has been very limited. It is notable that many of the organisations which maintain access to software over a long period do not claim to *preserve* software in itself. Software is seen as complex – forbiddingly so for people who were not involved in its development but nevertheless want to maintain access to software – and also its preservation is frequently seen as a secondary activity and one with limited ultimate purpose. Consequently, we discuss in this document some of the motivation and approaches taken to preserve software.

This document reports the result of a study into the significant properties for preservation of software so that it can be systematically preserved in a reusable state for the long-term. . However, it is not possible to establish the properties of software without a wider study into

what software preservation means, as what characterises software is open ended and dependent on the context in which preservation is being undertaken.

1.2 Significant Properties

In this report, we use the definition of significant properties as give in [1]:

> ***Significant Properties**, also referred to as "significant characteristics" or "essence", are essential attributes of a digital object which affect its appearance, behaviour, quality and usability. They can be grouped into categories such as content, context (metadata), appearance (e.g. layout, colour), behaviour (e.g. interaction, functionality) and structure (e.g. pagination, sections). In an ideal world, libraries and archives would completely characterize the significant properties of their holdings so that they could be accurately **recalled** and, crucially, **reused** at a later date. Significant properties are thus those attributes of a digital object which need to be recorded and preserved over time for the digital object to remain accessible and meaningful.*

Significant properties have been considered in for a number of digital objects, such as text documents and raster images within other projects, including the following.

The **Investigating the Significant Properties of Electronic Content Over Time (INSPECT)** project[2] supported by JISC is an investigation into significant properties. INSPECT aims to:

- expand and articulate the concept of 'significant properties' ;
- determine sets of significant properties for a specified group of digital object types (raster images, emails, structured text, digital audio) ;
- evaluate methods for measuring these properties for a sample of representation formats;
- investigate and test the mapping and comparison of these properties between different representation formats.

JISC has also commissioned a number of studies into significant properties of a number of type digital objects

- Vector Images: see the final report [2]
- Moving Images
- SPELOS: Significant Properties of E-Learning Objects for Digital Preservation [3]
- Software: SigSoft (this study)[4].

These studies work with and supplement INSPECT as part of a framework for significant properties.

To date there has been no substantial study specifically into the significant properties of software; indeed while there is a large literature and a number of projects considering the preservation of other digital objects, particularly documents designed for human comprehension, and more latterly computer data designed for processing, there has been relatively little consideration given to the specific problems of preserving software in itself.

[2] http://ahds.ac.uk/about/projects/inspect/
[3] http://spelos.ulcc.ac.uk/wiki/index.php/Main_Page
[4] http://sigsoft.dcc.rl.ac.uk/index.html

This is despite the frequent recognition, in for example OAIS and the emulation approach used by Planets, that preserving software is an important prerequisite to preserving the digital object itself. These initiatives are discussed further below.

The reasons for this has perhaps been two fold: firstly, software is intrinsically highly complex and specialised, so capturing it comprehensively in a reusable manner is difficult; and secondly, preserving software is seen as a secondary activity, a necessary evil for the preservation of another digital object rather than an ends to itself. We shall consider these points in more detail below. Nevertheless, there are good reasons to preserve software (also discussed below), and thus a systematic approach needs to be developed to enable the preservation of software. This study thus represents an attempt to survey the features which should be considered in a software approach and to identify those features which are the significant properties which need to be considered for preservation.

2 Scope of Study

2.1 Definition of Software

Software is defined as: "*a collection of computer programs, procedures and documentation that perform some task on a computer system*."[5] Computer programs themselves are sequences of formal rules or instructions to a processor to enable it to execute a specific task or function. However, note that the definition also includes documentation, a crucial element in defining the significant properties of software, and thus in scope of this study. We refer to a single collection of software artefacts which are brought together for an identifiable broad purpose as a software *package.*

The term is sometimes used in a broader context to describe any electronic media *content* which embodies expressions of ideas stored on film, tapes, records etc for recall and replay by some (typically but not always) electronic device. For example, a piece of music stored for reproduction on vinyl disc or compact disc is sometimes described as the software for the record or CD player, in analogy to the instructions of a computer. However, for the purposes of this study, such content is considered a data format for a different digital object type, and is thus out of scope of this study.

2.2 Diversity of Software

Software is a very large area with a huge variation in the nature and scale, with a spectrum including microcode, real-time control, operating systems, business systems, desktop applications, distributed systems, and expert systems, with an equally wide range of applications. There are also varying constraints of the business context in which the software is developed from personally coded systems (typical in research), open-source systems, to commercial packages. We can classify this diversity along a number of different axes, which require different significant properties for preservation.

- **Diversity of application.** Software is used in almost every domain of human activity**.** Thus there are software packages in for example business office systems, scientific analysis applications, navigation systems, industrial control systems, electronic commerce, photography, art and music media systems. Each area has different functional characteristics on at least a conceptual user domain. Significant properties need to classify the software according to some application oriented classification or description of the domain.

- **Diversity in hardware architecture.** Software is designed to run on a large range of different computer configurations and architectures, and indeed "levels" of abstraction in relation to the raw electronics of the underlying computing hardware. At a micro level, assembler and micro-code are used to control the hardware directly and low level operations such as memory management or drivers for hardware devices. At a higher level of abstraction, applications are intended to be deployed on a wide range of computing hardware and architectures (e.g. workstations, hand-held or mobile devices, main-frame

[5] http://en.wikipedia.org/wiki/Software

computers, clusters). In order to recreate the functionality of system, significant properties of the hardware configuration may need to be taken into account.

- **Diversity in software architecture.** Even within a common hardware configuration, there are different *software architectures*, requirements on the coordination of software components which need to interact using well-defined protocols to achieve the overall functionality of the system. For example, in the StarLink system (see below) there is an assumption that the system runs on a particular storage management component. Another common example is a client-server architecture, where user clients mediate the user interaction and send requests to services on a server, which performs processing and responds with the results to the user. In order to recreate the functionality of the entire system, the reconfiguration of a number of interacting components into a common architecture will need to be recreated, and significant properties recorded accordingly.

- **Diversity in scale of software.** Software ranges from individual routines and small programs which may only be a few lines long, such a Perl routines written for specific data extraction tasks; through packages which provide particular set of library functions, such as the Xerces XML processor; major application packages, such as Microsoft Word, which provides a large group of related functionality to the user with large range of extra features, user interface support and backward compatibility; to large multi-function systems which provide entire environments or platforms for complex applications, such as the Linux operating system, which have millions of lines of code and entire sub-areas which would be major packages in their own right, but are required to work together into a coherent whole.

- **Diversity in provenance.** Software is developed by a wide range of different people organised in different ways. These would range from individuals writing specialised programs for personal use or to support particular functionality required by that individual; through community developments, where code is passed from person to person who has an interest in developing further functionality; formal collaborative working as is widely undertaken in major open-source initiatives, such as Apache or Linux, where a mixture of diverse contribution to the core code base is combined with a more centrally controlled acceptance and integration procedure; to software developed and supported by a large or small team within a single organisation, for the internal purposes of the organisation, or else to be distributed usually as a commercial proposition. A single software package may pass through a number of different individuals and organisations with a number of different business goals, models, and licensing requirements. These different development models need to be reflected in the significant properties of the system, so proper attribution and licensing condition can be respected.

- **Diversity in user interaction.** Software can support a wide range of interaction with the user. System software which controls the low level operation of the machine itself is designed to have no user interaction at all; library functions typically are designed to interact with other software components and have no or little user feedback, possibly delivering error messages; broader packages are typically designed to have a user interface component which mediate commands from and responses to the user often via simple command-line or file based interaction. Other systems have rich user interactions with complex graphical user interfaces requiring keyboard and pointer and high-resolution displays, or audio input and output. Other require specialised input or output hardware devices such as joysticks and other control devices for games playing, or specialised screens and displays for virtual reality display. Clearly, in order to accurately reproduce the

correct functionality of the software in the future, the appropriate level of user interaction will need to be recreated in some form.

Clearly there is huge diversity in the nature and application of software. However, we believe that there is sufficient commonality between these different scales that there are categories of significant properties which can be identified which are applicable to a wide range of different software packages.

2.3 Scope of Study

Nevertheless, for the purposes of this limited study into the significant properties of software, it is not practically possible to investigate more than a limited sample of the vast spectrum of software. Consequently, in this study we have decided to concentrate largely (although not exclusively) on **mathematical, scientific and e-Science software** as used within areas of science in the UK academic community.

We believe that this category of software is of interest to a large part of the academic and research community, which is the constituency of JISC, and should be representative for a large number of the significant properties which are generally applicable to wider categories of software. Further, there are particular features of the mathematical and scientific software domain which make it particularly suitable for this study, as follows.

- **Narrow Focus**. Scientific software is focussed on the application domain involved, and does not tend to gather extraneous features ("feature creep") which can confuse or obscure the purpose of the software.

- **Application Focus**. Scientific software is aimed at providing functionality for end users interested in the science, rather than say operating system, networking or control system software which is directed at functionality for controlling the computing infrastructure itself. This means that while the audience does tend to be computer literate, it is not computer specialised, so there is an onus on providing software which is intelligible to the scientific domain specialist.

- **Long life times**. Scientific users were early adopters of computing technology, dating back to the 50s and 60s in some cases, so there is a long legacy of software within the community. Further, scientific data and analysis does not go out of date – there is a need to revisit data and analysis which was collected a long time ago, so the software involved may not be obsolete. And finally, scientific research programmes can often be of long duration; space missions for example may take 15 years of preparation and take many times longer than that to execute and analyse the results. Control systems and data transmissions are still operational on the Voyager spacecraft more than 30 years after their launch and the onboard software needs to be kept operational on 1970s hardware at remote distance. As a consequence of these long life spans, the scientific and mathematical software community has a strong need to preserve and reuse software over long periods.

- **Established development teams and practices.** As a consequence of the longevity of scientific software, there are long-established teams which have experience of maintaining software packages over long period of time. They have long experience of maintaining and extending the function of software in the face of frequent technology and environment change and have established strong processes and tools to manage this change. Further,

since there is close involvement with the scientific community, who tend to be computer literate, there is an emphasis on openness and collaboration so that the function and organisation of the system are easily available.

As a necessary consequence of this focus on scientific software, other aspects of interest are less well-covered in this study, and not considered for any particular characteristics. Some which are particularly important are the following:

- **Systems Software.** Software which controls computer systems at a low level, such as device drivers, operating systems software, and networking. These are vitally important to maintaining computer infrastructure, and other software packages will typically depend on them. Significant properties of application software should record these dependencies, but details of the systems software are not considered in detail.

- **Business Application Software.** A class of common office client tools (e.g. Word processors, spreadsheets, small databases, email clients, web-browsers, calendaring systems, presentation systems, graphics editors), and back office tools (e.g. web servers, mail servers, document management systems, databases) clearly should be considered for preservation to maintain a record of a large amount of commonly produced material for largely human consumption.

- **Databases.** Databases may be regarded as a special category of software, with their own characteristics of data representation and management, and should be considered for specific significant properties for preservation. However, care must be taken to distinguish between preserving the database software (e.g. the DBMS and accompanying database modelling and querying language) and the data represented and stored within a database.

- **Software processing of specific digital object types.** An important class of software is designed to represent, manipulate and display human readable digital objects, including word processors (e.g. Word, WordPerfect), document markup and processing system (e.g. LaTeX, Acrobat), graphics editors (e.g. Visio), image presentation and processing systems (e.g. PhotoShop).

  Clearly these software packages are closely associated with other digital objects types (e.g. documents, raster and vector images), and some of the significant properties of these systems are closely associated with the significant properties of those digital objects. For example, visual significant properties of data objects such as fonts or line thicknesses will be dependent on the capabilities of the processing software to render these properties (and indeed ultimately on the capabilities of the hardware). This is clearly an important area which requires close consideration, and also close interaction with the effort to define significant properties for those object types. However, for the purposes of this study, these properties are not explored in detail.

- **Commercial software.** A good deal of the software which is available for within the mathematical and scientific domain is the result of community effort and is freely available and often open-source, allowing recompilation and migration by users. Within this study we mainly considered the issues arising from the use and maintenance of such software, and did not consider special characteristics and properties which arise from the software being commercial. The major exception was the NAG library, developed by a commercial company. However, even in those circumstances we concentrated on the maintenance and

reuse efforts which have been undertaken, rather than any requirements which arose from a specifically commercial environment.

Clearly, there are special characteristics arising from commercial packages, particularly that usage and licensing conditions should be respected, including conditions on copying and distribution, and also typically only binary distributions are available. However, detailed consideration of these aspects was not covered in this study.

- **Software with complex user interactions.** As already noted above, computer software can have a wide range of different human-computer interactions, with in sophisticated modern user interfaces, complex layouts, displays and interactions with the user. Inputs and outputs can be audio or haptic as well as visual, and use a wide variety of specialist devices as well as the familiar monitor-keyboard-pointer arrangement of most workstations. The particular layout and characteristics (e.g. colour, font, positioning) of a screen may or may not be key elements for reproducibility of the functionality of the software.

  Typically for much mathematical and scientific software, the user interaction model is less crucial behaviour to be preserved than the accuracy of the algorithms used for analysis; often such software is designed to interact via internal function calls from another program or via a command line as well as using a graphical user interface, and even with a GUI, the exact layout may not be critical[6].

  Consequently, a full treatment of the significant properties of the user interaction of software is a much larger topic, almost certainly requiring specialist user interface expertise, and is considered beyond the scope of this study.

2.4 Methodology

The study undertook a number of activities to complete its task of providing a outline framework for the significant properties of software.

- **Set bounds of project**. A scoping exercise to set the boundaries of the study, to set application domain of interest considered and also consider areas out of scope. This identified mathematical and scientific software and being of particular interest, and scoped the project as in section 2.3 above.

- **Surveying literature.** An ongoing task was to survey the literature available to consider the current state of software preservation and work on identifying the significant properties of software in particular. It soon came apparent that there is relatively little work on software preservation and virtually none on identifying significant properties in this context. However, it soon was apparent that a number of related areas were of interest to software preservation, particularly aspects of software engineering, especially version control, testing and reuse. We consider various relevant aspects of software engineering in section 4 below.

[6] An obvious exception to this is visualisation software used to provide human readable representations of mathematical and scientific data and artefacts, typically graphs or geometric representations of data in 2D or 3D virtual space. In this case, the characteristics of the display are clearly vital. However, we do not cover these aspects in detail in this study. Note that there are close similarities between the significant properties of these visualisation and those of other digital object types, especially vector images and animations.

- **Consider case studies in software preservation.** In order to establish the current best practise in software reuse, we considered in detail a number of specific examples of software developments, where packages have been developed, distributed and maintained over a long period, and repositories which are either specifically designed to hold software. A number of visits were undertaken to discuss with software package and repository managers their approach to software maintenance and the ongoing problems of long-term preservation of software, and how to accommodate change in the technological environment. Discussions and visits were undertaken with Starlink, BADC, CCPForge, and NAG.

  In preparation for these visits, we prepared a number of questions which, although not a formal questionnaire which was given to managers, nevertheless provided guidelines for discussion, and could be used in future as a basis for a more formal analysis of significant properties. We reproduce these questions in Appendix B.

- **Develop framework and test on examples**. From the literature and the case studies, and also discussions with other projects on preservation and significant properties, notably, InSpect, the study on Vector Graphics, the CASPAR and SCARP projects, an analysis was undertaken and a conceptual framework for software developed, so that the complex organisation of software artefact could be captured and organised, and significant properties assigned appropriately. A number of smaller illustrative examples were considered.

3 Digital Preservation of Software

During the course of the study, it became clear that software preservation was a term that was not necessarily considered a great deal, and when it was, it means different things to different people. Although we acknowledge that a single definition will not satisfy everyone, in this section we set out some baseline concepts of software preservation, and consider some of the motivations behind software preservation. This is a necessary prerequisite before considering the significant properties of software; the properties are clearly only significant in the context of the preservation task in hand.

3.1 What is software preservation?

Software preservation has four major aspects.

- **Storage**. A copy of a software "package" needs to be stored for long term preservation. As we discuss below, software is a complex digital object, with potentially a large number of components constituting a package (c.f. an *information package* as in OAIS); what is actually preserved is dependent on the software preservation approach taken (see Section 7.2). Whatever the exact items stored, their should be a strategy to ensure that the storage is secure and maintains its authenticity (*fixity* again using OAIS terminology) over time , with appropriate strategies for storage replication, media refresh, format migration etc as necessary.

- **Retrieval**. In order for a preserved software package to be retrieved at a date in the future, it needs to be clearly labelled and identified (*reference information* in OAIS terminology), with a suitable catalogue. This should provide search on its function (e.g. terms from controlled vocabulary or functional description) and origin (*provenance information*).

- **Reconstruction**. The preserved package can be reinstalled or rebuilt within a sufficiently close environment to the original that it will execute satisfactorily. For software, this is a particularly complex operation, as there are a large number of contextual dependencies to the software execution environment which are required to be satisfied before the software will execute at all.

- **Replay**. In order to be useful at a later date, software needs be replayed, or executed and perform in a manner which is sufficient close in its behaviour to the original. As with reconstruction, there may be environmental factors which may influence whether the software delivers a satisfactory level of performance.

In the first two aspects, software (once a decision has been taken on what software components to preserve) is much like any other digital object type. Storage media which are secure and maintain integrity, and methods to identify and retrieve suitable objects are required in all cases. However, the problem of reconstruction and replay is especially acute for software. Digital objects designed for human consumption have requirements for rendering which again have issues of satisfactory performance; science data objects also typically require information on formats and analysis tools to be "replayed" appropriately. However, software requires an

additional notion of a software environment with dependencies to other hardware, software and build and configuration information.

Note that other digital objects require software to provide the appropriate level of satisfactory replay, and thus for other digital objects there is a need to preserve software (and thus record its significant properties) too; as we shall see, there is also a dependency on the preservation of other object types (e.g. documentation) for the adequate preservation of software.

3.2 Why preserve software?

A key question to answer with respect to preservation of software is why it is a desirable thing to do in the first place. After all, software has a track record of being both being very fragile and very disposable.

Software is *fragile* as it is very sensitive to changes in environment; as hardware, operating system, versions of systems (e.g. programming languages and compilers) and configuration change. When the environment changes, software notoriously stops working, crashes corrupting vital pieces of data, or works but not quite as originally intended, with missing or non-quite the same functionality. The last case can be particularly damaging, as the software may seem to operate but actually produces subtly different results. For example, compiling with a different floating point module may produce quite different results in the analysis.

Software is *disposable* as often in the face of environment change, and also in the face of the complexity of large-scale systems, developers often throw away the previous software and start again from scratch (*"not invented here syndrome"*). After all, if you know the problem to be solved, and you have preserved the original *data*, it may be easier to write new software rather than handle legacy code, and you may be able to produce a faster, more user-friendly system which operates in a modern environment, and with the developer who understand the code to hand, rather than long gone from the organisation.

However, there are also good reasons to preserve software - especially in a research and teaching environment. Some of these reasons would include the following.

3.2.1 Museums and Archives

A small but significant constituency of software preservation is museums and archives which specialise on preserving aspects of the history of computing and its influence on the wider course of events. These institutions thus want to preserve important software artefacts as they were developed at the time of their creation or use, so that future generations of historians of science (and the general public) can study and appreciate the computers available that particular period, and trace its development over time. It is also recognised that archives of software may be useful to resolve copyright or patent disputes.

Such museums themselves often concentrate on preserving hardware. For example, Bletchley Park[7] and the National Museum of Computing[8] preserve or rebuild historic machines, including

[7] http://www.bletchleypark.org.uk/
[8] http://www.tnmoc.org/index.htm

early code-breaking machines from WWII, as does the Science Museum[9], the Museum of Science and Industry in Manchester[10], and the Computer History Museum[11] in Silicon Valley in California, USA. These machines are often kept in working operation, so there is a need to preserve the software.

Others archives are interested in preserving the software alone, typically via a web presence. Examples include the Chilton Computing website[12], which includes the Atlas Basic Language Manual describing the software architecture of the Atlas computer from 1965, the Multics History Project[13], which preserves the code for the Multics operating system, or Bitsavers[14], which preserves documentation and software for minicomputers and mainframes from the 50's to the 80's.

To give more detail on the Multics History Project, this has a concerted effort to locate and involved the original experts on designing and using the system before they die to capture their knowledge. The project seeks to preserve the binary, to "preserve the bits" and document the formats, But it also has an emphasis on capturing the implicit knowledge of the development organization and process, and to create a "map" of the software describing different ways of approaching it, via for example, capturing its source code, the coding interfaces, and its functions. It also wants to capture the development history and as much of the documentation as is available. This is for historical purposes; it seems unlikely that the Multics system will be revived in itself, and most of the functionality could be emulated elsewhere and the data generated using it processed on different systems. Nevertheless, Multics is an object lesson in software engineering (good and bad), and is undoubtedly valuable for future generations of computing engineers.

Other groups wish to preserve hardware and software as research interests or private enthusiasms, for example the Computer Conservation Society[15], a specialist interest group of the BCS, the Software Preservation Group[16] supported by the Computer History Museum, or a number of groups such as the Software Preservation Society[17] interested in preserving Games software for obsolete platforms, such as the Sinclair Spectrum, Acorn BBC Micro or Amiga.

In this context, there has been given some consideration of how to preserve software. See for example *Preserving Software: Why and How* by Zabolitsky [3], but this is largely limited to preserving historic software as a unit with the historic hardware, so the major concern is preserving representation of the code on some physical media, with appropriate backup and replication strategies. The problem of preserving the usage of the software in a future context is not considered in detail. The proceedings of the Computer History Museum's workshop "The Attic & the Parlor: A Workshop on Software Collection, Preservation & Access", May 5, 2006[18] gives an overview of approaches to software preservation being undertaken in museums and archives. Major concerns are how to collect important software packages, especially with a variety of licensing constraints, and how to interpret and display them to the public.

[9] http://www.sciencemuseum.org.uk
[10] http://www.msim.org.uk/
[11] http://www.computerhistory.org/
[12] http://www.chilton-computing.org.uk/
[13] http://www.multicians.org/mhp.html
[14] http://www.bitsavers.org
[15] http://www.computerconservationsociety.org/index.htm
[16] http://www.softwarepreservation.org/
[17] http://www.softpres.org/
[18] http://www.softwarepreservation.org/workshop/

The enthusiasts who would like to preserve games software recognise the problem of maintaining the usability of the software, which is the point of preserving old games. They also recognise the problems associated with copyright and copy protection. They adopt preservation strategies which use software emulation of obsolete platforms, and conversion of the binary to universal

3.2.2Preserve a complete record of work

Software is frequently an output of research. This is particularly the case in Computer Science where the software itself is an important test-bed of the hypothesis of the research - if you can't implement and demonstrate the advantage of the assertion, in computer science terms the assertion is not proven. However, this software as an output of research extends beyond Computer Science as many research projects across all disciplines now frequently have an aspect of computing and programming to demonstrate the hypothesis of research.

If university archives and libraries are going to maintain a complete record of research, then the software itself should be preserved. Frequently, in practice, theses do come with appendices of code listings or with CD-ROM's inserted into the back cover with the supporting software. However, while the theses are stored on library shelves, software content is not necessarily preserved against media change (can we read those disks in a few years time?) or change in the computing environment making the code difficult to run. Research projects again frequently produce software, or specialist modifications to existing packages to support their claims, or to carry our special analysis of data, so the results of the project are hard to interpret and evaluate without the software. However, at the end of the project, unless the software is taken up as a community effort, or in a subsequent project, there is little incentive or resource to maintain access to the software in a usable form.

Library preservation strategies should accommodate the preservation of software as well as other research outputs.

3.2.3 Preserving the data

Related to the previous point, is the reproduction and verification of the results of research which has generated and analysed data, and published the results. In order to verify the asserted results of a research project, then it should be reproducible. In many circumstances it may enough to rerun the analysis on current software if the original data has been preserved. But in other circumstances, testing accuracy or detecting fraud for example, it may be necessary to rerun the original software precisely to reproduce the exact result. Scientific reputations may be at stake here - and they should be judged on the results available to them at the time, using the software *as it was available to them,* rather than newer software. Newer software may have errors corrected, have higher performance or accuracy characteristics, or else have improved analysis algorithms or visualisation tools. All these factors may lead later analysis of the data to different conclusions to those originally deduced, but the scientists should nevertheless be judged on the view they were able to take at the time.

A further issue here is the reuse of data. Data which is collected on sophisticated experimental equipment or facilities[19] is expensive; other data which is recording specific events, such as

[19] Such as those operated by the Science and Technologies Facilities Council; the reuse of science data is a prime motive for the STFC's interest in data and software preservation.

environmental conditions at particular times and places, is non-reproducible. In these circumstances, it is desirable to allow the data to be preserved and reused in order to maximise scientific potential of the data. In these circumstances, it is necessary to preserve some supporting software, to process the data format, and to provide the appropriate data analysis.

This reason, as has already been noted is also relevant to the preservation of other digital objects. Preservation of document or image formats requires the preservation of format processing and rendering software in order to content accessible to future users.

Thus it is necessary to preserve software to support the preservation of data and documents, to keep them live and reusable. In this case, the prime purpose of the preservation is not to preserve the software per se, so it may be suitable in to not ensure that that software is reproduced in its exact form, but only sufficient to process the target data. Thus we introduce the key notion of adequacy, to provide "good enough" preservation" with key properties of the software preserved and others disregarded.

3.2.4 Handling Legacy

Perhaps the prime motivation to preserve software for most people is to save effort in recoding Code from the past still needs to be used, due to its specialised function or configuration and it is frequently seen as more efficient to reuse old code, or keep old code running in the face of software environment change than to recode. This is certainly the reason for most existing software repositories, and a significant part of the effort which is undertaken by software developers both in-house to end-user organisation, and also within software houses. Handling legacy software is usually seen as a problem, and many strategies are undertaken in order to rationalise the process, to make it more systematic and more efficient. As a consequence, an important source of information on significant properties for preservation is the best practice on *software maintenance and reuse*, a long recognised part of good software engineering. If you can find an existing package or library routine, why bother rewriting it? Of course in these circumstances you need assurance that the software will run in your environment and provide the correct functionality

3.3 The nature of software artefacts

Software is inherently a complex object, composed of a number of different artefacts. At its simplest, a piece of software could be a single binary file; however, even in that case, it is unlikely to be standalone, but accompanied by documentation, such as installation guides, user manuals and tutorials. Further there may be test suites, specifications, bug-list and FAQs. More complete software packages will also include source code files, together with build and configuration scripts, possibly from a number of different systems and packages, with more complete documentation, including specifications and design documents (including diagrams) and API descriptions. Software will also have dependencies on a wider environment, including software libraries, operating system calls, and integration with other software packages, either for software construction, such as compilers or build management systems, or in the execution environment, for example web-applications depending on web servers for execution and client browsers for user interaction. Thus a complete software preservation task may seek to preserve some or all of these artefacts, and, equally importantly, their dependencies upon each other.

Different software artefacts have different significant properties when it comes to preservation. An archive may in practice want to preserve the software at these different levels, and they have different consequences when it comes to the application of reuse effort.

- **Software Binaries.** If software binaries are preserved, then in order to reuse it the precise software environment needs to be reproduced. Software binaries will only typically only run in the right configuration of hardware platform and operating system, and sometime auxiliary libraries and (in the case of byte-code such as Java) run-time interpreters. Thus the reuse effort is in reproducing this environment, either by preserving the platform configuration itself (and it is surprising how often in a computing environment a old machine is maintained with a old software environment just to keep one vital piece of software running), or by *emulating* the old environment on a new platform.
- **Software Source Code.** Software in practice is often maintained at a source-code level. Thus the source-code as written by the programmer, in a human but not machine readable code, such as C, Java or Fortran is preserved, and the software reuse effort goes into to generating machine executable binaries from this code which both execute and maintain the original functionality. The maintainer must thus adapt and test the code in the face of change in hardware platform, operating system, programming language and compiler version, and auxiliary libraries. In this case, supporting items such as test-suites need to be preserved to ensure that the behaviour of the system in the new environment is maintained. In the process, the maintainers may have recourse to altering the source code to adapt to a new environment, and thus we need to consider the question of *versioning*.
- **Software Specification** In the minimal case, only the specification of the functionality may exist, and the reuse effort goes into recoding the system. This is particularly the case when software algorithms are published for particular tasks, but it could extend to an entire system, and it is perhaps questionable to how much of the software is really "preserved". In this case, the adaptation to the new environment is relatively straightforward, as the recoding itself is in that environment. However, considerable effort must go into the coding itself - and also the testing to ensure that the behaviour of the new code does indeed respect the specification.

4 Software Engineering

4.1 Introduction

Software engineering provides a substantial body of existing theory and practice on analyzing and organizing the design, development, structure and lifetime of software.

It can also be observed that there is a large overlap between the requirements for software preservation and those of software engineering, especially for large software development which has a long lifetime in production and requires extensive adaptive maintenance. Both require the high-integrity storage, and replay of software. However, there are also significant differences.

Software engineers are mainly concerned with maintaining the functionality of current systems in the face of software and hardware environment change, correcting errors and improving performance, and in adding additional functionality. They will typically deprecate and eventually obsolete past versions of the software. They are much less concerned with maintaining reproducibility of past performance, which may be the concern of software archivists. So in general, software preservation is *not what most software developers and maintainers do.*

Nevertheless, we argue that many of the approaches to software preservation mean in practice that the practises of software engineers are in fact appropriate to software preservation, many of the tools, techniques and methodologies of software engineers are useful to software preservation, and good software preservation practice should adopt, adapt, and integrate these techniques. Indeed, a conclusion which arises from this study can be summarised as:

Good software preservation arises from good software engineering.

There are a number of specific software engineering techniques which should be considered to determine their role in software preservation.

Software Development Lifecycle

Software development processes typically have well-defined lifecycle which gives a framework for describing the nature, role and relationships of the various software artefacts which form a complete software package. As we shall see later, there is a relationship between the stage of the software lifecycle and the preservation approach.

Software Documentation

Good software engineering supports good documents, from requirements and design documents, through installation instructions and change notes, through to manuals and tutorials, and to issues, errors and bug-tracking lists. Also there are software licences which give usage conditions on the software. Of particular importance is good systematic documentation of interfaces and functionality, such as those provided by Unix Man pages, provided by the NAG documentation (see below) or those supported by JavaDoc. If used properly, this can give a strong specification of the functional significant properties of modules or libraries of code and its interface description.

Software Version Management

A core part of good modern software development practice is Software Version Management, otherwise known as source code management. This controls the changes which take place to the source code of a package, so that conflicting versions of code are avoided and clashes resolved, and so that via branching, different releases of the code can be supported cleanly by controlling the structuring and dependencies of different versions of software. Well known software version management systems include CVS[20] and Subversion (SVN)[21]. Software version management systems are important for software preservation as the allow (in a well-structured development) clean version of the software to be indentified, defining which components (and versions of the components) form part of which particular release and as they can cover non-code artefacts, they can also cover documentation, defining which functionality is supported by which version.

Software Testing

Testing is the process of running the executable code (or part of the code) against representative sample input data to ensure that the software performs as designed, and thus provide assurance of the functionality of the data. The sample data should cover the range of expected inputs, both valid and exceptional input. Testing takes place at different levels, with unit, integration, acceptance tests. As we shall discuss later, testing has a key role in establishing the adequacy of preservation when it tests the performance of the preserved system to ensure that desired significant properties are retained.

Software Reuse

There is a considerable body of research and practical expertise within the Software Engineering community in Software Reuse. Software reuse initiatives such as NASA ESDS http://softwarereuse.nasa.gov/ present a software development process which builds reusable components - a motivation behind Object-Oriented Design frameworks such as J2EE. Asset library management is considered part of the lifecycle, with a faceted classification of significant properties forms part of the Software Reuse. The Basic Interoperability Data Model (BIDM) provides an IEEE standard (1420.1) for interoperable software cataloguing on the Internet. Other tasks often undertaken in software reuse include also code canonicalization and generalisation which helps make the components more generic and usable in different circumstances, as well as more portable for a migration strategy, for example by having less dependence on non-standard code features.

Although this software reuse expertise is directed at a different aim than software preservation, it has broadly similar in concerns, and so should be considered as a useful source of experience for software preservation.

[20] http://www.nongnu.org/cvs/
[21] http://subversion.tigris.org/

5 Frameworks for Software Preservation

In this study we considered a number of different groups engaged in software development and software repositories which engaged aspects of software preservation. This was carried out by consulting the literature and/or discussing their experiences with them directly. The groups were engaged in a range of activities from looking at preservation of digital objects, and thus needing to consider the function of software in a preservation method, through repositories which are storing software packages so that they are available to a specific community, to development projects which have the aim of maintaining and updating a software package over a long period.

Some of the examples were considering what might be described as general approaches to repository management and preservation which involve software preservation. In this section, we describe a number of different frameworks and tools which provide a context for software preservation, all providing some input to a general conceptual approach, although none of them providing a comprehensive method.

5.1 The Role of Software in Approaches to Digital Preservation

A number of projects have considered aspects of digital preservation, including Cedars [4], Chamileon, Planets, Inspect, PREMIS. The Digital Curation Centre has been established to provide guidance on practical aspects of digital curation and to coordinate efforts from these and other projects. We will not go through all of these projects in detail– this is covered in for example [2], [5] and [15]. However, we do consider the role software plays within two important digital preservation initiatives; the ISO standard OAIS, and the European Integrated Project CASPAR. Further, the approach of Planets is considered in a later section when we consider software emulation.

5.1.1 OAIS

The Open Archival Information System (OAIS) reference model is an ISO standard for the structures and processes required for an archive for the long-term preservation of information, either physical or digital, but with the emphasis on digital information [10]. While OAIS does not deal with the preservation of software directly, software does nevertheless play an important role in the standard, and takes a view on software, which is discussed in this section.

Representation Information is OAIS term for the information that maps a Data Object into more meaningful concepts. An example of Representation Information for a bit sequence which is a FITS[22] file might consist of the FITS standard while defines the format plus a dictionary which defines the meaning of keywords in the file which are not part of the standard.

The term “Significant Properties” is sometimes used to indicate those properties of a Digital Object which needs to be preserved, and which often therefore will need to have specific Representation Information, usually either Structure or Other Representation Information, to denote how it is encoded.

[22] Flexible Image Transport System is a standard format for astronomical data endorsed by NASA and the International Astronomical Union. See http://heasarc.nasa.gov/docs/heasarc/fits.html

Long term preservation is the act of maintaining information, as authentic and independently understandable by a *Designated Community*; by defining it in this way OAIS [10] makes the statement "we are preserving this digital object" testable. It can be argued that if that statement is not testable, it is not really meaningful, and also any organisation claiming to be preserving digital objects could not therefore be tested and validated

The Designated Community is an identified group of potential consumers who should be able to understand a particular set of information. The Designated Community may be composed of multiple user communities. A Designated Community is defined by the archive management and this definition may change/evolve over time. This concept is introduced by OAIS in order to allow the claim of digital preservation to be tested and also to allow and archive to limit the amount of Representation Information which it must maintain.

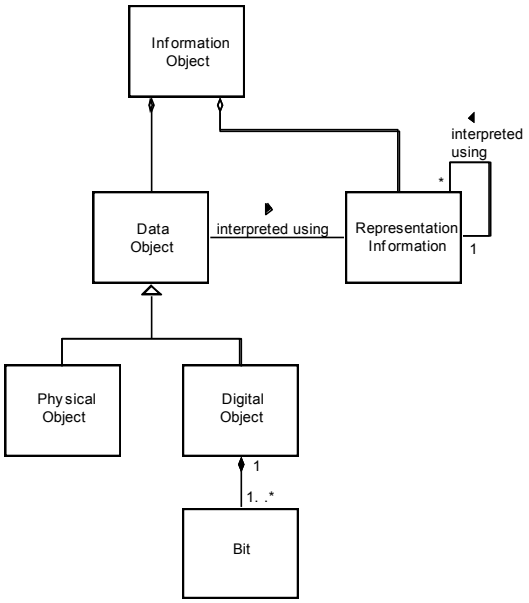

Figure 1 OAIS Information Model

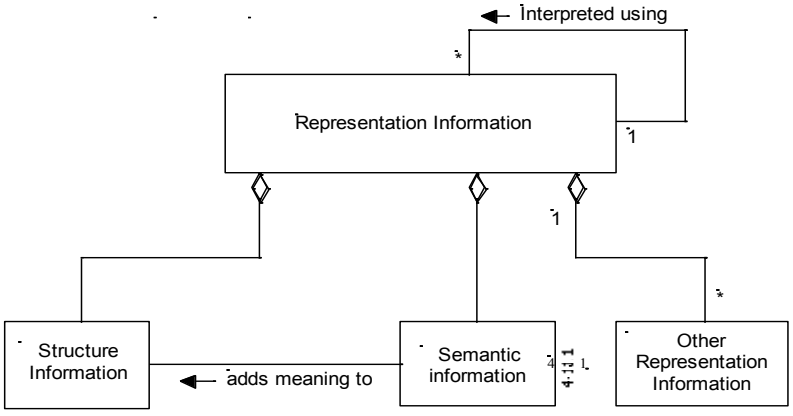

Figure 2 Representation Information Object

Representation Information is an Information Object that may have its own Digital Object and other Representation Information associated with understanding each Digital Object, as shown

in a compact form by the 'interpreted using' association, the resulting set of objects can be referred to as a Representation Network, as shown in Figures 1 and 2 above. Representation information can take one of a number of forms, including structural or semantic information, or indeed software.

Take the case of a piece of software, where perhaps the significant property is that when one clicks on a button a menu appears; let us consider what this might mean in terms of Representation Information.

In terms of software source code the Representation Network could include the definition of the text encoding e.g. the bit encoding of Unicode, plus the definition of the coding language syntax and standard libraries. In addition one would need the descriptions of the additional library calls that are embedded in the code. If one wishes also to preserve the build system then details of that build system, e.g. ant, the build libraries and the target machine such as operating system, hardware, peripherals (mouse, keyboard etc) would be needed. If the software operates on a Virtual machine, such as JAVA (the JVM), then one would need the version of the JVM, which in turn relies on a variety of underlying operating system capabilities.

Whether a repository would need to capture all this (and perhaps more) Representation Information depends on the definition of the Designated Community.

However much (perhaps most) software relies on a variety of other, perhaps remote, resources. This raises another level of complexity in software systems. This is discussed below.

OAIS specifically discusses Access Service Preservation which includes the following.

- The Dissemination API which is an Application Programming Interface (API) maintained by the OAIS as Access Software. This allows the Designated Community to use the digital objects in new ways, not restricted by earlier implementation limitations.

- Preservation of Access look and feel where we assume that the Designated Community wishes to maintain the original "look and feel" of the Content Information of a set of AIUs as presented by a specified application or set of applications. Conceptually, the OAIS provides an environment that allows the Consumer to view the AIUs Content Information through the application's transformation and presentation capabilities. For example, there may be a desire to use a particular application that extracts data from an ISO 9660 CD-ROM and presents it as a multi-spectral image. This application runs under a particular operating system, requires a set of control information, requires use of a CD-ROM reading device, and presents the information to driver software for a particular display device. In some cases this application may be so pervasive that all members of the Designated Community have access to the environment and the OAIS merely designates the Content Data Object to be the bit string used by the application. Alternatively, an OAIS may supply such an environment, including the Access Software application, when the environment is less readily available.

5.1.2 CASPAR

The European Project "Cultural, Artistic and Scientific knowledge for Preservation, Access and Retrieval (CASPAR)"[23] is a major integrated project to research, implement, and disseminate innovative solutions for digital preservation based on the OAIS reference model. Again, it does not concentrate on the preservation of software per se, but by necessity, software does play a major role, so the project takes a view on software preservation, as discussed below.

An approach to software preservation which the CASPAR project is adopting is to take the view that one would not expect to preserve the whole body of software, but rather the *interface*, which can then be re-implemented in future. This requires careful planning of the software interfaces to try to minimize dependencies on other software systems, for example communications protocols. This approach attempts to maintain interoperability between components while freeing future users from any current limitations.

A related way to capture the design of the software is to use a UML tool to produce a PIM (*Platform Independent Model*) which can be processed to produce implementations for a variety of languages and communication protocols. Another approach to free one of dependence on communication protocols is to use frameworks such as SPRING. These frameworks generate code, for example from interface definitions, to implement one of a choice of communication protocols.

Another area that CASPAR is investigating is that of software preservation to support specific data formats. For example there is proprietary software which processes sound in a workflow on a MacIntosh. The software has a time expiring licence and a dongle. The issue is to maintain the timings (which affects the sound) and the specific peripherals (including the dongle), which respecting or fooling the licensing system. In order to produce the right sounds specialised software drivers are needed which interact with specialised sound production hardware.

Large scientific data processing systems such as the ESA Multi Mission Facility Infrastructure (MMFI) linked to Grid systems is at yet another level of complexity being considered. Here one encounters tightly linked distributed software components in several languages on different hardware and operating systems, with interactions with large databases and specialised processing systems.

5.2 Virtualisation and Software Emulation

A number of approaches to preservation used *virtualisation* and *emulation* to preserve digital content. We look at some of the work undertaken in this area and consider the significant properties involved.

[23] CASPAR - Cultural, Artistic and Scientific knowledge for Preservation, Access and Retrieval - an Integrated Project co-financed by the European Union within the Sixth Framework Programme (Priority IST-2005-2.5.10). http://www.casparpreserves.eu/

5.2.1 Basic virtualisation

In Computer Science a Virtual Machine[24] is a software implementation of a machine (computer) that executes programs like a real machine. Virtual machines are separated into two major categories, based on their use and degree of correspondence to any real machine. A system virtual machine provides a complete system platform which supports the execution of a complete operating system (OS). In contrast, a process virtual machine is designed to run a single program, which means that it supports a single process – a common example of this would be the JAVA language JVM. An essential characteristic of a virtual machine is that the software running inside is limited to the resources and abstractions provided by the virtual machine - it cannot break out of its virtual world.

System virtualisation is a proven concept that was first developed in the 1960s by IBM as a way to logically partition large, mainframe computers into separate virtual machines. It was effectively abandoned during the 1980s and 1990s when client-server applications and inexpensive x86 servers and desktops established the model of distributed computing. Today, computers based on x86 architecture are faced with the same problems of rigidity and underutilization that mainframes faced in the 1960s.

Unlike mainframes, x86 machines were not designed to support full virtualisation[25]. There are 17 specific x86 CPU instructions that create problems when virtualized, causing the operating system to display a warning, terminate the application, or simply crash altogether. As a result, these instructions were a significant obstacle to the initial implementation of virtualisation on x86 computers. In 1999 VMware Inc.[26] patented an adaptive virtualisation technique that "traps" these instructions as they are generated and converts them into safe instructions that can be virtualized, while allowing all other instructions to be executed without intervention. The result is a high-performance virtual machine that matches the host hardware and maintains total software compatibility. VMware pioneered this technique and is today still a leader in virtualisation technology. In 2005 and 2006, extensions to their respective x86 architectures by Intel and AMD has also finally resolved the technical difficulties inherent in the x86 instruction set.

Microsoft also offers two Windows-based x86 virtualisation products, Microsoft Virtual PC and Microsoft Virtual Server, based on technology they acquired from Connectix.

The benefits of virtualisation to the user include:-

- **Compatibility**: Virtual machines are compatible with all standard x86 computers
- **Isolation**: Virtual machines are isolated from each other as if physically separated
- **Encapsulation**: Virtual machines encapsulate a complete computing environment
- **Hardware independence**: Virtual machines run independently of underlying hardware.

[24] http://en.wikipedia.org/wiki/Virtual_machine
[25] http://www.floobydust.com/virtualization/lawton_1999.txt and http://en.wikipedia.org/wiki/X86_virtualization
[26] http://vmware.com/

5.2.2 Virtualisation and Software Preservation

Today virtualisation allows users to easily run other x86 based operating systems in windows on their desktop. For example VMware Inc. make available both free[27] and commercially supported[28] versions of their virtualisation technology. These are available for both Windows and Linux platforms and give a very close approximation to a physical PC – with disks (either mapped to a file on the host OS or to a physical disk partition), networking (using the host Ethernet connection) and shared access to peripherals such as CD/DVD drives, USB devices and even to the host sound card. Other operating systems are installed into instances of the VM from their installation media and essentially give the ability to run either another copy of the same or a different version of the *host* operating system (for example, Windows 98 on Windows XP) or a different system entirely (for example, Linux under Windows – or vice-versa).

The *UK National Archives*[29] has entered into an interesting partnership with *Microsoft*[30] to try to preserve the exact *look and feel* of Digital Documents against changes of operating systems or the application used to render these files over time. Essentially this project uses a series of historical Microsoft operating systems with each running in their own *Microsoft Virtual PC* virtual machine implementation. Installed on these running Virtual Machines are the *exact* versions of the applications (such as *Microsoft Word, Excel* etc.) which would have been in use at the time the operating systems were current. Thus a very close match of OS and application can be selected simply by knowing when the digital document of interest was created.[31] [32] [33] Thus, with some confidence, it is possible to view it almost exactly as its creator did.

5.2.3 Planets and Dioscuri

Preservation and Long-term Access through NETworked Services (PLANETS)[34] is a European project under the 6th framework programme looking at developing a practical tools and services to support long-term preservation of digital content. PLANETS is developing a framework for preservation planning and for integrating preservation support tools within an OAIS based framework.

PLANETS aims explicitly to support the preservation of significant characteristics of digital objects via the addition of appropriate representation information in the OAIS model. In [9], software is considered as RI for a digital object, and a more detailed breakdown of software item is given than in the OAIS standard, so that the appropriate software for characterising and preserving the digital object can be captured. However, there is as yet no consideration of the problem of preserving the software itself, nor are the specific characteristics of software itself which need to be preserved defined.

In [7] a number of different scenarios are discussed to support different preservation approaches, from migration to emulation. However, the project appears in practice to promote

[27] http://www.vmware.com/products/server/
[28] http://www.vmware.com/products/ws/
[29] http://www.nationalarchives.gov.uk
[30] http://www.microsoft.com/en/gb/
[31] http://www.nationalarchives.gov.uk/documents/04july07.pdf
[32] http://news.bbc.co.uk/1/hi/technology/6265976.stm
[33] http://www.nationalarchives.gov.uk/news/stories/164.htm
[34] http://www.planets-project.eu/

an emulation based approach, and the project is supporting the ongoing developing of the open source x86 emulator Dioscuri[35]

To quote from Dioscuri's documentation:

> *Dioscuri is an X86 computer hardware emulator written in Java. It is designed by the digital preservation community to make sure that documents and programs from the past can still be experienced in the future. To do so, this emulator has two key features above any other emulator: it is durable and flexible. Durable because it can be ported to any other computer platform which has a Java Virtual Machine (JVM) without any extra effort. This reduces the risk that the emulator fails to work on one computer system in the future, because it will continue to work on another. Dioscuri is also flexible because it is completed component-based, just like a real computer. Each hardware component is emulated by a software surrogate called a module. Combining all modules allows the user to configure any computer system (if compatible). Add a new module or upgrade another one and the emulator is capable of running it.*[36]

Dioscuri has been developed by the Koninklijke Bibliotheek (KB), National Library of the Netherlands, and the Nationaal Archief of the Netherlands, and taken up by the PLANETS project where it is being further developed lead by a software company, Tessella Support Services plc.

Dioscuri is motivated to support the long-term preservation of document formats, especially commercial offerings are no longer supported by their vendor and archives are required to preserve the documents in as unchanged a form as possible, including content, structure and layout. Migration may alter the original document into a new version, so they advocate emulation to mimic the environment in which the original document was rendered.

Of particular interest to us are the environmental features which Dioscuri aims to emulate computer components. The complete list (again from the documentation) is shown below:

- 16-bit Intel 8086-based CPU (real-address mode only)
- 1 MB RAM (expandable)
- Storage devices: floppy, HDD
- Input devices: XT/AT/PS2 compatible keyboard
- Output devices: virtual screen
- VGA video graphics adapter
- DMA-support
- IRQ-handling based on a Intel 8259 PIC
- Timing mechanism based on an Intel 82C54 PIT and Crystal Clock
- Real-time clock with integrated CMOS
- System BIOS using Plex86/Bochs BIOS
- Video BIOS using VGA LGPl'ed BIOS

Dioscuri provides a library of modules which can be imported to support particular features as the application software (and indeed the end digital object) requires. Thus we can consider

[35] http://dioscuri.sourceforge.net/index.html

[36] http://dioscuri.sourceforge.net/manual.html

these to be the significant properties of the application software which the emulator seeks to preserve.

The developers recognised that for emulation to be an effective preservation strategy, the emulator itself has to be sustainable in the long-term. Consequently, they adopt the notion of the Universal Virtual Machine (see for example [9]) , in this case using Java Virtual Machine (JVM) (while recognising the risk of different behaviour arising from different versions of the compiler) and use plugabble modules to flexibly support different environmental features, such as processor, memory, keyboard, etc

5.3 Existing Software Repositories

Current software repositories exist to collect and distribute software. For example, there are several web-based archives of open-source, shareware or software available to the community under common licensing agreements, such as Eduserv Chest[37]. In domain specific areas, repositories have been established which collect and distribute specialised software, including the US National HPCC Software Exchange for high performance computing[38], Starlink[39] for astronomy, GAMS for mathematical software[40], OMII for Grid software[41], CCPForge for Collaborative Community Projects in computational chemistry (discussed below). The primary purpose of these repositories is to provide a central point for a community interested in a particular topic to share and exchange software and also include tools for community software development; the most well known and most general of these is the SourceForge site for open source software development, discussed below.

Software characterisation in these repositories varies from using simple alphabetic lists, through simple categorisation for broad functional areas, through to indexing via detailed hierarchical thesauri of terms such as the GAMS thesaurus, used by GAMS, NHSE and NAG[42]. Thus significant properties characterised are typically restricted to discovery terms, together with programming language and supported operating systems. The quality control on the software in the repository varies. However, these properties are not considered in the context of preservation.

5.4 Classification Schemes for Software

Classification schemes for software have been proposed and are used to classify the function and to some extent the context of software. These schemes provide a functional classification of software within a controlled vocabulary. They are generally designed to be broadly based and do not give a lot of detail into the precise functioning of the software package itself. Thus they are largely useful as a significant property for categorisation, search and discovery rather than enabling the replay of the software.

Some well-known examples would include the following.

[37] http://www.eduserv.org.uk/chest.aspx
[38] http://www.nhse.org/index.htm
[39] http://www.starlink.rl.ac.uk/
[40] http://www.gams.com/
[41] http://www.omii.ac.uk/
[42] http://gams.nist.gov/,

The **Computing Classification Scheme of the Association of Computing Machinery (ACM)**[43] is used usually to classify computer science articles within ACM publications. It provides a broad view of computing, mixing different facets of computing, such as application domain (e.g. business applications), computing science area (e.g. Artificial Intelligence), and hardware platform. While a good starting point for the general aspects of the significant properties of software, much more work would be required to make this complete.

More domain specific classifications such as the **Guide to Available Mathematical Software (GAMS) Thesaurus**[44], give focussed and detailed information into the functionality supported. For the functional significant properties of software packages (down to a level of detail of a library item) in this area, this may be sufficient.

Other thesauri also exist such as INPEC or the USPTO (United States Patent and Trade Office) classification scheme. Again these are too general to be of great use to specify the significant properties of software in anything other the most general functional terms.

5.5 SourceForge and CCPForge

Over the last ten years, the notion of a *software forge* has been developed. A software forge is a web-site which hosts software development projects which involve developers in highly distributed locations. These are typically open-source efforts (though there is no reason why they always should be), with a large number of volunteer developers operating across a number of countries and organisations. Such community efforts require a common place to lodge and share their codebase, provide documentation and community communications; the software forge site provides this functionality.

The most well-known of these sites is SourceForge[45], which supports 100,000 projects with 1,000,000 users. SourceForge provides a space for source code management using CVS, documentation, simple web site and community forums for discussion and bug tracking. Projects in SourceForge are categorised against a fairly crude topic mapping. Other sites provide similarly functionality, typically for a specialised community, where there is the opportunity for more specialised support and interaction with a like minded community.

CCPForge[46] is a software forge which provides a self service space for software from the Computational Chemistry community. Again, CCPForge provides a CVS repository for source code management and a number of tools to support community engagement. Unlike SourceForge this repository is actively managed and users normally need to create an individual username by self-registration and then have this authorised for individual project access. CCPForge is modelled on SourceForge principles but also provides some facilities for building and maintaining binary versions of the code. Projects can provide *Bug Tracking, Support* and *Feature Request* facilities – in addition to the more common *Documentation Management, Mailing Lists* and, of course, *Platform Specific* pre-built binary packages. CCPForge sees its task as very much to provide a meeting place for developers and a storage and source code management facility for the projects in the Computational Chemistry. It does

[43] http://www.acm.org/class/

[44] http://gams.nist.gov/
[45] http://sourceforge.net/
[46] http://ccpforge.cse.rl.ac.uk/

not regard that it has any role of controlling the projects or putting specific requirements on them - for example for software preservation. It is up to the projects themselves to take what ever actions they see fit to maintain their documentation and make them suitable for migration. That is part of the software engineering practice of the project itself.

Nevertheless, providing better facilities to support preservation in software forges could be a suitable future developement of such systems, to build on the existing good practice in source code control and documentation to record the significant properties of software for preservation and thus convert the forges into archiving repositories.

6 Case Studies of Software Developments

Another group of case studies were those providing software, either as part of their line of business in developing and supporting software products over a long period or time, or like the BADC providing software as an adjunct to data management and distribution services.

It is notable that none of these groups would claim to be doing software preservation, rather either maintaining currency for a current software package, providing a common holding place for community development of current software, or else provide additional software to support the preservation of another digital object type, such as documents or scientific data. Thus it soon became apparent that it was not necessarily constructive to ask them about the significant properties of software that they were interested in or they took effort to preserve. They simply did not think in those terms. Rather the discussion took the direction of how the software they maintained was adapted and maintained over time and how it could accommodate changes in environment and external technology as well as the changing functional requirements of the intended audience.

We give a description of the problems and strategies of maintaining long-term usability of software for a number of initiatives.

6.1 BADC

The NCAS British Atmospheric Data Centre[47] (BADC) is a NERC funded centre which has the role: *to assist UK researchers to locate, access and interpret atmospheric data and to ensure the long-term integrity of atmospheric data produced by Natural Environment Research Council (NERC) projects.*

The BADC has substantial data holdings of its own and also provides information and links to data held by other data centres. BADC holds Datasets produced by NERC-funded projects, which are of high priority since the BADC may be the only long-term archive of the data; and also third party datasets that are required by a large section of the UK atmospheric research community and are most efficiently made available through one location (e.g. Met Office and ECMWF datasets). To support this aim BADC develops, supports, supplies and provides access to a variety of software necessary to locate access and interpret this atmospheric data. Thus associated with the need to preserve the data is also a need to consider appropriate preservation actions required for software. In this section we consider a number of the software tools and their preservation properties[48].

The BADC would categorise the types of software it interacts into the following classes:

1. Software which it utilises to facilitate the direct discovery, permit remote or local access to data

[47] http://badc.nerc.ac.uk/home/index.html
[48] This work undertaken in conjunction with the JISC funded SCARP project http://www.dcc.ac.uk/scarp/ which is conducting a wider study into the preservation needs of BADC.

2. Software which processes archived data for the "on-the-fly" provision of processed data product
3. Generic Analysis tools
4. Large Scale Modelling specifically the Met Office Unified Model
5. Data Set Specific software tools and scripts which are informally archived
6. Community based models and analysis tools.

The BADC considers the long term archiving of software an impractical option principally due to the complex dependencies of software. It takes the view that it expects much current software will be superseded by newer software which will be capable of recreating and enhancing much of the existing analysis and access functionality. There will however be data set specific analysis models based in the user community for which it is anticipate this will not happen. The cost of archiving such models by migrating to new technologies as they evolve is prohibitive and emulation technologies have not yet matured sufficiently to allow confidence that storage of binary executables will be sufficient for preservation purposes. The BADC additionally considers it to be outside their core remit to harvest and archive such models. We examined key examples from the above categories exploring the functionality the software provides users, the human/technical dependencies and other issues associated with them.

6.1.1 On the fly provision of processed data

6.1.1.1 Trajectories

Functionality: The BADC trajectory model derives the parcel paths from a set of analysed winds. These winds are from 40 years of archived data in a mixture grib and pp data formats held at the BADC. The trajectories software allows you to track a specified wind parcel over time and project path onto global map. They also allow you to create plots of pressure, temperature and potential temperature

User and Technical Dependencies. The software is written in IDL with a perl web interface. User will require some knowledge to use this software but it currently well documented and supported by the BADC helpdesk.

Versions and Preservation. The BADC has had only one version of the trajectories software which although the software author has left the BADC it is still capable of maintaining. Again it anticipated that this software has no real preservation merit in itself.

The ECMWF (European centre for medium range weather forecasts) provides detailed technical documentation about the Integrated Forecasting System (IFS) used to generate the data sets held at the BADC. This technical detail regarding the following types of processes and procedures is however critical data provenance information for any trajectories generated: Observation processing; Data assimilation; Dynamics and numerical procedures; Physical processes; The Ensemble Prediction System; Technical and computational procedures.

6.1.1.2 Data Extractor

Functionality: The Data Extractor provides the following functionality to BADC users:

1. Extraction of NetCDF datasets.
2. Differencing between datasets.
3. Browsing and selection of subsets.
4. Selection in space and time.

User and Technical Dependencies. Data extractor is written in Python .If a user wishes to install there own version of Data Extractor it relies on the following software

- A webserver (probably Apache).
- CDAT – Climate Data Analysis Tools (for more detail see below)
- Python – if not installed with CDAT.

Versions and Preservation. Currently on its first version and again no real preservation merit

6.1.1.3 Geosplat

Functionality GeoSPlAT (GeoSpatial Plotting and Animation Tool) was developed to fill a requirement of both the BADC and the NERC DataGrid. Geosplat is normally used in conjunction with the Data extractor to provide a data extraction suite providing User-defined plotting and User-defined animation.

User and Technical Dependencies as with the Data Extractor above

6.1.2 Generic Analysis tools

6.1.2.1 Xconvsh/convsh

Functionality. Xconvsh/convsh are binary utilities developed at the University of Reading which allow the user to access, subset, interpolate, manipulate, convert and visualise data files of the following formats:

- NetCDF format
- GRIB format
- GrADS format
- UK Met Office Unified Model Data Output format
- UK Met Office PP format
- DRS format

Xconv is an X windows utility which allows the user to interactively manipulate the data and produce an on-screen plot of the data field. It has an intuitive, user-friendly interface and is able to read and write a wide variety of different data formats. Convsh is the command line equivalent of Xconv, and allows GRIB files to be processed in 'batch' mode. The BADC has additionally developed some Convsh scripts to batch process files related to individual datasets. User and Technical Dependencies.

In order to use an older versions of xconv than 1.90, then it is possible that byte swapping of the input data files may be required before they can be read on your systems using xconv/convsh. This may be done using a (unix) based byte-swapping utility called swapbytes. This allows a 4 byte word file to be converted from big to little endian, and vice versa. The tar file contains the source code and a basic makefile. (Should you require byte-swapping of 8-byte word files, use this utility instead; usage: swap8 < infile > outfile.)

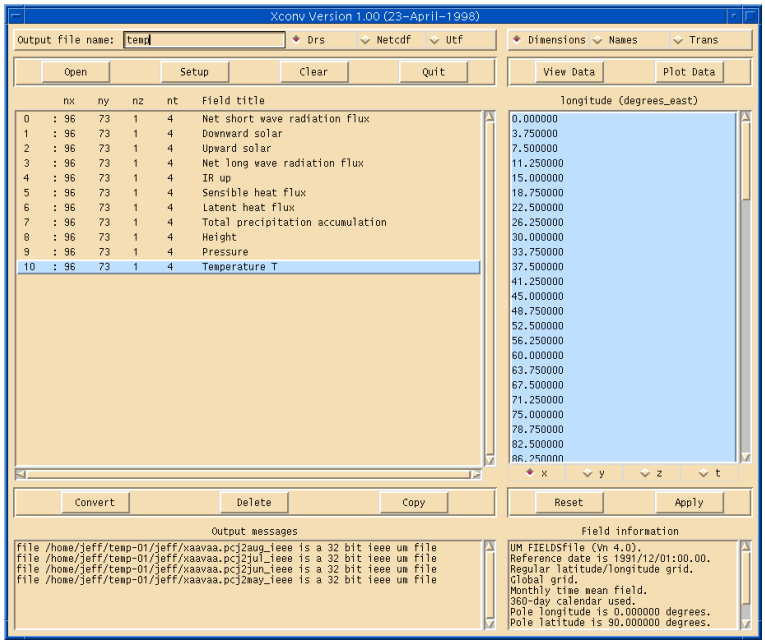

Figure 3: Screenshot of XConv

Version and preservation. A number of different binary versions are stored for different platforms.

Version 1.91

- **IBM AIX Powerpc Executables**
- **Linux ia64 Executables**
- **Linux x86 Executables**
- **Linux x86_64 Executables**
- **Mac OS X Power c Aqua Executables**
- **Mac OS X Powerpc X11 Executables**
- **SGI IRIX Mips n32 Executables**
- **SGI IRIX Mips 64 Executables**
- **Sun Solaris x86 Executables**
- **Windows_x86 xconv Executable**
- **Windows_x86 convsh Executable**
- **Windows_x86 xconv Starkit file**
- **T3E**

Older Version 1.90 are available for

- **Linux (dynamic library)**
- **Linux (static library)**
- **Dec Alpha**
- **Fujitsu (v1.05)**
- **HP**
- **Linux_ia32**
- **Linux_ia64**
- **SGI**
- **SGI_origin**
- **SUN**
- **SUN_static_f77 (v1.05)**

6.1.2.2 GrADS

Functionality. The Grid Analysis and Display System (GrADS) is used for easy access, manipulation, and visualization of data. It performs these functions for the GRIB, NetCDF and HDF-SDS data formats. GrADS has a programmable interface (scripting language) that allows for sophisticated analysis and display applications GrADS will typically be used for operations such as:

- Plotting a variable from a file on a shaded plot and overlaying contours from a second variable.
- Aggregating multiple files into one control file so that slices of data can be read in across multiple files.

- Differencing 2 different datasets.
- Calculating departures from a climatology from a dataset.
- Regridding a dataset.
- Calculating the statistical data from variables.

Data may be displayed using a variety of graphical techniques: line and bar graphs, scatter plots, smoothed contours, shaded contours, streamlines, wind vectors, grid boxes, shaded grid boxes, and station model plots. Graphics may be output in PostScript or image formats

User and Technical Dependencies. Operations are executed interactively by entering FORTRAN-like expressions at the command line. A rich set of built-in functions are provided, but users may also add their own functions as external routines written in any programming language. The full GrADS distribution contains pre-compiled binary executables, the source code, documentation, and the supplementary data sets that are required to run GrADS (fonts and map files). The binary distribution contains only the suite of executables. Two MS Windows builds of version 1.8 are available: xwin32 requires an X-window server in order to display graphics, and win32e uses native windows. The MS windows versions are packaged with an install script. All other versions, the tar file needs to be uncompressed and unpacked after download.

Versions and Preservation. GrADS has been implemented for the following operating systems:

- DEC
- Intel / LINUX
- SUN
- Macintosh OSX
- SGI / IRIX
- SGI / IRIX
- IBM / AIX
- MS Windows

6.1.2.3 CDAT

CDAT (Climate Data Analysis Tools) was developed at the Program for Climate Model Diagnosis and Intercomparison (PCMDI). It was specifically designed for climate science data. CDAT makes use of an open-source, object-oriented, easy-to-learn scripting language (Python) to link together separate software subsystems and packages to form an integrated environment for data analysis.

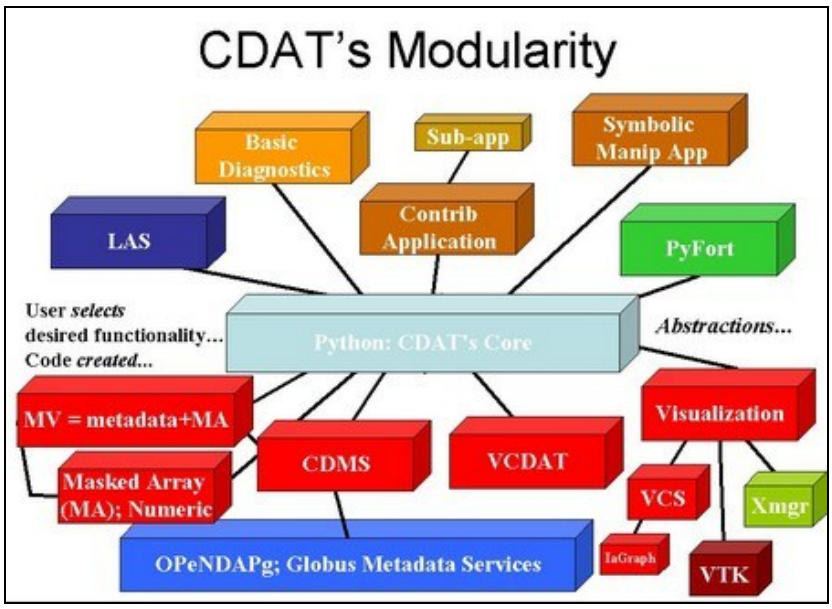

Figure 4: Dependencies of modules in CDAT

CDAT provides a number of modules, which are illustrated in Figure 4.

- cdms - Climate Data Management System (file I/O, variables, types, metadata, grids)
- cdutil - Climate Data Specific Utilities (spatial and temporal averages, custom seasons, climatologies)
- genutil - General Utilities (statistical and other convenience functions)
- numPy - Numerical Python (large-array numerical operations)
- vcs - Visualization and Control System (manages graphical window: picture template, graphical methods, data)

Functionality. BADC users of CDAT will typically be to use it for operations such as:

- Plotting a variable from a file on a polar stereographic projection.
- Aggregating 1000s of files into one XML file so that slices of data can be read in across multiple files.
- Differencing 2 different datasets (as the Python Numeric package allows array algebra).
- Calculating departures from a climatology from a dataset.
- Regridding a dataset and then calculating a spatial average.
- Calculating the covariance between two variables.
- Calculating the mean and standard deviation of a variable.

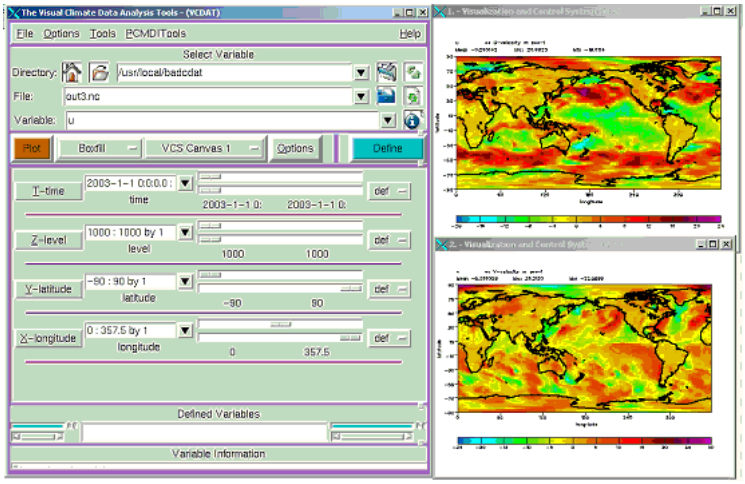

Figure 5: Screenshot of CDAT

Some key features of CDAT for BADC users include:

- a choice of interfaces: command-line, scripting or graphical user-interface (Visual CDAT (VCDAT)).
- an XML-based format and tools for aggregating large datasets.
- manipulation of large data arrays possible due to use of Python Numeric package.
- interfaces to external packages such as the Live Access Server (LAS) for web-based access to datasets.

User and Technical Dependencies. A user can potentially utilise CDAT in number of ways depending on their skill level and scientific objectives CDAT is scriptable along you to perform bespoke operation or users use VCDAT Graphical User Interface (VCDAT) which is the graphical user interface for CDAT. It helps users become familiar with CDAT by translating every button press and keystroke into Python scripts. VCDAT does not require learning Python and the CDAT software. CDAT possesses a number of predefined analysis, conversion, sub-setting and array operations. It also has interfaces to FORTRAN and C/C++ allowing it to interact with user created models and programs.

One factor which has proven to be a barrier to its uptake has been the length of time in effort it takes to initially install CDAT. CDAT requires a Linux/Unix distribution. There are many different platform-specific operations that need to be carried out during the installation process, setting of environment variables, changing shell modes, installing libraries etc[49]. In order to remove these barriers a CDAT "Lite" is under development.

Versions, Platforms and Preservation. CDAT is fully supported on the following platforms:

- Macintosh OS X 10.4.x/10.3.x/PowerPC
- RedHat Enterprise Linux WS 3.x/i386 and Enterprise Linux WS 4.x/i386

The CDAT team will help port to these platforms, but is not actively supporting them:

- RedHat Linux 8.x and 9.x/i386
- Sun/Solaris 8 and 9
- SuSE Linux 8.x and 9.x/i586

[49] Full details of these complexities can be found at http://www2-pcmdi.llnl.gov/software-portal/cdat/download/installation-guide

CDAT has also been known to be ported to the following additional platforms:

- Cygwin (Windows) 1.5.x/i386
- RedHat Fedora Core 1, 2, 3, 4/i38
- SGI Altrix (64-bit) running RedHat Linux
- HP-UX 11
- IBM AIX 5L
- Linux flavours not mentioned above (e.g., Mandrake, Caldera, and Debian)
- OSF1 V4.x
- SGI IRIX 6.5

Note this is available for multiple platforms but not Windows

6.1.3 Met Office Ported Unified Model

The Unified Model is the name given to the suite of atmospheric and oceanic numerical modelling software developed and used at the Met Office. The model supports global and regional domains and a wide range of temporal and spatial scales that allow it to be used for numerical weather prediction as well as a variety of related research activities including climateprediction.net (a distributed project to consider a number of climate models to investigate the likely effects of climate change) . The Ported Unifies Model software allows the Unified Model to be run on a user's own system.

Functionality. The main model components are:

- **User Interface.** An X-Windows application for setting up model integrations. It comprises over 200 separate windows and may be customised by the user.
- **Reconfiguration**. A generalised interpolation package used to convert model data files to new resolutions and areas.
- **Atmosphere Model.** Grid point split-explicit dynamics and physical parameterizations.
- **Ocean Model**
- **Atmosphere-Ocean Coupling**. Software to run atmosphere and ocean models in coupled mode, including a dynamic sea-ice model.
- **Diagnostics and Output**. Internal model package to output a range of diagnosed and derived quantities over arbitrary time periods, sub-areas and levels.

User and Technical Dependencies. The UM is a large and complex software system, primarily designed for use in a research and operational forecasting environment. According to the met office anyone wishing to install or use the PUM requires, at least, the following competencies.

- Have a good working knowledge of:
  - Unix
  - Fortran77 and Fortran90
- Preferably have experience in:
  - Problem solving
  - Use of debuggers
  - Compiler usage and manipulation
- Have someone available who has:
  - C programming experience
  - Unix system administration experience

    - Access to system files
    - Data manipulation experience
    - Knowledge of visualisation techniques
- Desirable, but not essential to have knowledge of:
    - Perl programming language
    - Bourne shell script languages
    - Tcl/Tk or programming in X

A ported version of the Unified Model, the Ported Unified Model (PUM), has also been developed suitable for running on workstations, PCs running the Linux Open Source operating system as well as the massively parallel computer systems used for Operational forecasting at the Met Office. The focus of most recent, work has been to optimise the Unified Model for use with vector supercomputers like the Met Office's NEC SX-8. This has been built upon previous work, including incorporating a non-hydrostatic dynamical core into the PUM. The latest release of the Ported Unified Model, 6.1, represents a significant upgrade of both the scientific and technical capabilities of the model. This included significant porting work to support the use of clusters of commodity-based computers.

Versions and Preservation. A complex model such as the Unified Model is under continuous development by a large team of scientists and programmers. A code configuration management system is vital to the successful coordination of these code developments. The Unified Model is tracked by a version release number in the form X.Y where X denotes major changes and Y denotes general developments. Source code developments, or modification sets as they are known are under the overall control of the Unified Model system manager and day-to-day control of a code librarian. A proprietary source code revision system is used to merge new code with the development stream, but plans are underway to automate the process further using modern source code control software tools that will be portable across computer platforms.

6.1.4 Data Set Specific software tools and scripts

A lot of software is created for specific data sets for a number of reasons the most common being

- Uncommon data formats which generic analysis tools cannot read
- Skill sets of project scientists
- Specialised requirements for dimensions
- Specific visualization requirements
- Prejudice of scientist or cultural inertia within a scientific field

6.1.4.1 MST data plotting software

Functionality. One example of data set specific plotting and analysis programs is the MST GNUplot software. This software plots Cartesian product of wind profiles from netCDF data files. Software needed to developed due to specialised visualization requirements where finer definition of colour and font was needed than that provided by generic tools.

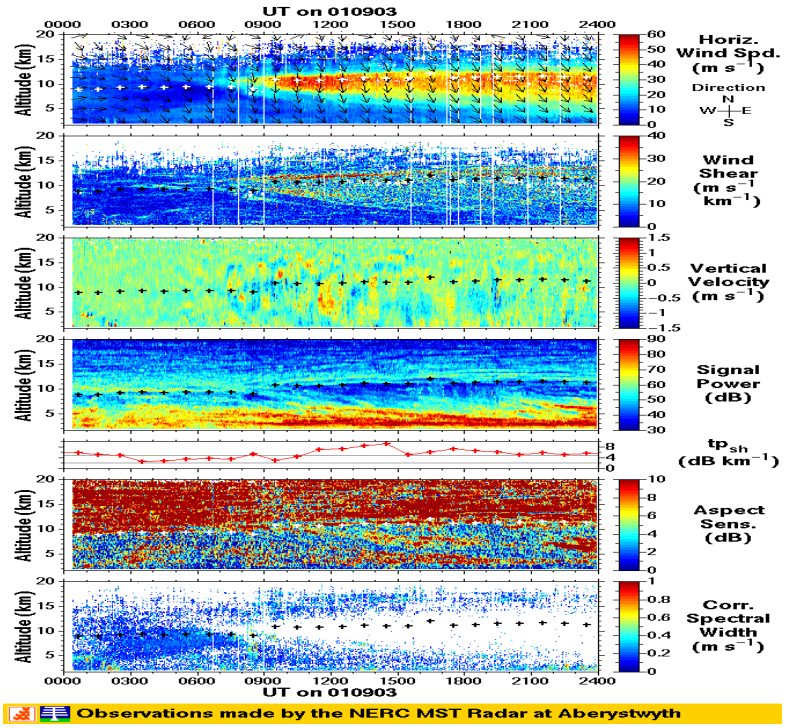

Figure 6: MST plotting software

User and Technical Dependencies. This software requires a Unix or Linux distribution and Python with python-dev module installed with numpy array package and pycdf (required for NetCDF files). It also requires GNUplot to be installed to set environmental variables. A previous version was created using Matlab but due to licensing restrictions it was necessary to move to GNU plot

6.1.4.2 Scripts in the data collection

Although there is no policy in place to formally archive software specifically associated with data sets. BADC have "instinctively" stored executables, documentation, and scripts along side the data in directories. In the ACSOE (Atmospheric Chemistry Studies in the Oceanic Environment) dataset example below the software directory contains the follow

- Fortran code, executables and documents to check file integrity
- SQL queries to facility data access to ACSOE database
- IDL scripts with documentation

6.2 NAG

The Numerical Algorithms Group (NAG)[50] is a not-for-profit company limited by guarantee with long history of providing software for the solution of mathematical, statistical and data mining problems. It originated out of partnership between a number of universities to develop a general purpose library of numerical and statistical subroutines. This library was first released on 1st October 1971, and has been in continuous use since. NAG have since

[50] http://www.nag.co.uk/

diversified into a number of other tools and projects, such as recently being engaged to support the porting of computational software onto the new HECToR high-end computing resource for the UK academic sector. However the NAG software library still remains the core of the business of the company.

As a consequence, with over 35 years of experience in maintaining and developing a software product, NAG have developed good practice in porting and migrating to keep a software product alive, although preservation per se is not a prime aim of the company. Over this period they have had to accommodate an enormous amount of change in the computing environment. The library was originally designed for ICL 1906A/S machines – and each machine could have specific performance characteristics. Now they have to accommodate commodity machines (such as x86 architecture) and languages (C and Java), as well as many specific environments which have to be tailored for individual customers. Also in that time, there has been a continual turnover of staff, and many of the original developers are no longer available. So while not professing to be software preservers, their practice and experience provide important insights into the processes and tools required for software preservation.

6.2.1 Maintaining the NAG Library

In the face of the large environmental change over the last 35 years, the main effort in maintaining the relevance and usability of the library has been in migrating the NAG library to new platforms and software environments and architectures, and what porting has meant to NAG has varied over time. Initially, the library had to be tailored to specific machines and compiler, but as over time machine architectures and language compilers, NAG have been able to provide versions for different classes of architectures and languages. Now, different versions of software packages which the library works with (e.g. Matlab) are important factors. In the future, multicore/parallel environments are likely to be important. A refactor of the codebase a few years ago has been part of an ongoing effort to develop good practices to make the process of porting as manageable as possible.

The NAG Library is maintained in FORTRAN (F77 with some special in-house extensions for example to add data types not covered in the standard, such as double precision complex numbers) with specially marked up in-house documentation files in XML which capture much additional information about the detailed working of the routines – things such as argument constraints, output values and even some of the internal working, for example, when arrays (or sub-arrays) are used internally for workspace. The interface to functions is never changed; functions maybe deprecated, with suggestions for a newer alternative, but not altered or removed.

The FORTRAN library is thus “almost single source”. There are special DLLs for reporting errors via pop-up windows for embedding in Microsoft Windows Programs such as Excel or Word, but for most purposes the single set of FORTRAN code with its interfaces defined using the XML format suffices. This facilitates porting to different compilers, and there are currently over 60 supported platforms,. This approach also keeps the system documented for new developers to preserve the system as the original developers are no longer available.

There is also a C-release of the NAG library which is machine created from the FORTRAN master and the specially marked-up documentation, with an interface layer to the C library routines as a C wrapper to the translated FORTRAN. Other languages had been considered

since the NAG library was first issued in 1971 (Algol 60, Pascal and ADA for examples). Java support is provided through the JNI interface.

Testing is vitally important for a trusted library such as NAG, and correctness is the primary performance goal. There are published tests that users can run and also ever stricter tests suites (derived from the XML routine markup) used internally. In the case of test errors, approaches to handle these have included lowering compiler optimisation for that routine and correcting the source code. Thus when migration is undertaken, there is an emphasis on the "lots of testing" approach. With mathematical software, there is a dependency on the environment (e.g. floating point co-processors, floating point libraries) which can affect the accuracy of the result, in extreme results quite radically as small variations become amplified. This is complicated in the case of some mathematical functions which could have multiple "correct" answers to test cases, and the routine may generate a different one from the one generated in the previous environment.

6.2.2 Documenting the NAG Library

NAG provides extensive and detailed documentation on its library, which can be found online[51]. The library is divided functionally into a number of chapters organised into the area of mathematics covered according to the ACM modified SHARE classification index [11], although NAG also provide an indexing of their library according to the GAMS classification [12]. Each routine is then given a code based on its place the classification and its functionality.

Each function in the library is then given a detailed description. This includes the following:

- **Purpose**: textual statement of the function, including a mathematical notation.
- **Specification**: a formal description of the routines form, the number of parameters and their data types.
- **Description**: more detailed description of the operation of the function.
- **References**: References to the literature where a detailed description of the algorithm can be found.
- **Parameters**: A detailed description of the role of each of the parameters in turn, describing the values which they should contain on calling the routine (input) and on the routines completion (output). FORTRAN is state based, so they are state memory variables.
- **Error indicators and warnings**: description of any exceptional or erroneous behaviour
- **Accuracy**: a description of how to estimate the accuracy of the result – an important feature of evaluating the efficacy of a floating point routine to approximate a real-valued operation.
- **Further comment**: further notes on the behaviour of the routine.
- **Example**: an example problem, a fragment of sample code to demonstrate how parameter may be set and the routine called in a typical program, with the expected results on sample program data. This gives the user the opportunity to test the performance in a particular installation.

[51] For example, http://www.nag.co.uk/numeric/FL/FLdocumentation.asp documents the FORTRAN library.

Thus these functional descriptions should remain stable over time as the environment of the library and versions change. Thus for our purposes, these are an important source of significant properties.

6.2.3 Lessons for Software Preservation

From the experience of NAG we can draw a number of lessons for software preservation.

To keep a long-lasting system usable, preservation means migration in many circumstances, as it is simply not useful to keep the original environment. A working system is highly-sensitive to the computing environment and architecture. This migration may not even maintain source code – refactoring and rewriting the code enhance its preservability.

Preservability in this environment also becomes easier with good software engineering practice. Well designed and documented interface descriptions which are published and maintained over time a key feature of allowing good preservability and maintainability. This provides a stable interface for the user, which can rely on the significant behaviour remaining constant over time. It also gives a clear description of the system for the software maintenance team, as the team itself changes as well as when new target environments arise for migration, describing clearly how the system is intended to perform and how it fits together. NAG has developed a system with a single abstract XML based interface description, and as much as possible a single reference implementation in FORTRAN. By maintaining one code base and generating other versions from the number of variations in the system is controlled.

These functional descriptions also allow good documentation to be produced with functional descriptions, descriptions of interfaces, and code examples and test cases. This again enhances the stability for the user and constrains the maintenance task.

A final key observation is the vital importance of testing to ensure that the migration has been carried out successfully. This gives the validation of the preservation task undertaken in the migration. That is testing is required to ensure that the significant properties (functional characteristics and performance) are preserved accurately in the migration. Thus the extent of the validity of the preservation is constrained by the tests applied.

6.3 Astronomical Analysis Software – Starlink

6.3.1 Astronomical Software

Astronomical Software covers a wide range from supporting purely theoretical research (such as Cosmology) through observational data in different wavelength ranges to acquisition and control systems closely associated with an individual observatory or even a single instrument on a telescope. In observational astronomy the analysis techniques split naturally into wavelength ranges – with analysis of radio frequency observations being very different from the group of ultraviolet/optical/infrared data and both the previous types being very different again from X-Ray astronomy -where data is obtained from satellite missions.

Different organisations around the world have attempted to generate frameworks ("*environments")* to make it easier to write new software applications which would both interwork with existing applications within that particular software *suite* - which would often be targeted at the type of astronomy being performed.

Some examples of these *environments* are:

- AIPS [52]: Developed by NRAO (National Radio Astronomy Observatory[53]) in the USA. Written in FORTRAN, tailored towards radio-astronomy, freely available.

- AIPS++ [54]: A more recent product of NRAO, designed to replace AIPS (but never totally did!) written in C++, licensed against commercial use – potential users may need to agree to licence terms before use.

- Starlink: The software system described here. Applications written in FORTRAN, mainly used with optical/IR/UV data – but had a variant used for X-Ray data as well. Freely distributed.

- IRAF [55] : Developed by the National Optical Astronomical Observatory[56], USA. It is used for – and gets additional software support from – the Space Telescope Science Institute[57] (NASA – "Hubble Space Telescope"). Applications written in a pre-processed form of FORTRAN *(SPP*). Targeted at optical/UV/IR data.

- MIDAS [58]: Former analysis suite from the European Southern Observatory, Munich. Believed to be mothballed

The rest of this discussion concentrates on the Starlink Software Collection (*SSC*).

6.3.2 Introduction to Starlink

This section concentrates on some of the potentially *Significant Properties in the context of preservation* of the *Starlink Software Collection (SSC)* – a diverse suite of applications software, based around a common framework and used for astronomical data analysis.

The *Starlink Project*[59] was SRC/SERC/PPARC funded, ran for an unusually long duration between about 1980 until 2005 and had the mission of providing general *Astronomical Computing Support* to UK based astronomers at both universities and establishments (such as the Royal Observatory at Edinburgh) This support included computer hardware, site management effort, central purchasing etc. In addition it had a major software component - the *Starlink Software Collection (SSC)* - which was supported by a programming team based centrally at the Rutherford Appleton Laboratory and at some universities. While the *SSC* was

[52] http://www.aoc.nrao.edu/aips/
[53] http://www.nrao.edu/
[54] http://aips2.nrao.edu/docs/aips++.html
[55] http://iraf.noao.edu/
[56] http://www.noao.edu/
[57] http://www.stsci.edu
[58] http://www.eso.org/sci/data-processing/software/esomidas//doc/index.html
[59] http://www.starlink.ac.uk and http://en.wikipedia.org/wiki/Starlink_Project

installed at all Starlink controlled sites there were never any restrictions about local astronomers requesting other, often American, analysis software systems to be installed in parallel to the *SSC* on their Starlink provided hardware. The *SSC* was made available without charge and, while it was promoted widely at international conferences, it was mostly popular outside the UK at observatory sites with major UK participation.

Starlink was originally an exclusively VAX/VMS based project. Starting in about 1992 (after pilot trials) the increasing price/performance of Unix based systems led to a very major, but gradual (over about 3 years) adoption of a combination of Sun and DEC hardware (to minimise use of any propriety features while limiting support issues) and to a corresponding *port* of the *SSC* to these platforms. About 5-years later, after the emergence of *Linux* on inexpensive PC hardware, the *SSC* was further ported to this platform. The adoption of Linux also led to the distribution mechanisms being significantly modified to allow users to install the software themselves on personal PCs and laptops from CDs whilst selecting only desired applications and features.

The *SSC* was perhaps unusual in that, while it had a well defined *Environment (Infrastructure and support libraries – described later),* coding and documentation standards, and a large documentation collection[60] defining these in detail, it would also, in general, accept for distribution most *Applications Software* developed by astronomers which was thought to be generally useful – even if this software was not perfect according to these standards. In cases where this contributed software was used widely central effort was assigned to bring it up to a supportable level.

PPARC reduced the scope of the Starlink Project from around the turn of the millennium by eliminating the central support role for both site management and hardware. The central support of the *SSC* continued – and, indeed, efforts were made to modernise it by introducing *JAVA* applications and experimenting with *Web-service wrappers* to existing applications in the hope of interesting the new *Astrogrid Project*[61] to help fund it. Finally in 2005 the Project's central funding at RAL was curtailed. However the *Joint Astronomy Centre (JAC)* [62]– the support base for the two major UK telescopes in Hawaii - had become heavily dependent on the Starlink Software which used by them in a *pipeline-based workflow process* (entitled "ORAC") which they had developed. They successfully applied for limited PPARC funding to allow them to continue to maintain the Starlink Software until the end of their development cycle. To this date JAC still make periodic pre-built stable and tested versions of the SSC available[63], without guaranteed support, to a wider community. The work required to automate the build systems and testing such that now only the very limited residual effort at *JAC* can build such a complex system reliably will also be discussed.

6.3.3 Principle Features

We consider some of the principle features of Starlink and discuss their relevance to significant properties for preservation.

[60] http://www.starlink.ac.uk/Documentation/
[61] http://www.astrogrid.org
[62] http://jach.hawaii.edu
[63] http://starlink.jach.hawaii.edu/

Functionality

The first aspect to be considered is the *functionality* of the software. In the case of the Starlink *SSC* this could be categorised as the ***processing*** (for example to remove some known instrumental effect) or ***display*** (for example for editing or visualisation) of astronomical data. An important *property* of applications in such a *suite* is that software components should cooperate – for example an application to edit the data should output its results in a form which the succeeding display application could interpret. This lead to the concept of an *Environment within which the software exists* rather than the simple *API* commonly found with a single software library.

Software Environment – the *Composite Object*

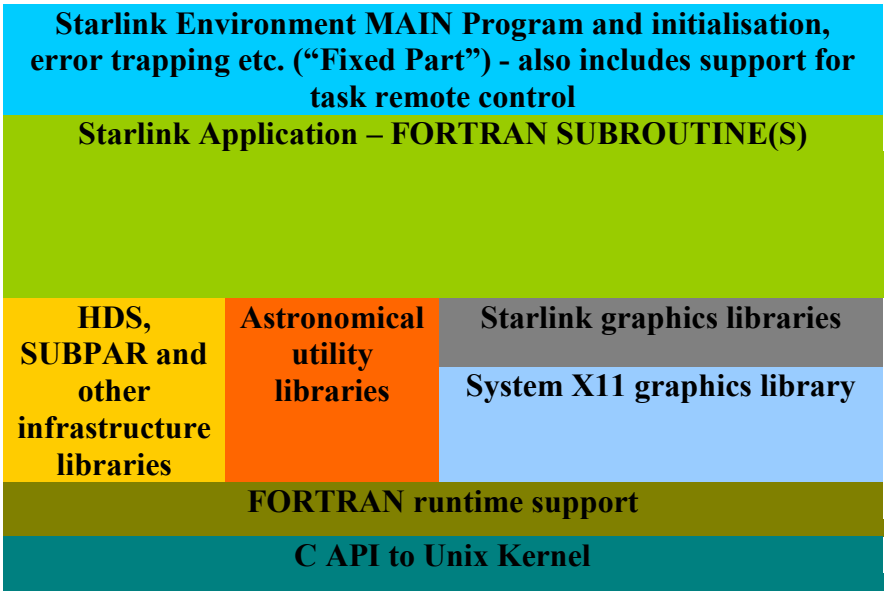

Figure 7: Starlink Software Architecture

In a system such as the Starlink *SSC* applications do not exist in isolation. They are written to conform to a complex external software system (the *Environment)* which can be thought of as a well-defined means of allowing a new application to *slot-in* to the existing suite. In practice this meant both adhering to well defined rules and using common subroutine library APIs to perform standard operations – some of these operations would be related to the *environment* while others would give *astronomical functionality*. The components of this environment are shown in Figure 7. Examples of the benefits of this approach for the Starlink *SSC* were:

- The ability to process and display data through a chain of applications written over time by many different astronomers or programmers. In fact, as part of the *SSC Infrastructure* was derived from an earlier *online* observatory system (called *ADAM)* it was possible to extend this manual control of disparate applications into the concept of a *data pipeline* at a higher level. Starlink provided a scripting language (*ICL)* to do this and this was the basis for the, much improved and extended, system developed at JAC, Hawaii which used extension libraries to the *Perl* language to control applications – both for data acquisition from the telescope and for further offline analysis by astronomers (*ORAC-OT* and *ORAC-DR*

respectively). For astronomers *ORAC-DR*[64] further abstracts their control of the overall analysis process into *recipes* which drive the pipeline processing at a more conceptual level.

- A common *look and feel.* As all applications were using common library APIs for input, output and data display.

- A single format for data files. All Starlink applications stored their results in a unique data system known as the *Hierarchical Data System (HDS)*[65]. HDS files were single files on-disk with a top-level *structure record* internally (with ASCII names for components) and then, at lower levels but still within the same file, further *structure or data records* – arranged in a hierarchal format. To further constrain application programmers inventing different names for the same *standard astronomical data type* HDS data was most often accessed via a higher-level access library API (*NDF – the n-Dimensional Data Format)* which imposed agreed data naming conventions.

The principal programming rules to make applications conformant to the *SSC Environment* were:

1. Applications would be written in FORTRAN **BUT** there could be **NO** *Main Program.* Applications were written as *Subroutines* – taking and return an *INTEGER* status value. These subroutines would be called by Environment code and could be grouped into composite systems (called *monoliths)* – a system developed to cope with slow executable loading on early VAX/VMS systems.

2. The *environment* would initially pass a *success* status value into the top-level subroutine. Thereafter all lower-level routines would both inherit and return this argument through their API. In addition, every subroutine should immediately return upon entry with a non-success status. Any failure within a called subroutine would be indicated by that routine setting this common *inherited status* to a non-zero failure code – as subsequent routines would inspect this before any action they would simply return. If not reset the failure error code would, as a last resort, be returned to the *environment* at the end of the top-level application subroutine and reported by it.

3. There could be no direct use of FORTRAN input or output to the user. Prompting for user input would be done via *Parameter System API* calls – this gave a uniform appearance to prompts for user input but also provided additional functionality – such as input value persistence, defaulting, limit checking etc.

   Output to the user was similarly performed using either the *Message System* (for normal progress reports) or the *Error System* (which could interpret and report on the meaning of the *inherited status error code)* APIs. A secondary feature of this error system was its ability to *stack* potential errors and to *annul* these if a possible error condition could be recovered.

4. While application data could be read from and written to any form of external data files any data designed to be used subsequently by other applications **within** the *environment* was to

[64] http://www.starlink.rl.ac.uk/cgi-bin/htxserver/sun230.htx/sun230.html
[65] http://www.starlink.rl.ac.uk/cgi-bin/htxserver/sun92.htx/sun92.html

be stored (using its API calls) within the *Hierarchical Data System (HDS).* This HDS data system was a proprietary record based single binary file on disk.

5. Graphical display capabilities would be performed by dedicated libraries based mainly on the GKS 2-dimensional graphics API. Use of graphics through APIs at this level gave, along with the advantages of use of new devices when a device driver was added to the GKS system, additional functionality such as storing and recalling the layout of multiple plots on the same surface.

Building and testing the *SSC*

Obviously in a complex system such as the Starlink SSC a successful build and deployment of the *Core Infrastructure* (*environment* and *astronomical utility* libraries*),* the applications and the numerous supporting data and configuration files to exact locations was a continual challenge. The interdependencies were very complex and obtaining the correct build order for a given *component* or *application* was never simple. Another challenge was making the software updates simple and reliable for the *site mangers* – particularly when early slower network links made complete updates totally impractical except at infrequent intervals. A later challenge for *Linux* was making simple and, optionally, partial installations from CD easy to do on a self-service basis.

Another issue was that, while the *SSC* only migrated once between dissimilar operating systems (VAX/VMS to Unix) the various Unix systems it was supported on were not totally identical to each other at the fine-grain level.

To automate the building of such a large software system – given limited effort (typically, only a single *Software Librarian*) and support on a number of hardware platforms, a number of different approaches were explored during the life of this Project. All were designed to capture or include the build dependencies and to be as automatic as possible.

In rough chronological order the following techniques were employed and refined.

- Build scripts for VAX/VMS.

- Makefiles were introduced for Unix but needed slight adjustments for the supported Unix variants (SunOS, OSF1/DEC Unix and, later Linux). Pre and post-processing scripts were added to cope with these differences.

- On early slow networks only partial incremental releases were feasible. There was continual debate over the merits of these as opposed to less frequent complete releases.

- Finally, as the Project came close to losing its permanent central support team, considerable work was put into making the build process compatible with the GNU *autoconf* system[66]. Specialised macros had to be added to this system to cope with the FORTRAN code in the *SSC* but the end result was that it was possible to build anything between a single application (together with *just* those parts of the *infrastructure* and other *utility libraries* that it needed) through to the entire *SSC* for a particular hardware platform. The strength of

[66] http://www.gnu.org/software/autoconf

this approach was demonstrated by the fairly straightforward port of the entire *SSC* to Macintosh hardware using appropriate *autoconf* macros.

- The source code was placed into *cvs*[67] [68] and, later, *svn*[69] [70] repositories, and an automated nightly build system was created. Any regression caused by the *check-in* of faulty code should become apparent almost immediately.

- The principal benefit of **BOTH cvs** and svn was that the SSC was now, for the first time in its history, truly *single source* – rather than a number of slightly different versions being needed for various architectures (leading to the possibility of regression if not kept in step).

Versions

Software typically goes through many versions, as errors are corrected, functionality changed, and the environment (hardware, operating system, software libraries) evolves. Earlier versions may need to be recalled to reproduce particular behaviour.

The Starlink *SSC* was an unusually long lasting Software System and some history of its evolution process may be instructive and relevant to other such systems. At every stage there was very limited effort to make changes so, while decisions were not being made explicitly in the context of preservation, they were being considered in the, possibly related, contexts of both simplifying the build process (reducing the complexity of the inter-dependencies, automating builds and testing further etc.) or in terms of ease of future maintenance. As well a number of obvious changes (such as those caused by new hardware) there were also changes driven by outside technologies (for example the rapidly increasingly large pixel counts of the CCD detectors being currently installed on telescopes - relative to the norms of just a few years ago)

Operating system and hardware changes

Starlink started in 1982 with a network of 6 DEC VAX 11/780 computers connected by (slow) DECNET communications links.

Properties of such a system related to the build and maintenance of the *SSC* might include

- A single platform of a vendor supported operating system and compilers with excellent documentation

- A common executable format – only complied binaries needed to be sent to remote sites Updates to the *SSC* could be built and tested at RAL then, when necessary, one or more updates to various applications or other core components would be assembled into a single VMS *BACKUP* format package – together with a command procedure for installation - the site managers would then be instructed to collect this.

[67] http://en.wikipedia.org/wiki/Concurrent_Versions_System
[68] http://www.nongnu.org/cvs/
[69] http://en.wikipedia.org/wiki/Subversion_%28software%29
[70] http://subversion.tigris.org/

- The use of VAX/VMS *sharable libraries* meant that infrastructure libraries could be updated without a requirement to install new versions of all the applications using these libraries.

Later, from about 1990, the price/performance benefits of alternative hardware could no longer be ignored. These alternative hardware platforms all ran some variant of the *Unix* [71] operating system. The decision to gradually change to this platform led to the most significant change in the Starlink *SSC* – the *Port to Unix*

- Two versions of Unix on different hardware (DEC/OSF and Sun/SunOS) were deliberately targeted to avoid any accidental *lock-in* to specific vendor features. Both had FORTRAN compilers which were highly compatible with VAX/VMS extensions...

- Sharable libraries (as had been used on VAX/VMS) were supported on SunOS – but not on DEC/OSF – thus on this Unix variant a change to a low level library **DID** require a rebuilt and re-installation of all applications.

- VAX/VMS FORTRAN extensions had allowed API calls through to the full functionality of the underlying operating system and use of this feature had enabled much of even the lowest level *Environment* related code to be written in this language. This was replaced by C-language code (with a FORTRAN callable API) which was developed over time to give as exactly as possible the same features. This C-code had to, in time, also emulate some intrinsic VAX/VMS features (such as command line recall). The *user experience* on Unix was thus, initially, significantly poorer – and expectations had to be managed!

Later, around 1995, the emergence of the popular *public domain Linux*[72] *Unix variant* was again a *Significant External Event* which the Project was forced to react to. It appeared at the time that this gave the prospect of even lower cost hardware and also, for the first time, the potential for astronomers to run the *SSC* on their home system or their laptop.

Changes to the SSC caused by Linux included

- For the first time compilation of the entire FORTRAN applications suite was attempted by *public-domain* compilers (initially GNU f2c and then GNU g77[73]). Differences in such things as FORTRAN *common block naming,* mechanisms for calling C-language routines and even such, apparently simple, things as undeclared variables (which were no longer being automatically initialised to zero) led to significant porting effort.

- As astronomers were now expected to be able to install the *SSC* by themselves parallel changes to the distribution systems – especially the creation of CDs with simplified automatic installation scripts were required.

The final port of the Starlink SSC was much more recent when the decision by Apple to replace the heart of MacOS by a Unix variant (Darwin) increased enormously the popularity of MacOS laptops with astronomers. This change coincided with the change of the SSC build system to using GNU *autoconfigure* - and, indeed, was its first major success.

[71] http://en.wikipedia.org/wiki/Unix
[72] http://en.wikipedia.org/wiki/Linux
[73] http://www.gnu.org/software/fortran/fortran.html

External technology events

As an example of the Starlink *SSC* having to respond to external technology changes the example of the Hierarchical Data System (*HDS)* will be considered. This is a crucial subsystem within the *SSC* which facilitates applications written for the Starlink *Environment* in cooperating in the storage and shared processing of data.

HDS is a file-based hierarchical data system designed for the storage of a wide variety of information and is particularly suited to the storage of large multi-dimensional arrays together with their ancillary data. HDS organises data into hierarchies, broadly comparable with a hierarchical file system, but contained within a single *HDS Container File.* The structures stored in these file are self-describing and flexible; HDS supports modification and extension of structures previously created, as well as deletion, copying, renaming etc. All information stored in HDS files is portable between the machines HDS is implemented upon – the implementation transparently takes care of format and endian conversion problems when data is accessed via the API.

HDS is probably the single most important component underpinning the SSC. It is also the most complex and, to further increase the maintenance challenge, the original programming expertise which created HDS had all long-since left the Project! Less than perfect limited design documentation remains – together with the source code (with limited comments!)

When HDS was designed in the early 1990s one of its key record size pointers was dimensioned at 20-bits – a consequence of defining the API for the HDS routines to use standard INTEGER FORTRAN values. This restriction led, internally, to a size limit for a single structure (but not the file itself) of an HDS file of ~500Mbyes. From around the year 2000 onwards it was clear that current and future enhancements of telescope detector sizes made this limit untenable and, unless it could be removed, it would effectively curtail the life of the entire *SSC*. The *data-reduction pipeline* system (*ORAC)* used on the UK Telescopes in Hawaii was used in data acquisition from telescope and was in the *front-line* for this problem.

HDS was, with some difficulty (and considerable effort!), modified to use *64-bit pointers* internally while, at the same time, allowing it to continue to read and write all of the vast number of existing data files. Any typical application will have several concurrent HDS files open (as subsystems such as the *Parameter System* also use these) and, as the HDS code is single-threaded, the library has to adapt to its use of each file.

Look and feel

While, for many complex software systems, the exact look-and–feel may be an important aspect of its long term preservation, this is perhaps less true of the Starlink *SSC*. Astronomers generally use reasonably simple 2-dimensional plots or 2-d false colour imaging of astronomical objects.

An underlying property of the various Starlink graphics libraries were, however, their reliance on its various lower-level graphics systems (including GKS, IDI and PGPLOT). It was important to continue to develop device-drivers for these systems to keep them in step with those sorts of displays and other output devices that astronomers wished to use. While X11 output on workstations became dominant this was not always the case. Even then the detailed properties of X11 displays could have unexpected impacts. Early 8-bit graphics cards were

typically used by their X11 device driver in *PseudoColor* mode - which gave the side-effect of easy manipulation of the colour lookup-table for dynamic image visualisation. Later, higher performance, graphics cards lost this simple ability and applications had to be recoded to redraw images in order to modify the colour representation to the user.

Software architecture

The architecture of the Starlink *SSC* depends crucially on the continuing existence of components such as compilers for the FORTRAN and C languages. In addition it is currently a Unix-based software system and relies heavily on X11 graphics.

Licensing

In common with the other Astronomical *Environments* describe earlier (AIPS, MIDAS, IRAF etc.) licensing has never been a major issue for the Starlink SSC. Rather belatedly a license was developed to attempt to limit any commercial use of the software but, apart from that, CDs were being distributed at meetings wherever possible and pre-build systems were downloadable from the Web.

7 Conceptual Framework

In order to express the significant properties of software, we need to develop a conceptual framework to capture the approach taken to software preservation and the structuring of the software artefact and the significant properties of software for preservation.

7.1 Software Preservation Approach

Various approaches to digital preservation have been proposed and implemented, usually as applied to data and documents, but they do usually apply to the means of preserving the underlying *software* used to process or render the data or document. Thus these preservation approaches directly relate to the preservation of software.

The Cedars Guide to Digital Preservation Strategies [3] defines three main strategies, which we give here, and consider how they are applicable to software.

- **Technical Preservation. (techno-centric).** Maintaining the original software (binary), and sometimes hardware, of the original operating environment. Thus this is similar to the use case for software preservation arising from the museums and archives where the original computing hardware is also preserved and as much of the original situation is maintained as is possible. This is also an approach which is taken in many legacy situations; otherwise obsolete hardware is maintained to keep vital software in operation.

- **Emulation (data-centric).** Re-creating the original operating environment by programming future platforms and operating systems to emulate the original operating environment, so that software can be preserved in binary and run "as is". This is a common approach, undertaken in for example the PLANETS project, and also by groups such as the Software Preservation Society.

- **Migration (process-centric).** Transferring digital information to new platforms before the earlier one becomes obsolete. As applied to software, this means recompiling and reconfiguring the software source code to generate new binaries, apply to a new software environment, with updated operating system languages, libraries etc.

  In practice, software migration is a continuum. The minimal change scenario is that the source code is recompiled and rebuilt unchanged from the original source. However in practice, the configuration scripts, or the code itself may require updating to accommodate differences in build systems, system libraries, or programming language (compiler) version. An extreme version of migration may involve rewriting the original code from the specification, possibly in a different language. However, there is not necessarily an exact correlation between the extent of the change and the

  Software migration (or “porting” or “adaptive maintenance”) is in practice how software which is supported over a long period of time is preserved. Long term projects such as StarLink, or software houses such as NAG spend much of their effort maintaining (or improving) the functionality of their system in the face of environment change.

These three approaches have their advantages and disadvantages, which have been debated in the preservation literature.

Technical (hardware) preservation has the minimal level of intervention and minimal deviation from the original properties of the software. However, in the long-term this approach is difficult to sustain as the expertise and spare components for the hardware become harder to obtain.

The emulation approach for preserving application software is widespread, and is particularly suited to those situations where the properties of the original software are required to be preserved as exactly as possible. For example, in document rendering where the exact pagination and fonts are required to reproduce the original appearance of the document; or in games software where the graphics, user controls and performance (it should not perform too quickly for a human player on more up to date hardware) are required to be replicated. Emulation is also an important approach when the source code is not available, either having been lost or not available through licensing or commercial restriction. However, a problem of emulation is that it transfers the problem to the (hopefully lesser) one of preserving the emulator. As the platform the emulator is designed for becomes obsolete, the emulator has to be rebuilt or emulated on another emulator. Thus a potentially growing stack of emulation software is required.

The migration approach does not seek to preserve all the properties of the original, or at least not exactly, but as observed in the CASPAR project, only those up to the API – which we could perhaps generalise to those properties which have been identified as being of significant for the preservation task in hand. Migration then can take the original source and adapt to the best performance and capabilities of the modern environment, while still preserving the significant functionality required. This is thus perhaps the most suited where the exact (in some respects) characteristics of the original are not required – there may be for example difference in user interaction or processing performance, or even software architecture – but core functionality is maintained. For example, for most scientific software the accurate processing of the original data is of key importance, but there is a tolerance to change of other characteristics.

These three different preservation strategies thus require different levels of detail of significant property for the software artefacts. In this report, we are neutral to the preservation approach, but consider how the preservation of the key properties can be identified and checked.

7.2 Performance Model and Adequacy

Closely related to the preservation approach is the notion of how sufficient level of *performance* to adequately preserve required characteristics of software. Performance as a model for the preservation of digital objects was defined by the National Archives of Australia in [6] to measure the effectiveness of a digital preservation strategy. Noting that for digital content, technology (e.g. media, hardware, software) has to be applied to data to render it intelligible to a user, they define a model as in Figure 8. Here *Source data* has a *Process* applied to it, in the case of digital data some application of hardware and software, to generate a *Performance*, where meaning is extracted by a user. Different processes applied to a source may produce different performances. However, it is the properties of the *performance* which need to be taken into account when the value of a preservation action. Thus the properties can

arise from a combination of the properties of the source with the technology applied in the processing.

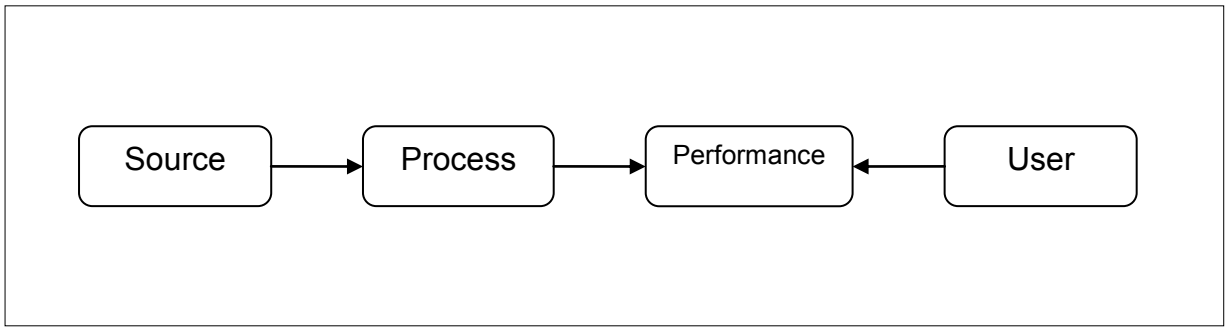

Figure 8: NAA Performance Model

The notion of performance has been developed in the context of traditional archival records, and has been adopted in other studies into the significant properties of different media types (see [5], [2]), which considers comparing the performance created by the original process of rendering with that created by a later rendering processes on new hardware and software. The question which arises is how this model applies to software.

In the case of software, the performance is the execution of software binary files on some hardware platform configured in some architecture to provide the end experience for the user. However, the process stage depends on the nature of the source used. This is illustrated in Figure 9.

- In the case where binary is preserved, the process to generate the performance is one of preserving the original operating software environment and possibly the hardware too, or else emulating that software environment on a new platform. In this case, the emphasis is usually on performing as closely as possible to the original system.

- When source code and configuration and build scripts are preserved, then a rebuild process can be undertaken, using later compilers and linkers on new a new platform, with new versions of libraries and operating systems. In this case, we would expect that the performance would not necessarily preserve all the properties of the original (e.g. systems performance, or exact look and feel of the user interface), but have some deviations from the original.

- In an extreme case, only the specification of the software may be preserved. In this case, a performance could be replicated by recoding the original specification. In this case, we would expect significant deviation from the original and perhaps only core functionality to be preserved. This case would seem to be exceptional, however, it is less unusual in coding practice, as packages are often migrated into a different language; for example the NAG library originated in FORTRAN, but later produced a C version. In some circumstances, this is a result of *reverse engineering* where source code (or even in extreme cases binary code) is analysed to determine its function and recoded.

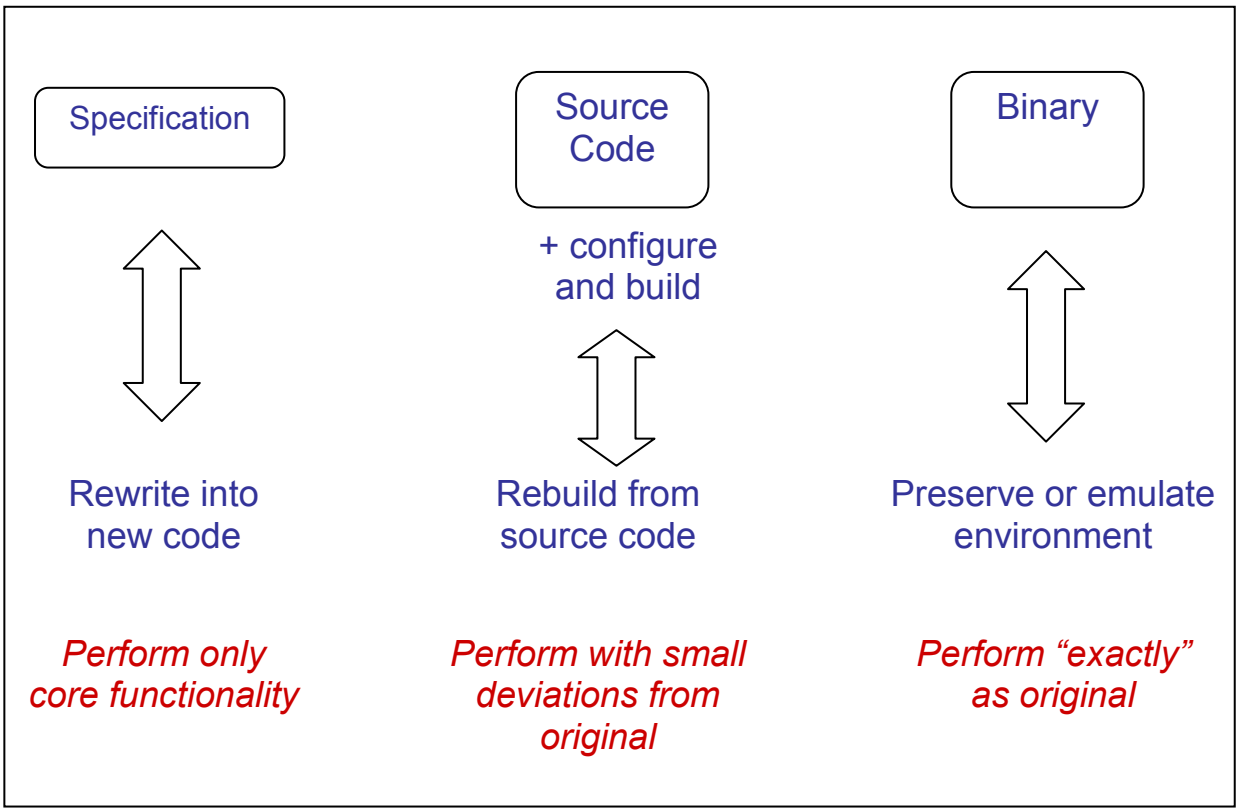

Figure 9: Software Performance models from different sources

Software performance can thus result in some properties being preserved, and others deviating from the original or even being disregarded altogether. Thus in order to determine the value of a particular performance, in addition to the established notion of ***Authenticity*** of preservation (i.e. that the digital object can be identified and assured to be the object as originally archived) we define an additional notion of ***Adequacy***.

> *A software package (or indeed any digital object) can be said to perform adequately relative to a particular set of significant properties, if in a particular performance (that is after it has been subjected to a particular process) it preserves that set of significant properties to an acceptable tolerance.*

By measuring the adequacy of the performance, we can thus determine how well the software has been preserved and replayed.

7.2.1 Performance of software and data

A further refinement of the performance model for software is that the measure of adequacy of the software is closely tied to the performance of the input *data*. The purpose of software is (usually) to process data, so the performance of a software package is *processing* of its input data. This relationship is illustrated in Figure 10.

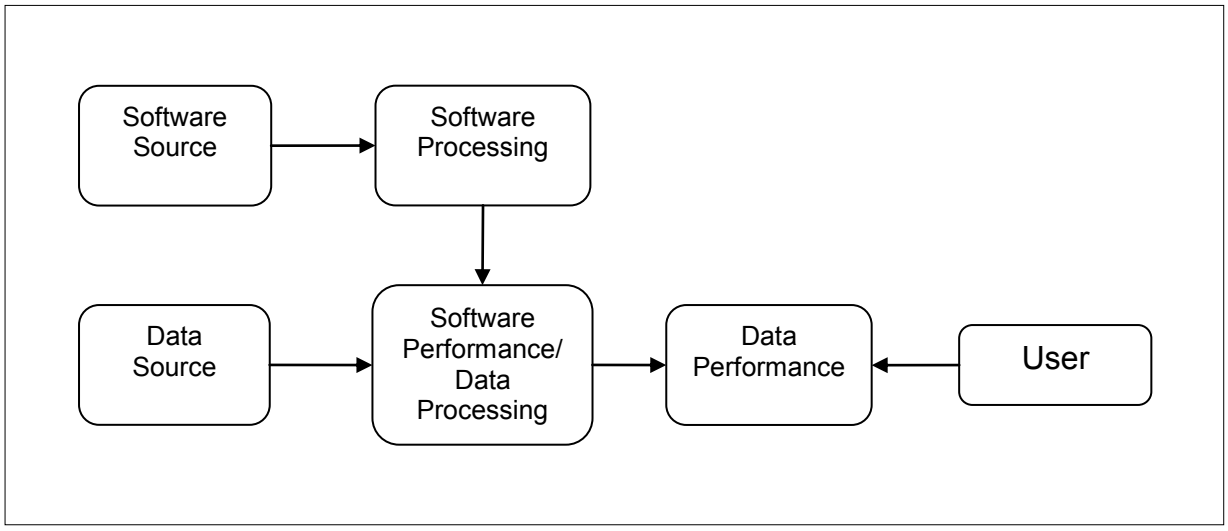

Figure 10: Performance models of software its input data

So for example, in the case of for example, a word processing package which is preserved in a binary format, which is processed via operating system emulation, the performance of the package is the processing and rendering of word processing file format data into a performance which a (human) user can experience via reading it off a display. Thus for many functional properties of software, the measure of adequacy of the software is the measure of the adequacy of the performance when it is used to process input data and how well it preserves the significant properties of its input data.

This can be applied recursively to software which processes other software, for example software used for emulation or compilers to build software binaries, which also needs to be preserved. In this case, the performance of software is the processing of the application binaries or source code, which in turn is measured by its adequacy in processing its intended input data.

Thus different preservation approaches require different significant properties, and the adequacy of the preservation is dependent upon the performance of the end result on the end use on *data*. The adequacy can be established by performing trial executions against test data. Thus, the adequacy of preservation of a particular significant property can be established by testing against pre-arranged suites of test cases with the expected behaviour.

7.3 A Conceptual Model for Software

As in the InSPECT work on developing a framework for significant properties for digital objects in general [5], we recognise that a conceptual data model is required to capture the digital object (i.e. software) under consideration. This data model will guide us on the level of granularity at which significant properties can be identified, and provide an understanding of the relationship between digital objects, thus giving traction on handling the complexity of the objects, a particularly important aspect in handling software. InSPECT considered a number of conceptual models which have been proposed for digital objects, including FRBR [13], PREMIS [14] and the National Archives data model, and based on these develop a model for associating significant properties at different levels of granularity.

We propose to develop a similar conceptual data model for software. However, there are a number of factors which need to be taken into account for developing a model for software.

- **Software is a composite object.** Typically software is composed of several items. Normally these would include binary files, source code files, installation scripts, usage documentation, and user manuals and tutorials. A more complete record may include requirements and design documentation, in a variety of software engineering notations (for example, UML), test cases and harnesses, prototypes, even in some cases, formal proofs. These items each have their own significant properties, some of which are the properties of their own digital object type, e.g. of documents or of data for test data. The relationships between these items need to be maintained.

- **Versioning.** Software typically goes through many versions, as errors are corrected, functionality changed, and the environment (hardware, operating system, software libraries) evolves. Earlier versions may need to be recalled to reproduce particular behaviour. Again the complex relationships need to be maintained.

- **Adaptation to operating environment.** Each version itself may be provided for a number of different platforms, operating systems and wider environments. In extreme cases, there may be different variants provided for specific machines (this was particularly the case in the past, and still applies when codes are tailored for high-performance systems where the performance is sensitive to the specific architecture of the target machine). Thus each version, while having essentially the same code base, may have variations, which may also vary in functional characteristics as different environments provide different features.

We provide a general model of software digital objects, which has a parallel with the FRBR model. We will go on to relate each concept in the model with a set of significant properties.

7.3.1 The Software System

We define a four layer model for software, given schematically and with its correspondence to the major entities of the FRBR model in Figure 11.

This model has four major conceptual entities, which together describe a complete ***Software System***. These are *Package, Version, Variant* and *Download.* This is in analogy with the FRBR model. The four levels roughly correspond to Work / Expression / Manifestation / Item, although we would warn against taking this analogy too far.

We consider each of these in turn, noting the types of significant properties we would typically associate with each level. Note at this level we also do not distinguish between source code, binaries and other supporting digital objects; these are considered below as the components of the software system, which is discussed in a later section.

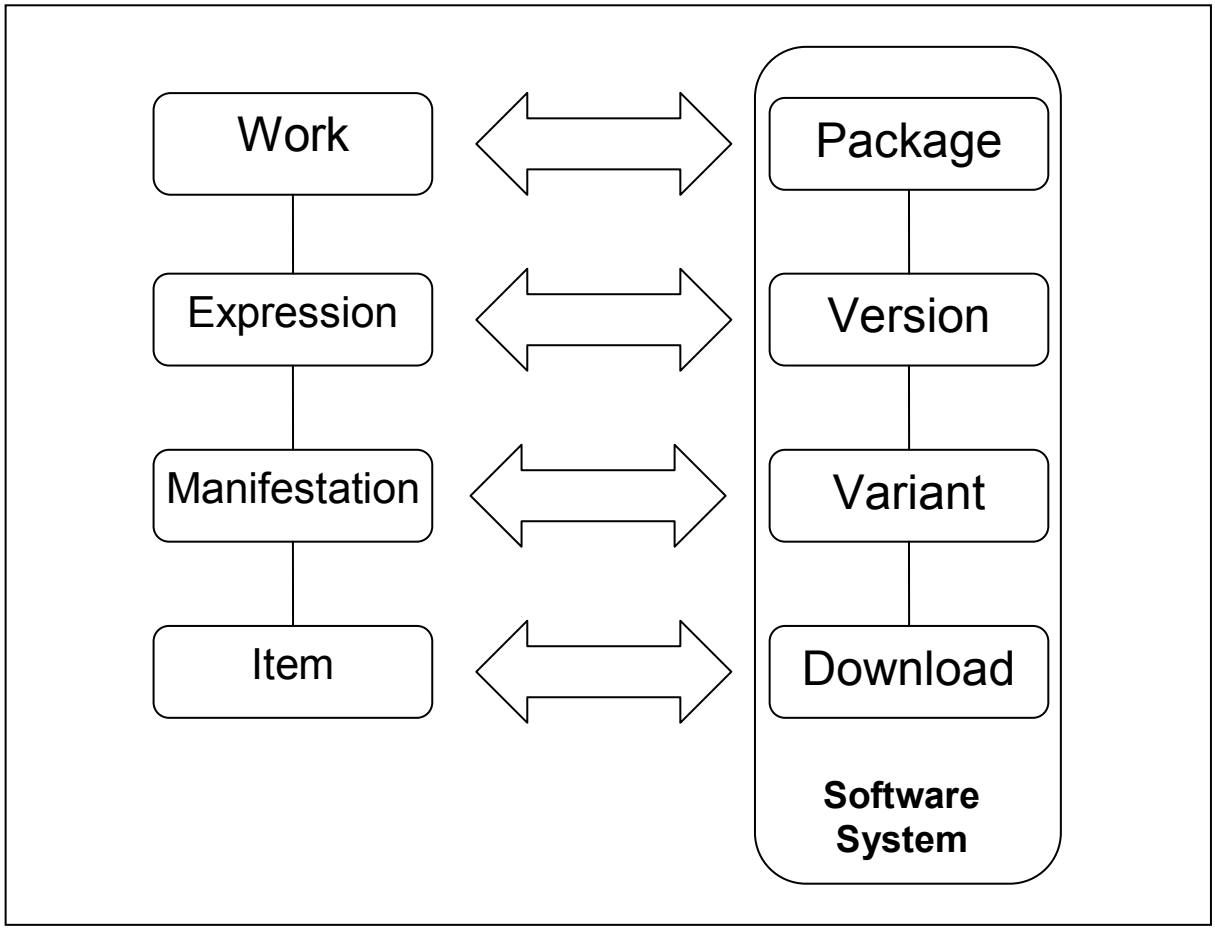

Figure 11: Conceptual model for Software and relationship to FRBR

Package. The package[74] is the whole top-level conceptual entity of the system, and is how the system may be commonly or informally referred to. Packages can vary in size and could range from a single library function (e.g. a function in the NAG library), to a very large system which has multiple sub-packages, with independent provenances (e.g. Linux). Thus examples would be "Windows", "Word", "Starlink", "Xerces". Packages themselves can be composite objects (a software library or framework) and may have a number of other sub-packages within them.

Packages are characterised by the following main features.

- A package has a single functional purpose, the overall gross goal of the software for example
  - "Word Processor" for Word;
  - "Framework to support astronomical software" for StarLink;
  - "Gram–Schmidt orthogonalisation of n vectors of order m" for function identifier F05AAF in the NAG library. This function can be regarded as a package in its own right, but it also a sub-package of the whole NAG library (which is also a package).
- A package has an "owner" responsible for developing, distributing and supporting the software and also having rights to control the usage of the software, although not always of the sub-packages within the package. Software often changes ownership, but then it should

[74] Not to be confused with the notion of information package as used in OAIS.

change as a package - the function and the way the function is delivered is likely to change as well as the authorising body. Typically, there might be a software licence associated with a software package as a whole. However, licences may also vary according to the version of the software, so we also allow the possibility of assigning licences to particular version. Software owners are not always straightforward to establish, particularly in the case of open-source software, although primary individuals responsible for the developing and maintaining a particularly coherent code-base can usually be identified. Thus:
    - Word is owned by Microsoft (www.microsoft.com)
    - Apache is owned by the Apache Software Foundation (www.apache.org)
- A coherent history and provenance associated with its responsible authority.
- Overall conceptual architecture of the system. This is likely to be stable for the whole package, though for long-lasting software, a major refactor of the software may result in different conceptual software architecture, as in the case of StarLink. In those cases, it may be considered as a new (but related) package entirely, although maintaining many of the same components and sub-packages.

Version. A version of a software package is expression of the package which provides a single coherent presentation of the package with a well defined functionality and behaviour and usually in environmental features. Differences in versions are characterised by changes to its functionality and also potentially performance. Typically for publically available software, versions are associated with the notion of a software release, which is a version which is made publically available, but in a development system, there are likely to be other versions in the system. Versions are also captured in version control systems such as CVS and Subversion by the branches of the development. Release branches represent snapshots over time of the development, and can reflect the relationships between the various releases.

Note also that in composite packages, the sub-packages will themselves have a number of versions which will be related to versions of the complete package. These releases will not necessarily be synchronised, so the relationship will need to be captured.

The properties which characterise the difference between versions would include:

- Changes in detailed functionality, e.g. presence of commenting in Word, coverage of XML standard versions in Xerces.
- Corrections to previous version's buggy behaviour.
- Changes in behaviour in error conditions.
- Changes to user interaction.

Variant. Versions may have a number of different variations to accommodate a number of different operating environments, thus we define a *Variant* of the package to be a manifestation of the system which changes in the *software operating environment*, for example target hardware platform, target operating system, library, programming language version. In this case, the functionality of the version is maintained as much as is practical; however, due to different behaviour supported by different platforms, there may be variations in behaviour, in error conditions and user interaction (e.g. the look and feel of a graphical user interface).

The properties which characterise the difference between variants would include:

- Changes in operating environment, including hardware platform, operating system and programming language version, auxiliary libraries, and peripheral devices.

- Changes in functional behaviour as a result of change in software environment.
- Different operating performance characteristics (e.g. speed of execution, memory usage).

In practice, Version and Variant may be very difficult to distinguish: changes in environment are likely to change the functionality; new versions of software are brought out to cope with new environments. It may be arguable in some circumstances that Versions are subordinate to Variants, and in others we may wish to omit one of these stages (software which is only ever targeted at one platform). But it is worth distinguishing the two levels here, as it makes a distinction between adaptations of the system largely to accommodate change in *functional properties (versions),* with those which are largely to accommodate change in *properties of the operating environment.*

Download. An actual physical instance of a software package which is to be found on a particular machine is known as a *Download.* It may be also referred to as an installation, although there is no necessity for the package to be installed; a master copy of stored at a repository under a source-code management system may well not be executable within its own environment.

The properties which characterise the difference between variants would include:

- Ownership – that is the user of the software (licensee), rather than the owner of rights in the system (the licensor).
- An individual licence tailored the use of the particular download and user.
- A particular MAC or IP address, URLs etc identifying particular locations or machines.
- Usage of particular hardware and peripheral devices as appropriate.

7.3.2 Software Components

All of the entities in the above conceptual model of software which form a software system are *composite*. Some of them may be subsystems, with sub-packages. All systems however, will be constructed out of many individual *components*[75]. A component is a storable unit of software which when aggregated and processed appropriately, forms the software system as a whole. Components can thus represent the following software artefacts:

- either, a part of its code base; or
- an executable machine readable binary; or
- a configuration or installation file capturing dependencies; or
- documentation and other ancillary material which while not forming a direct part of the machine execution process, nevertheless forms an important part of the whole system so that it is (re-)usable.

Components typically (but not necessarily always) roughly corresponds with a *file* (a unit of storage on an operating system's memory management system). However, multiple

[75] Note that this use of the term *Component* contrasts with the use of the term in the InSPECT project, given in [4]. Component entities in [4] are described as "the method in which manifestations are stored physically", and thus correspond more closely to *Downloads* in our model. We make the distinction to handle the inherently composite nature of software.

components can be stored within in one file (e.g. a number of subroutines within one file) or across a number of files (e.g. help system or tutorial stored within a number of HTML files).

Components may also be formed of a number of different digital objects, (e.g. text files, diagrams, sample data) which themselves would have significant properties associated with their data format. A comprehensive preservation strategy for the full software system would have to consider those significant properties as well, but we do not consider these significant properties further in this report, but refer to the literature on the significant properties of those digital objects as appropriate.

Software components are thus associated with a package, version or variant in the conceptual model of software as in Figure 12.

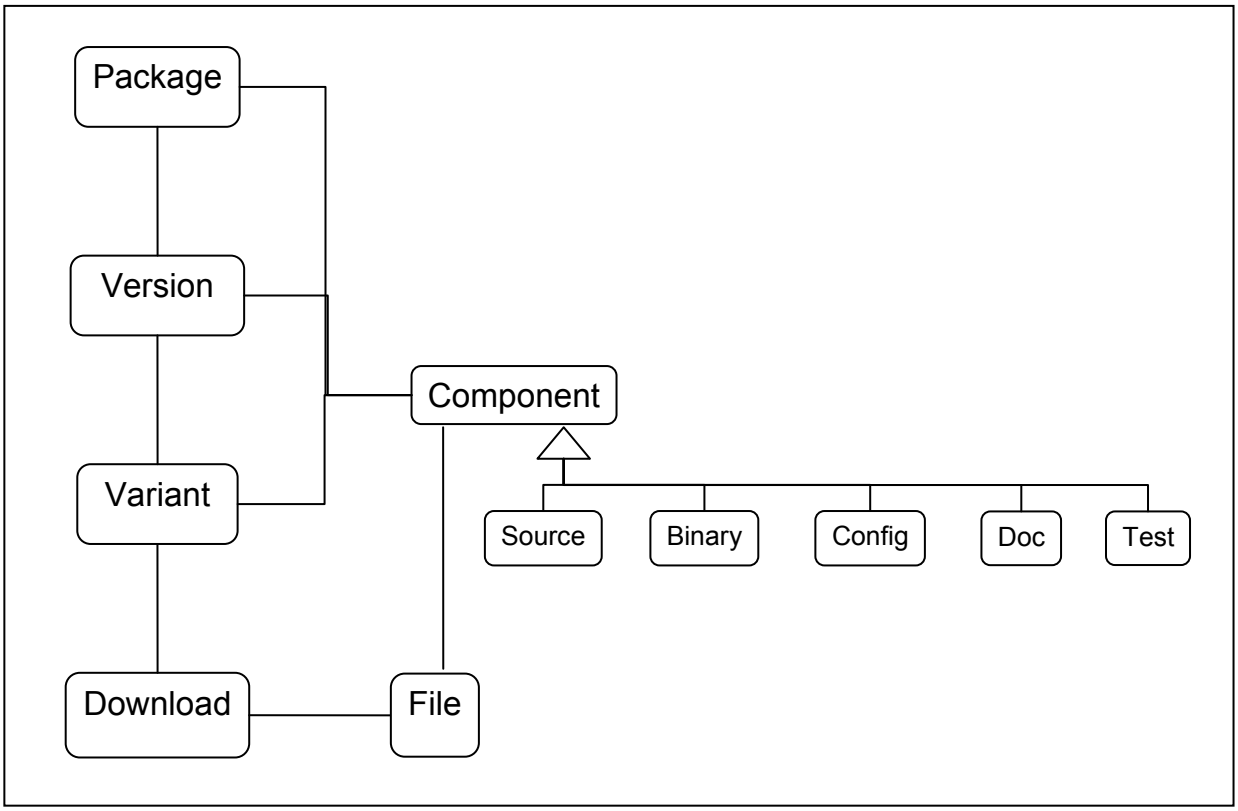

Figure 12: The Software Component Conceptual Model

In this model, we give a number of different kinds of software component. Note that this list is not exhaustive, and additional kinds of component may be identified. We give here the most common.

- **Source.** A unit of formal code written in human readable and machine processable programming language. Source code would normally need to be compiled into machine readable code, or else interpreted via an interpreter in order to execute. Source code components come under a variety of different names in different programming languages, such as “module”, “method”, “subroutine”, “class” or “function”. Theoretically, we could break down source components into individual statements or instructions; however, we do not consider that level of detail as essential to capture significant properties.

- **Binary.** An software artefact in machine processable code, not usually human readable, which is either directly executable on some target operating environment, or else executable by some virtual machine (e.g. a Java Virtual Machine). Binaries are usually standalone, or may require to be linked to dynamically linkable library binaries to execute.
- **Configuration.** A component which describes the configuration of the components to generate a working version of the code and captures dependencies between components. Three notable types would include: Build scripts, which capture the dependencies between source code to build an executable; Installation scripts, which control the installation of a package, including setting environmental dependencies and variables; Configuration scripts which set a number of environment specific variables.
- **Documentation.** Human readable text-based artefacts which do not form part of the execution process of the system, but provide supplementary information on the software. There are a number of different documents which may be typically associated with software, of which we distinguish: Requirements definitions; Specifications; User Guides (manual, tutorials); Installation Guides; Version Notes; Error Lists; Licences.
- **Test Suite.** Representative examples of operation of the package and expected behaviour arising from operation of the package. Produced to test the conformance of the package to expected behaviour in a particular installation environment.

Components have dependencies between them, which is often captured in the configuration files. For preservation, we may not need to explicitly model the dependencies, but need to be aware that they are captured and maintained. Significant properties can also be associated with components as well as on the package/version/variant and as noted the significant properties of a component may be of a different digital object type.

8 Significant Properties of Software

8.1 Categories of Significant Properties

Significant properties are defined as "*those characteristics of digital objects that must be preserved over time in order to ensure the continued accessibility, usability, and meaning of the objects*" [15]. In considering what significant properties would apply to software, we need consider the following seven general categories of features which characterise software.

- **Functionality** Software is typically characterised by what it does. This may be in terms of its input and outputs, a description of its operation and algorithm, or a more semantic-based description of its functionality in terms of the domain it addresses. All these levels may be significant and should be considered for preservation.

- **Software Composition.** Typically software is composed of several components. Normally these would include binary files, source code modules and subroutines, installation scripts, usage documentation, and user manuals and tutorials. A more complete record may include requirements and design documentation, in a variety of software engineering notations (for example UML), test cases and harnesses, prototypes, even in some cases, formal proofs. These items each have their own significant properties, some of which are the properties of their own digital object type, e.g. of documents or of data for test data. The relationships between these items need to be maintained. Further, software typically goes through many versions, as errors are corrected, functionality changed, and the environment (hardware, operating system, software libraries) evolves. Earlier versions may need to be recalled to reproduce particular behaviour. Again the complex relationships need to be maintained.

- **Provenance and Ownership.** The provenance and ownership of the software should be recorded. Different software components have different and complex licensing conditions. In order to maintain the usability of software, these need to be considered in the preservation planning.

- **User Interaction.** If complete applications are preserved, there is also the question of the human-computer interaction, including the inputs which a user enters through a keyboard, pointing device or other input devices, such as web cameras or speech devices, and the outputs to screens, plotters, sound processors or other output devices. The Look and Feel and the model of user interaction can play a significant factor in the usability of the software and therefore should be considered amongst its significant properties. If using a tool such as a Web browser or a Java platform to provide an interface, then client libraries need to be taken into account.

- **Software Environment.** The correct operation of the software is dependent on a wider environment including; hardware platform, operating system, programming languages and compilers, software libraries, other software packages, and access to peripherals (e.g. a high-definition graphics system may run differently according to the resolution of the display). Each of these factors is not in the direct control of the software developer, and each also goes through a series of versions. Such dependencies must be recorded. Further, artefacts have different requirements on their environments. Binaries usually require an

exact match of the environment to function; source code may function with a different environment, given a compatible compiler and libraries; while designs may be reproducible even with different programming language, given sufficient effort to recode.

- **Software Architecture.** The software architecture can play a significant part in the reproducibility of the function of the software. For example, client/server, peer-to-peer, and Grid systems all require different forms of distributed system interaction which would require the configuration of hardware and software to be reproduced to reproduce the correct behaviour.

- **Operating Performance.** The performance of the software with respect to its use of resources (as opposed to its performance in replaying its content) may play a significant part of the reproducible behaviour of software. For example, speed of execution, data storage requirements, response time of input and output devices, characteristics of specialised peripheral devices (e.g. resolution of plotters, screens or scanners), colour resolution capability may all be important. Note that in some circumstances, we may wish to replay the software at the original operating performance rather than a later improved performance. A notable example of this is games software, which if reproduced at a modern processor's speed would be too fast for a human user to play.

The InSPECT project proposes a taxonomy of properties into five categories [5]:

1. **Content**. The abstract content of an expression of an intellectual work; thus the components of the work according to the digital object's abstract data model.
2. **Context**. The environment in which the content was created or that affects it intended meaning. InSPECT also uses this term to capture the *provenance* of an object, in terms of its creators and source.
3. **Rendering**. Information which contributes to the recreation of the message.
4. **Structure**. Relationship between two of more types of content; in practice this seems to reflect the physical components of the object.
5. **Behaviour**. Information which describes interaction with external stimuli.

A possible mapping of these categories onto the InSPECT categories is given in the following table.

InSPECT Category	Software Property	Comments
• **Content**. The abstract content of an expression of an intellectual work.	• **Software Composition.**	The abstract content of the intellectual work is reflected in the code modules and their relationships, build and configuration, and in the resulting binaries and associated documentation.
• **Context**. The environment in which the content was created or that affects it intended meaning.	• **Functional Description** • **Provenance and Ownership** • **Software Environment** • **Software Architecture** • **Operating Performance**	The context of the operation of the software is described by the inputs and outputs of the software, and the computing environment and architecture in which the software operates to a desired operating performance. Context also

		includes provenance and ownership
• **Rendering**. Information which contributes to the recreation of the message.	• **Software Composition, especially build and installation components** • **Software Environment** • **Software Architecture**	Rendering are those factors which determine the recreation of the "message", which in the case of software is how the software is *executed,* via the combination of the capabilities of the software architecture and environment, together with (for source code at least) the source code components, their dependencies and their compilation within a particular environment.
• **Structure**. Relationship between two of more types of content.	• **Software Composition into a number of components and files.**	InSPECT's Structure category are those properties which physically reconstruct the object, which in the case of software is the components of the software and their distribution into files.
• **Behaviour**. Information which describes interaction with external stimuli.	• **Functional Description** • **User Interaction**	Software's interaction with external stimuli is determined by the functional behaviour of the software to inputs and outputs together with the user interaction model.

The InSPECT categories were defined with data objects in mind, and they do not necessarily map well to software objects, with several categories of software properties contributing to a single InSPECT category, and also categories of software categories contributing to more than one InSPECT category. Also note that some of the mappings do not quite match expectations; "Rendering", which for most data object describes the "appearance" of the object on an output device, for software becomes instead how the software is compiled and executed in a specific environment, while the user interaction (which includes layout and appearance on an output device – usually thought of as "rendering" - actually fits better into the category of "Behaviour". More clarification may be needed here to reconcile the InSPECT approach to software, but it would seem appropriate at present to leave these categories to one side and concentrate on the categories identified for software.

Note that one of the categories of properties is encapsulated in the conceptual model of software itself; that is the breakdown of the software structure into sub-entities, versions and entities, and into components with dependencies between components. For the other six categories, we can give different significant properties for different entities in the model. We consider each in turn. Note that as specified earlier, we do not give details on the significant properties of the user interaction.

8.1.1 Package Properties

Packages properties provide general and provenance information on the system, including general descriptions of functionality and architecture, ownership of the system, overall licence, tutorial material, requirements and purpose of the package. We would also expect a general classification of the system within a controlled vocabulary to refer to a package. The following properties are associated with a Package.

Property Category	Software Property	
Functionality	**purpose**	Description of overall functionality of software system
	keyword	Classification of software under a specified controlled vocabulary
Provenance and Ownership	**package_name**	Name of the package
	owner	Owner of the package, with contact details
	licence	Overall licensing agreement
	location	URL of website of software
Software Environment	-	
Software Architecture	**overview**	Overview of software architecture
Operating Performance	-	
Software Composition	**software overview**	Documentation on the overview of the software
	tutorials	Teaching material on the system.
	requirements	requirements of package

8.1.2 Version Properties

Versions are associated with a release with specific functionality, and would typically provide access to source code modules within specific programming languages, which would be provided with a build and install instructions to establish the version on a specific machine. Thus the properties associated with a version would describe the function of the version in detail, dependencies on architecture, device types and programming languages, and provide installation and manual material. The following properties are associated with a software version:

Property Category	Software Property	
Functionality	**functional_description**	Description of relationship of between inputs and outputs of the version.
	release_notes	Description of changes of this version from other versions.
	algorithm	Description of the algorithm used.
	input_parameter	Details of names and formats of inputs
	output_parameter	Details of names and formats of outputs

	interface	API description
	error_handling	Description of how errors are handled.
Provenance and Ownership	**version_identifier**	Identifier for this particular version
	licence	Licence specific to this version.
Software Environment	**programming_language**	Programming language used for this version.
	hardware_device	Category of hardware device which the software version depends upon.
Software Architecture	**detailed_architecture**	Detailed description of architectural dependencies of the version.
	dependent_package	Dependency on another software package being installed.
Operating Performance	-	
Software Composition	**source**	Source code modules for this version.
	manual	Usage instructions for this version
	installation	Installation, build and configuration instructions for this version.
	test_cases	Test suite for this version.
	specification	Specification of this version

8.1.3 Variant Properties

A variant is associated with an adaptation of a version for a specific target environment. Usually it would be associated with an executable binary, but also could provide addition source modules which are tailored to the target environment. Thus we would expect details of the environment, with specific dependencies,, and also the expected operating characteristics in such an environment. The following properties are associated with a software variant:

Property Category	Software Property	
Functionality	**variant_notes**	Description of the variations in behaviour specific to this variant.
Provenance and Ownership	**licence**	Licence specific to this variant.
Software Environment	**platform**	Target hardware machine architecture of version.
	operating_system	Version of operating system
	compiler	Version of compiler used to construct this variant.
	dependent_library	Version of dependent software libraries used.
	hardware_device	Specific auxiliary hardware devices supported by the variant.

Software Architecture	**dependent_package**	Dependency on another software package being installed.
Operating Performance	**processor_performance**	A specification that a specific speed of processor is required.
	memory_usage	Minimal/typical memory usage for RAM and disk of the variant.
	peripheral_performance	Performance of specific peripheral hardware, for example screen or colour resolution, audio range.
Software Composition	**binary**	Machine executable code for this version.
	source	variants of source modules for this version
	configuration	installation and configuration instructions for this variant

8.1.4 Download Properties

A download is associated with a number of different files stored at specific locations on a specific machine. Thus we would expect to find properties identifying the components. The following properties are associated with a software download:

Property Category	Software Property	
Functionality	-	
Provenance and Ownership	**licensee**	Named licensee of the download
	conditions	Local conditions of use of this download.
	licence_code	Licence key value
Software Environment	**environment_variable**	Specific settings for environmental variables.
	IP_address	Specific IP address
	hardware_address	Specific MAC address (or equivalent) identifying a specific machine.
Software Architecture	-	
Operating Performance	-	
Software Composition	**file**	Names and addresses of specific files in the download.

8.2 Component Properties

Components can also have properties associated with them, which can overlap with the properties of the version or variant they are in, depending on the detail required. They could

have most of the functional and environmental properties associated with versions or variants, so the following table is a selection of the properties available.

Property Category	Software Property	
Functionality	**functional_description**	Description of relationship of between inputs and outputs of the version.
	release_notes	Description of changes of this version from other versions.
	algorithm	Description of the algorithm used.
	input_parameter	Details of names and formats of inputs
	output_parameter	Details of names and formats of outputs
	interface	API description
	error_handling	Description of how errors are handled.
Provenance and Ownership	**licence**	Licence specific to this version.
Software Environment	**programming_language**	Programming language used for this component.
	hardware_device	Category of hardware device which the software version depends upon.
	dependent_library	Version of dependent software libraries used.
Software Architecture	**detailed_architecture**	Detailed description of architectural dependencies of the version.
	dependent_package	Dependency on another software package being installed.
Operating Performance	**-**	

8.3 An example of Significant Properties: Xerces

As an example of assigning significant properties, we consider Xerces[76]. Xerces is a "collaborative software development project dedicated to providing robust, full-featured, commercial-quality, and freely available XML parsers". The main product is a robust reliable and memory efficient XML parser which takes a XML document (usually from a text file or input stream), and provides machine access to the structured content of the XML file via one of a number of supported APIs. It is an established product which has been available for a number of years, is supported in three major languages (Java, C++, Perl) and has gone through a number of versions. Being an open source product with a community development, the details of the system are easily available, and being a commonly used XML format processor, it provides a useful example of the kind of tool which needs to be preserved in many software contexts, as a sub-package of scientific software, or for document processing. Thus for our

[76] http://xerces.apache.org/

purposes, it provides an example of a relatively simple software package which we can assign significant properties to.

Even in this case of a relatively self-contained system which has had most of its development and documentation in public, raises issues and problems. For example, it is not entirely clear what constitutes the package. There are several developments in different languages (Java, C, Perl), so should we treat this as one package with several language versions, or as a separate package for each development. This is a judgement; for Xerces we can say that each language version has an different team with a separate development programme, so it may make sense to treat them as separate packages,

Thus for Xerces C (the C language version) we break the system down in conceptual model and assign properties as follows. Note that in many cases we refer to a file or URL which give further details associated with the property.

Package Properties

Property Category	Software Property	
Functionality	**purpose**	XML Parser
	keyword	D.3.4 Processors: Parsing (ACM Computing Classification System)
Provenance and Ownership	**package_name**	Xerces C
	owner	Apache Software Foundation
	licence	Apache Software Licence v. 2.0 http://www.apache.org/licenses/LICENSE-2.0.html
	location	http://xerces.apache.org/xerces-c/
Software Environment	-	
Software Architecture	**overview**	Module to convert input text files into machine processable data structures http://xerces.apache.org/xerces-c/index.html
Operating Performance	-	
Software Composition	**software overview**	http://xerces.apache.org/xerces-c/index.html
	tutorials	http://xerces.apache.org/xerces-c/program.html
	requirements	

Version Properties

If we chose version 2.8.0 we can get the following properties.

Property Category	Software Property

Functionality	**functional_description**	as package
	release_notes	Description of changes of this version from other versions.
	algorithm	Description of the algorithm used.
	input_parameter	files using XML 1.0, XML 1.1, XML Namespaces, XML Schema 1.0
	output_parameter	DOM level 1 & 2, SAX 1 & 2
	interface	API description at http://xerces.apache.org/xerces-c/apiDocs/index.html
	error_handling	FAQ at http://xerces.apache.org/xerces-c/faqs.html Bug reports at http://xerces.apache.org/xerces-c/bug-report.html
Provenance and Ownership	**version_identifier**	Version 2.8.0
	licence	as package
Software Environment	**programming_language**	C++
	hardware_device	-
Software Architecture	**detailed_architecture**	-
	dependent_package	-
Operating Performance	-	
Software Composition	**source**	Source code modules for this version at http://xerces.apache.org/xerces-c/download.cgi
	manual	Manual at http://xerces.apache.org/xerces-c/program.html
	installation	Installation, at : http://xerces.apache.org/xerces-c/install.html Build at:http://xerces.apache.org/xerces-c/build.html
	test_cases	samples at http://xerces.apache.org/xerces-c/samples.html
	specification	Specification of this version

Variant properties

If we then chose the variant Xerces-C++ Version 2.8.0 for Linux x86-64, we can give the following properties.

Property Category	Software Property	
Functionality	**variant_notes**	http://xerces.apache.org/xerces-c/build-winunix.html#UNIX .

Provenance and Ownership	**licence**	as package
Software Environment	**platform**	x86 architecture
	operating_system	Linux x86-64
	compiler	GCC 3.4.x or later
	dependent_library	-
	hardware_device	-
Software Architecture	**dependent_package**	GZIP, TAR, GNU Make (for source)
Operating Performance	**processor_performance**	-
	memory_usage	-
	peripheral_performance	-
Software Composition	**binary**	xerces-c_2_8_0-x86_64-linux-gcc_3_4.tar.gz
	source	-
	configuration	http://xerces.apache.org/xerces-c/build-winunix.html#UNIX

Download properties

We then assume for a specific installation we would give the following information.

Property Category	**Software Property**	
Functionality	-	
Provenance and Ownership	**licensee**	*user x*
	conditions	see licence
	licence_code	none
Software Environment	**environment_variable**	XERCESROOT set to a specific path
	IP_address	-
	hardware_address	-
Software Architecture	-	
Operating Performance	-	
Software Composition	**file**	location of executable on machine.

9 Conclusions and Recommendations

9.1 Conclusions of Study

Software preservation is a relatively new topic of research and there is little practical experience in the field of software preservation per se.

- As we have already noted, although there are many groups who are holding software to support archives, or to support a community, and many others who are maintaining a usable software package for a long time, these groups do not consider themselves to be doing software preservation and have other priorities.

- Those that do carry out software preservation are often amateur or specialised in science museums or special interest groups; these are often ad hoc and small scale and do not tackle the systematic problems of keeping software replayable in a broad context for the long term.

- Other projects are looking more systematically at digital preservation and tend to rely on the persistence of software, or at least access to software with similar functionality to the original. However, they also tend not to concentrate on the problem of how to preserve the software itself.

Thus the area of software preservation is relatively open, which lead us to consider why software might need to be preserved in the first place, and if it was preserved, what this would mean.

- There are good reasons of preserving research effort in software.

- Preserving software is a vital adjunct for preserving other digital objects.

- Preserving software essentially means that software can be reconstructed and replayed to behave sufficiently closely to the original.

- Software is inherently complex with a large number of components related in a dependency graph, and with specification, source and binary components, and a highly sensitive dependency on the operating environment. Handling this complexity is a major barrier to the preservation of software.

- Different preservation approaches can be adopted which can execute binaries directly, can emulate the software, or carry out software migration by recompiling source code, or even recoding. All can in different circumstances support good preservation.

Adopting the notion of performance from the NAA and InSPECT, we developed a notion of performance of software which is closely related to the adequacy of the performance on the target data. The adequacy of the software can therefore be established via testing.

- Establishment and preservation of test cases for expected behaviour of end software on test data is a key feature for assessing the adequacy of performance of software preservation on specific chosen significant properties.

- Good software engineering practice to support software version control, software maintenance, migration and especially software testing can also support software preservation. Groups which have successfully maintained software over a long period, such a NAG or StarLink have developed rigorous software engineering practice and developed techniques to support software migration in particular.

- Software reuse, via code classification and libraries, and also code canonicalization and generalisation can also assist good software preservation.

To capture and control the inherent complexity of software, we have developed a conceptual model for software which is more complex than that of InSPECT, although it does have some parallels. Many of the structuring significant properties of software are thus captured in this model. Significant properties of software are then categorised according to this model and also according to their role:

- The InSPECT categorisation of significant properties does not match comfortably with the significant properties of software. This is probably because of the indirect performance model of software, which is tested by the performance of the end data.

- Contextual significant properties play a key role and software is dependent upon them being satisfied for satisfactory reconstruction and replay.

- Behavioural significant properties determine the performance of the software on end data.

Given the relatively immature state of the art in software preservation, we consider our definition of a conceptual model of software and the associated identification and classification of significant properties to be a proposal, which needs to be evaluated further in practice to judge its value and effectiveness in practice.

The significant properties identified in this study are still relatively general and do not go into the detail of other significant studies. For example, we decided that we would stop at the level of granularity of code represented by the common coding concept represented by a public class or module or subroutine (terminology varies between programming language) and it would not be worthwhile detailing any further. Other significant properties also stop at a high level, and do not for example enumerate the possible values which they could take[77]. Further testing and evaluation is required to see if this is sufficient and whether the significant properties are always appropriate and whether they can be extracted and used in practice.

Tools support should be eventually forthcoming to support the significant properties of software; however, we feel that the above development of the methodology needs to be investigated further before rushing into tool support.

[77] As an example of how this may be undertaken, see the example of licensing in Appendix C.

9.2 Recommendations

We conclude the report with a set of recommendations for JISC.

- Raising awareness of the software preservation within the JISC community.
- Further consideration should be given to the justification and role of software preservation within a broader digital preservation strategy.
- Specific consideration should be given to the role of software preservation in preservation processes which are conformant to OAIS.
- Further studies should be undertaken to test and extend the notion the conceptual model of software and its significant properties. Studies and test cases should be undertaken specifically in areas which were seen as outside the scope of this study, in particular:
    - Database software
    - Commercial software
    - Business and office software
    - Software which supports the performance of other key digital objects (e.g. documents, vector images, science data).
    - Systems and networking software.
- Studies and practice of software preservation should involve experience software engineers to introduce best practice in code development, testing maintenance and reuse.
- Specialist consideration should be given to the problem of preserving the user interaction model of a software package.
- Guidance developed on the relative value of adopting an emulation or a migration strategy to support the preservation of software.
- Reconsideration of the categories of significant properties identified in InSPECT and those appropriate for software.
- Development of methodologies to exploit software testing as a measure of adequacy of performance.

Acknowledgements

This project was funded by the JISC Digital Preservation Programme and the copyright of this document is the property of JISC. We would like to thank the programme managers Helen Hockx-Yu and Neil Grindley for their support. We would like to thank the following for contributing with discussions and visits during this study: Chris Greenhough and David Worth (CCPForge); Mike Dewar and Jeremy Walton (NAG Ltd), Sam Pepler (BADC); Mike Coyne, David Duce and Bob Hopgood (JISC Significant Properties of Vector Graphics Project), Gareth Knight (InSpect), and the CASPAR, SCARP and DCC projects for their collaboration.

References

[1] Margaret Hedstrom, Christopher A. Lee, *Significant properties of digital objects: definitions, applications, implications*, Proceedings of the DLM-Forum 2002. http://www.ils.unc.edu/callee/sigprops_dlm2002.pdf.

[2] M. Coyne, D. Duce, B. Hopgood, G. Mallen, M. Stapleton. *The Significant Properties of Vector Images.* Report of JISC Digital Preservation Programme Study, November 2007.

[3] John G. Zabolitzky *Preserving Software: Why and How*. Iterations: An Interdisciplinary Journal of Software History, vol. 1, 13 September 2002. http://www.cbi.umn.edu/iterations/zabolitzky.html

[4] The Cedars Guide to Digital Preservation Strategies http://www.leeds.ac.uk/cedars/guideto/dpstrategies/dpstrategies.html (2002)

[5] Gareth Knight, *Inspect Project Report,* draft, King's College London, October 2007.

[6] H Heslop, S. Davis, A. Wilson. *An Approach to the Preservation of Digital Records*, National Archives of Australia, 2002. http://www.naa.gov.au/Images/An-approach-Green-Paper_tcm2-888.pdf

[7] Adam Farquhar, Helen Hockx-Yu. *Planets: Integrated Services for Digital Preservation.* The International Journal of Digital Curation, 2(2), 2007. http://www.ijdc.net/ijdc/article/view/46/59

[8] Lorie, R. A. 2001. *Long term preservation of digital information*. In Proceedings of the 1st ACM/IEEE-CS Joint Conference on Digital Libraries (Roanoke, Virginia, United States). JCDL '01. ACM, New York, NY, 346-352. DOI= http://doi.acm.org/10.1145/379437.379726

[9] A. Brown. *Representation Information Registries*, Planets project white paper, March 2008. http://www.planets-project.eu/docs/reports/Planets_PC3-D7_RepInformationRegistries.pdf

[10] ISO 14721:2003: Space data and information transfer systems -- Open archival information system -- Reference model http://public.ccsds.org/publications/archive/650x0b1.pdf

[11] ACM *Collected algorithms from ACM index by subject to algorithms* (1960–1976)

[12] Boisvert R F, Howe S E and Kahaner D K *GAMS – a framework for the management of scientific software*. ACM Trans. Math. Software 11 313–355 (1985).

[13] IFLA Study Group. *Functional Requirements for Bibliographic Records, 1998.* http://www.ifla.org/VII/s13/frbr/frbr.pdf

[14] PREMIS Working Group. *Data Dictionary for Preservation Metadata* http://www.oclc.org/research/projects/pmwg/premis-final.pdf

[15] Andrew Wilson *Significant Properties Report* InSPECT Work Package 2.2, April 2007.

Appendix A: An OWL Ontology for Significant Properties of Software

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns="http://www.owl-ontologies.com/Ontology1206345526.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xml:base="http://www.owl-ontologies.com/Ontology1206345526.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="File">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >A physical unit of storage on a particular machine.</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="UserGuide">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Guidance to the intended end user of the operation of the software
package.</rdfs:comment>
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Document"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Specification">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >A design document whch provides a definitive description of the
functional and performance characteristics of a software
package.</rdfs:comment>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Document"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Package">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Represents the whole conceptual entity of an identifiable software
system, with a unity of functioanal putpose, comon overall name and
responsible "owner".</rdfs:comment>
    <rdfs:subClassOf>
      <owl:Class rdf:ID="System"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Manual">
    <rdfs:subClassOf rdf:resource="#UserGuide"/>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >A  description of the operation of the package of particular features
and functions providied in the package.  Intended as reference
material.</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:about="#Document">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >A component which provides human readable documentation of the
package.</rdfs:comment>
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Component"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Source">
```

```
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >A component contaning human readable source code written in a formal
machine programming language.  Requires a compilation or interpretation
phase to be executed.</rdfs:comment>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Component"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="OtherComponent">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Component"/>
    </rdfs:subClassOf>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Any other software artifact not covered by the other sub-classes of
component.</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="Licence">
    <rdfs:subClassOf rdf:resource="#Document"/>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Document stating the legal ownership of the package and the
condiitiions of use under which it is provided to the user and the user can
operate the software.</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="Requirements">
    <rdfs:subClassOf rdf:resource="#Document"/>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >A description of the requirements of the systems, the behavioural and
performance characteristics which the package is intended to
satisfy.</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="InstallationGuide">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Provide instructions on the correct configuration and installaion of
the package in a particular environment,</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#Document"/>
  </owl:Class>
  <owl:Class rdf:ID="Installation">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >A configuration which controls the installation of a package.
Typically will check environmental dependencies (e.g. the existance and
version of dependent external packages or hardware), set environmental
variables, set code variables and alternate code block choices, invoke build
scripts.</rdfs:comment>
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Configuration"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="TestSuite">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Representative examples of operation of the package and expected
behaviour arising from operatioin of the package.  Prodiced to test the
conformance of the package to expected behaviour in a particular
installation environment.</rdfs:comment>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Component"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="ErrorList">
    <rdfs:subClassOf rdf:resource="#Document"/>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
```

```
    >List of known errors, issues, and warnings associated with a version or
variant of the software.</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:about="#Configuration">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >A component which describes the configuration of the components to
generate a working  dependencies between components and</rdfs:comment>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Component"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Version">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#System"/>
    </rdfs:subClassOf>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >expression of the system which has variations in  the functionality of
the system</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="Binary">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Component"/>
    </rdfs:subClassOf>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >A component which is in a machine readable encoding which can be
executed directly by the compujter hardware, or via a Virtual
Machine.</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:about="#Component">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >An identifiable item which forms a part of a complete software
package</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="Download">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >An  instance of the system on a particular machine</rdfs:comment>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#System"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Variant">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#System"/>
    </rdfs:subClassOf>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >a manifestation of the system which changes in environment (e.g.
operating system, hardware platform, library, programming language
version)</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="Tutorial">
    <rdfs:subClassOf rdf:resource="#UserGuide"/>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >Instructional material provided to the user to enable the end user to
learn the correct operation of the package, typically providing an overview
of the package and a staged introduction to its features.  Will also usually
provide examples and walkthroughs.</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="VersionNotes">
    <rdfs:subClassOf rdf:resource="#Document"/>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
```

```
    >Provides a description on the particular features of a version,
outlining it functional differences and environemntal dependencies which
differ from a previous version.</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="Config">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >A configuration component with typically a number of system internal
and environmental variables set to configure the system in a particular
operating environment.</rdfs:comment>
    <rdfs:subClassOf rdf:resource="#Configuration"/>
  </owl:Class>
  <owl:Class rdf:about="#System">
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >A package is a collection of components which are treated together to
provide an item of software.</rdfs:comment>
  </owl:Class>
  <owl:Class rdf:ID="Build">
    <rdfs:subClassOf rdf:resource="#Configuration"/>
    <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
    >A congiguration of the source files, with the dependencies of the
source code, typically to control the compilation order and dependencies
between them.</rdfs:comment>
  </owl:Class>
  <owl:ObjectProperty rdf:ID="versionIdentifier">
    <rdfs:subPropertyOf>
      <owl:ObjectProperty rdf:ID="provenanceProperty"/>
    </rdfs:subPropertyOf>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="errorHandling">
    <rdfs:subPropertyOf>
      <owl:ObjectProperty rdf:ID="functionalProperty"/>
    </rdfs:subPropertyOf>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="source">
    <rdfs:subPropertyOf>
      <owl:ObjectProperty rdf:ID="compositionProperty"/>
    </rdfs:subPropertyOf>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="significantProperty">
    <rdfs:domain rdf:resource="#System"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="subVariantDownloadOf">
    <rdfs:domain rdf:resource="#Download"/>
    <rdfs:range rdf:resource="#Download"/>
    <rdfs:subPropertyOf>
      <owl:ObjectProperty rdf:ID="modelProperties"/>
    </rdfs:subPropertyOf>
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="hasSubVariantDownload"/>
    </owl:inverseOf>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="programmingLanguage">
    <rdfs:subPropertyOf>
      <owl:ObjectProperty rdf:ID="environmentalProperty"/>
    </rdfs:subPropertyOf>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="devicePerformance">
    <rdfs:subPropertyOf>
      <owl:ObjectProperty rdf:ID="performanceProperty"/>
    </rdfs:subPropertyOf>
  </owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="specification">
  <rdfs:subPropertyOf>
    <owl:ObjectProperty rdf:about="#compositionProperty"/>
  </rdfs:subPropertyOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#environmentalProperty">
  <rdfs:subPropertyOf rdf:resource="#significantProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="platform">
  <rdfs:subPropertyOf rdf:resource="#environmentalProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="processorPerformance">
  <rdfs:subPropertyOf>
    <owl:ObjectProperty rdf:about="#performanceProperty"/>
  </rdfs:subPropertyOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="operatingSystem">
  <rdfs:subPropertyOf rdf:resource="#environmentalProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#functionalProperty">
  <rdfs:subPropertyOf rdf:resource="#significantProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="manual">
  <rdfs:subPropertyOf>
    <owl:ObjectProperty rdf:about="#compositionProperty"/>
  </rdfs:subPropertyOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="variantNotes">
  <rdfs:subPropertyOf rdf:resource="#functionalProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasSubPackageVersion">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="subPackageVersionOf"/>
  </owl:inverseOf>
  <rdfs:domain rdf:resource="#Version"/>
  <rdfs:range rdf:resource="#Version"/>
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#performanceProperty">
  <rdfs:subPropertyOf rdf:resource="#significantProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="keyword">
  <rdfs:subPropertyOf rdf:resource="#functionalProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="outputParameter">
  <rdfs:subPropertyOf rdf:resource="#functionalProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#provenanceProperty">
  <rdfs:subPropertyOf rdf:resource="#significantProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="subPackageOf">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="hasSubPackage"/>
  </owl:inverseOf>
  <rdfs:range rdf:resource="#Package"/>
  <rdfs:domain rdf:resource="#Package"/>
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="algorithm">
  <rdfs:subPropertyOf rdf:resource="#functionalProperty"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="memoryPerformance">
  <rdfs:subPropertyOf rdf:resource="#performanceProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasSubPackage">
  <rdfs:domain rdf:resource="#Package"/>
  <rdfs:range rdf:resource="#Package"/>
  <owl:inverseOf rdf:resource="#subPackageOf"/>
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="releaseNotes">
  <rdfs:subPropertyOf rdf:resource="#functionalProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasSubVariantDownload">
  <owl:inverseOf rdf:resource="#subVariantDownloadOf"/>
  <rdfs:range rdf:resource="#Download"/>
  <rdfs:domain rdf:resource="#Download"/>
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="owner">
  <rdfs:subPropertyOf rdf:resource="#provenanceProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="dependentPackage">
  <rdfs:subPropertyOf>
    <owl:ObjectProperty rdf:ID="architectureProperty"/>
  </rdfs:subPropertyOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="softwareOverview">
  <rdfs:range rdf:resource="#Component"/>
  <rdfs:subPropertyOf>
    <owl:ObjectProperty rdf:about="#compositionProperty"/>
  </rdfs:subPropertyOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="inputParameter">
  <rdfs:subPropertyOf rdf:resource="#functionalProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="purpose">
  <rdfs:subPropertyOf rdf:resource="#functionalProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="fileOfComponent">
  <rdfs:range rdf:resource="#Component"/>
  <rdfs:domain rdf:resource="#File"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="componentInFile"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#compositionProperty">
  <rdfs:range rdf:resource="#Component"/>
  <rdfs:subPropertyOf rdf:resource="#significantProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="licence">
  <rdfs:subPropertyOf rdf:resource="#provenanceProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="interface">
  <rdfs:subPropertyOf rdf:resource="#functionalProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="homeLocation">
  <rdfs:subPropertyOf rdf:resource="#provenanceProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#architectureProperty">
  <rdfs:subPropertyOf rdf:resource="#significantProperty"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="detailedArchitecture">
  <rdfs:subPropertyOf rdf:resource="#architectureProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hardwareDeviceType">
  <rdfs:subPropertyOf rdf:resource="#environmentalProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="functionalDescription">
  <rdfs:subPropertyOf rdf:resource="#functionalProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="installation">
  <rdfs:subPropertyOf rdf:resource="#compositionProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="tutorial">
  <rdfs:subPropertyOf rdf:resource="#compositionProperty"/>
  <rdfs:range rdf:resource="#Component"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="configuration">
  <rdfs:subPropertyOf rdf:resource="#compositionProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hardwareDevice">
  <rdfs:subPropertyOf rdf:resource="#environmentalProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="binary">
  <rdfs:subPropertyOf rdf:resource="#compositionProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="testCases">
  <rdfs:subPropertyOf rdf:resource="#compositionProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="requirements">
  <rdfs:subPropertyOf rdf:resource="#compositionProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#componentInFile">
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
  <owl:inverseOf rdf:resource="#fileOfComponent"/>
  <rdfs:domain rdf:resource="#Component"/>
  <rdfs:range rdf:resource="#File"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="userInteractionProperty">
  <rdfs:subPropertyOf rdf:resource="#significantProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="package_name">
  <rdfs:domain rdf:resource="#Package"/>
  <rdfs:subPropertyOf rdf:resource="#provenanceProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="architectureOverview">
  <rdfs:subPropertyOf rdf:resource="#architectureProperty"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#subPackageVersionOf">
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
  <owl:inverseOf rdf:resource="#hasSubPackageVersion"/>
  <rdfs:range rdf:resource="#Version"/>
  <rdfs:domain rdf:resource="#Version"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="subVersionVariantOf">
  <rdfs:domain rdf:resource="#Variant"/>
  <rdfs:range rdf:resource="#Variant"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="hasSubVersionVariant"/>
  </owl:inverseOf>
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:about="#hasSubVersionVariant">
  <owl:inverseOf rdf:resource="#subVersionVariantOf"/>
  <rdfs:domain rdf:resource="#Variant"/>
  <rdfs:range rdf:resource="#Variant"/>
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
</owl:ObjectProperty>
<owl:TransitiveProperty rdf:ID="childVersion">
  <rdfs:range rdf:resource="#Version"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  <rdfs:domain rdf:resource="#Version"/>
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
  <owl:inverseOf>
    <owl:TransitiveProperty rdf:ID="parentVersion"/>
  </owl:inverseOf>
</owl:TransitiveProperty>
<owl:TransitiveProperty rdf:about="#parentVersion">
  <owl:inverseOf rdf:resource="#childVersion"/>
  <rdfs:range rdf:resource="#Version"/>
  <rdfs:domain rdf:resource="#Version"/>
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
</owl:TransitiveProperty>
<owl:FunctionalProperty rdf:ID="versionOf">
  <owl:inverseOf>
    <owl:InverseFunctionalProperty rdf:ID="hasVersion"/>
  </owl:inverseOf>
  <rdfs:domain rdf:resource="#Version"/>
  <rdfs:range rdf:resource="#Package"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="componentOf">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
  <owl:inverseOf>
    <owl:InverseFunctionalProperty rdf:ID="hasComponent"/>
  </owl:inverseOf>
  <rdfs:range rdf:resource="#System"/>
  <rdfs:domain rdf:resource="#Component"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="variantOf">
  <owl:inverseOf>
    <owl:InverseFunctionalProperty rdf:ID="hasVariant"/>
  </owl:inverseOf>
  <rdfs:domain rdf:resource="#Variant"/>
  <rdfs:range rdf:resource="#Version"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="fileInDownload">
  <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  <rdfs:domain rdf:resource="#File"/>
  <rdfs:range rdf:resource="#Download"/>
  <owl:inverseOf>
    <owl:InverseFunctionalProperty rdf:ID="downloadIncludesFile"/>
  </owl:inverseOf>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="downloadOf">
  <owl:inverseOf>
    <owl:InverseFunctionalProperty rdf:ID="hasDownload"/>
```

```
    </owl:inverseOf>
    <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
    <rdfs:domain rdf:resource="#Download"/>
    <rdfs:range rdf:resource="#Variant"/>
  </owl:FunctionalProperty>
  <owl:InverseFunctionalProperty rdf:about="#hasVariant">
    <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
    <rdfs:domain rdf:resource="#Version"/>
    <rdfs:range rdf:resource="#Variant"/>
    <owl:inverseOf rdf:resource="#variantOf"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  </owl:InverseFunctionalProperty>
  <owl:InverseFunctionalProperty rdf:about="#downloadIncludesFile">
    <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
    <rdfs:range rdf:resource="#File"/>
    <rdfs:domain rdf:resource="#Download"/>
    <owl:inverseOf rdf:resource="#fileInDownload"/>
  </owl:InverseFunctionalProperty>
  <owl:InverseFunctionalProperty rdf:about="#hasVersion">
    <rdfs:domain rdf:resource="#Package"/>
    <rdfs:range rdf:resource="#Version"/>
    <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
    <owl:inverseOf rdf:resource="#versionOf"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  </owl:InverseFunctionalProperty>
  <owl:InverseFunctionalProperty rdf:about="#hasDownload">
    <rdfs:domain rdf:resource="#Variant"/>
    <rdfs:range rdf:resource="#Download"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
    <owl:inverseOf rdf:resource="#downloadOf"/>
    <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
  </owl:InverseFunctionalProperty>
  <owl:InverseFunctionalProperty rdf:about="#hasComponent">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
    <rdfs:subPropertyOf rdf:resource="#modelProperties"/>
    <rdfs:domain rdf:resource="#System"/>
    <rdfs:range rdf:resource="#Component"/>
    <owl:inverseOf rdf:resource="#componentOf"/>
  </owl:InverseFunctionalProperty>
</rdf:RDF>

<!-- Created with Protege (with OWL Plugin 3.3.1, Build 430)
http://protege.stanford.edu -->
```

Appendix B: Questions to Analyse Software Preservation Practices

Significant Properties of Software

Significant properties, also referred to as “significant characteristics” or “essence”, are essential attributes of a digital object which affect its appearance, behaviour, quality and usability. They can be grouped into categories such as content, context (metadata), appearance (e.g. layout, colour), behaviour (e.g. interaction, functionality) and structure (e.g. pagination, sections). This is defined within the context of preservation; significant properties must be preserved over time for the digital object to remain accessible and meaningful.

Significant properties have been considered in for a number of digital objects, such as text documents and raster images. However, to date there has been no substantial study specifically into software. In this study, we are considering how this concept might apply to software and develop a basic framework for capturing the significant properties of software.

We recognise that we cannot consider the whole of software - this is a very large topic in itself. But we would like to consider a number of examples within the scientific and mathematical domain. In order to develop this framework, we would like to consider a number of scenarios where software is developed and stored for a significant period of time, such as long term development projects or software libraries and repositories.

Within each scenario, we would like to consider for each software "object": (taken from CASPAR requirements capture process):

- conceptual purpose of software items (community, domain of interest, algorithms)
- knowledge/behaviour encoded within the software
- encodings, programming languages, input/outputs data formats.
- supporting documentation objects.
- relationships and dependencies between software items
- relationships and dependencies to the s/w environment
- storage mechanisms

and additional information about:

- production - description of the way in which the software is written and used and maintained, and how it gets to the repository
- current use:
    - software used to manage the digital encodings
    - software/mechanisms to use/perform the software (emulation, configurations, build scripts, manuals)

For each of these broad headings we include some guidance of the intention and also some questions which are relevant.

Outline Questions

Each repository is characterised at the outset by the following basic features.

1. Holdings: overview of the type of software held, and a list of software as much as practical.
2. Software: For a selection of the software in the repository, offering a variety of challenges, provide a description of the digitally encoded information to be preserved, from the bit level to the knowledge it conveys to its user community. We do not at this stage need very detailed descriptions. In addition we need a brief descriptions of
    - any special importance, or special unique quality
    - access restrictions and controls
    - what behaviour the software supports
        - how are inputs and outputs (data types) to data recorded
        - Controlled vocabulary to encapsulate functionality
    - how the required software is located and retrieved
    - what additional environmental knowledge is employed to characterise the software
        - hardware, software, versions, library conditions
    - Software usage process - how is the software meant to be used? Interaction with a (business) process? Software architecture it supports.
3. Software Producer: A brief description of the group, individual or institution that produced the software.
4. User Community: A description of the current user community and the characteristics of the designated community for whom this software might be preserved.
5. Current plans/approaches for:
    - cataloguing s/w
    - maintenance of s/w versions
    - software reuse
    - software preservation

More detailed questions

1 What Performance/Behaviour does your current user(s) expect from this software and what needs preserving?

This is quite an open ended question but it is essential to establish the nature of the information being are attempted to be preserved as this helps to define what needs to be preserved. This definition does not limit the potential information the archive is capable of providing, but rather helps define a minimum level of information to be preserved.

Some possible example of types of software and supported behaviour.

1. Library routine which has a number of inputs in particular formats, produces a number of outputs in particular formats, implementing a particular algorithm
    - e.g. fast-fourier transform,
    - e.g. data format conversion,
    - e.g. compiler (?)

2. software package to support particular set of functions and user interactions on a class of data object
    - e.g. text editor, takes a text file in a particular format, allows the user to perform a set of edit actions via some interface, and save object.
    - e.g. eclipse s/w editor and builder (?)
    - e.g. software visualisation package NAG IRIS Explorer ?
3. software framework for linking and integrating a series of different functions and applications together upon a common data format/storage system.
    - e.g. Starlink
    - e.g. CDAT (BADC) - (?)

2 What information do you provide to a new user, and what support do you give them during their use of the software?

This is to ascertain if there is any useful information would be given to a new user to help them get started using the software. Including manuals, installation scripts and guides, tutorial material. More importantly, it is also intended to "get at" the types of information that are not written down but are typically asked for and are needed by users to produce results.

Inevitably there will always be information in the heads of the people that run the software archive that is not written down, but would be useful to some users both now and in the future. This is also assuming that the people that created and run the software archive are not around to help the "unborn users", so in future they will not have the support.

Typical Questions:

- Do you give out any training material that informs a new user about using the software?
- Do you provide any training days for new users that inform users about the software?
- Do you log support queries and answers?
- Can you think of any information that you or your colleagues know that would be useful to new users that is not written down?

3 How is the software provenance captured?

- Who/where does it come from?
- How is it verified (e.g. any checksums or signatures)?
- How is it packaged (e.g. Zipped , Linux RPM etc) ?
- For one software "object", how many "files" does it consist of? How are they related?
- Is the software regularised to conform to coding or API standards in any way?
- Is information added (e.g. additional metadata, references etc)?

4 How is the software currently catalogued?

- What information do current users need/possess that allows them to locate the software that they are seeking?
- Does the access software utilise any supplementary metadata, e.g. an index database, a thesaurus, a catalogue?

5 Are there any access restrictions?

- Are there any restrictions on whom or how you can access or use the software?
- What are there reasons for these restrictions?
- Who or what imposes them?
- Are these restrictions likely to change over time?

6 Identify common "domain objects" currently used .

Can you provide a listing and definition of all separate data entities (most granular type of data held within file) contained within the software "object"? Can you fully describe any entity relationships?

- e.g. source files, compiled object files, library objects, configuration files, build scripts, documentation files, test cases, examples.
- How do you currently extract and instantiate these entities and their relationships ?

7 What information is required to reconstruct the software objects or reproduce the performance or duplicate the required behaviour?

If the software were to become unusable, could its functions be reconstructed from the technical specifications of the file format, the data entity definitions and their relationships? If not, what further information would be required to do this?

- What external digital resources does the user refer to (specifications, requirements, pseudo-code, tutorial) ?
- What external non digital resources (i.e. books/microfilm) refer to?
- What external bodies/organisation does a user refer to?
- What is the knowledge base and skill set of current user that allows them to use the software effectively?
- How do your current users acquire the knowledge base and skill set which allows them to use the software effectively and what are these?
- What knowledge or skills gap might arise between the current user group and the designated user community?
- What effect would such a gap have upon the usability of the software and on the ability to process it further?
- Is anyone identified as responsible for monitoring the community knowledge base and initiating changes as needed?
- What effect would the permanent loss of such representation have upon the interpretation of the stored data?

8 Structural Representation Information – (non media dependent encoding)

Closely connected with single file formats, but also includes complex inter-related collections of files.

- Provide a list of file format(s) in which the software processes
- Provide a list file format the software artifacts are represented in (e.g. programming language, interpreted bytecode (JVM), processor specific binary).

- What are the technical specifications of this/these file format(s)? The information derived from the technical specification should be sufficient to extract to reassemble and run the software given the appropriate environment.
- Specify any packaging connecting various separate components together
- How do you manage versioning and source control?
- How do you manage providing versions for different software environments (operating system versions, compiler versions, library versions etc)?

9. How is the software physically stored?

- How many independent off-site copies are there?
- What is the physical media upon which the software is stored e.g. CD, SDLT tape (if any)?
    - Can you provide any relevant technical specification and physical description of how the software is mechanically transferred onto the storage media?
- Was there any media specific encoding employed in writing to the physical media?
    - Can you provide decoding instruction which allow the file to be reconstructed?
    - Has any integrity checking mechanism been allowed which will assist in file reconstruction?
    - Is any metadata physically recorded along with the files e.g. time stamps or id of machine writing to the media
- How will the integrity of the software store be maintained?
- What disaster recovery procedures need to be put in place?
- What is the storage medium current lifetime?

Appendix C: A possible categorisation of software licensing

One issue which may need to be addressed is how to provide more specific detail of specific properties, by via for example providing sub-categories or enumerations of possible values for specific properties. Detailing this may be a direction of research.

As an example of this may be achieved, we give here an extract from Lee Courtney's presentation "*Organizing the Attic, Furnishing the Parlor - Considerations for Moving Forward*", from presentations from the Computer History Museum workshop "The Attic & the Parlor: A Workshop on Software Collection, Preservation & Access", May 5, 2006. http://www.softwarepreservation.org/workshop/courtney_Organizing%20the%20Attic%20V1.0.ppt/view. This presentation gives a categorisation of a number of different types of software licence.

Closed proprietary	Source code not released because of proprietary, competitive, or marketplace concerns. (eg: Windows XP)
Available strictly encumbered	Source code released thru agreement strictly restricting use or redistribution of the source code. (Example: HP MPE-V source code available under source code non-disclosure agreement)
Available loosely encumbered	Source code released after signed agreement loosely restricting use or redistribution. (Example: Educational institution or development consortium software. Precedes contemporary open source)
Available unencumbered	Source code released source code into the public domain with no copyright or other licensing burden. (Example: IBM OS/360?)
Open Source	Source code for the system under any of the open source licenses (GPL, LGPL, BSD, Artistic, etc.).
Closed Classified	System owned by government organization for which source code is not available due to security concerns. (Example: DoD AWACS)
Unknown	Unknown IP encumbrance on original source code.

Further, he gives the status of a number of well known software packages. In practice, the significant property would also have to give access to the specific licencing conditions so that software preservers can undertaken preservation actions which respect the licence.

Software Name	**IP Owner**	**Source State**
Fortran	IBM	Available unencumbered
Unix (AT&T and Berkeley)	ATT & U. of California	Available strictly encumbered
Multics	Bull	Closed proprietary
VisiCalc	unknown	Closed proprietary
Smalltalk-72	Xerox?	Available loosely encumbered
OS/360	IBM	Available unencumbered
Mosaic	U. of Illinois	Available loosely encumbered
Algol-60 compiler	unknown	Available unencumbered
Lisp 1.5	unknown	Available unencumbered

Pascal	unknown	Available unencumbered
C	ATT	Available loosely encumbered
TeX	SRI	Available loosely encumbered
DOS	Microsoft	Closed proprietary
Emacs	unknown	Available loosely encumbered
troff	unknown	Available loosely encumbered
APL	IBM	Closed proprietary
Bravo	Xerox?	Closed proprietary
COBOL	IBM	Closed proprietary
Mac OS	Apple	Closed proprietary
Pong	unknown	Closed proprietary