JISC

Framework for Software Preservation

Brian Matthews, Juan Bicarregui, Arif Shaon, Catherine Jones

STFC

Rutherford Appleton Laboratory

Chilton OX11 0QX

UK

April 2009

Abstract

Software preservation has not had detailed consideration as a research topic or in practical application. In this report, we first discuss some of the motivations and problems of software preservation partly due to the high complexity of software artefacts. We then go onto present a conceptual framework to capture and organise the main notions of software preservation, which are required for a coherent and comprehensive approach. This framework has three main aspects.

- A discussion of what it mean to preserve software in of via a *performance model* which considers how a software artefact can be rebuilt from preserved components and can then be seen to be representative of the original software product. This model introduces the concept of *adequacy*, a special case of the general notion of authenticity of preservation. Adequacy seeks to establish the preservation of specific properties of the original artefact via a process of testing against predefined cases to determine whether the software product performs to a acceptable tolerance.

- The development of model of software artefacts, describing the basic components of all software. This model has four layers. Software products representing the whole item developed by some actor for some purpose. This is broken down the notion of a software version, representing changes in the functionality of a product, and variants, representing a version of the software adapted for a specific working environment. Finally, a specific instance (physical manifestation) of the software may be deployed in a specific physical context. This model of software is loosely based on the FRBR model for representing digital artefacts and their history within a library context.

- The definition and categorisation of the properties of software artefacts which are required to ensure that the softare product has been adequately preserved. These are broken down into a number of categories and related to the concepts defined in the OAIS standard. In particular, Preservation Description Information properties are required to manage and recall preserved software objects, Representation Information properties to reconstruct a working copy of the software within a new environment, and Significant Properties are properties to test the performance of the software within the new environment.

Revision History:

Version	Date	Authors	Sections Affected / Comments
0.1	03/03/2009	BMM	Outline
0.2	08/04/2009	BMM	Full Draft
0.3	17/04/2009	AS	Integrated OAIS discussion
0.4	07/05/2009	BMM	Added conclusion for final draft.
0.5	12/05/2009	BMM/AS	Final typos, abstract and contents

Table of Contents

1 Introduction

Software is a class of electronic object which is frequently the result of research and is often a vital pre-requisite to the preservation of other electronic objects. However, there has only been limited consideration of the preservation of software as a digital object in its own right.

Software is inherently complex – forbiddingly so for people who were not involved in its development but nevertheless want to maintain access to software. A typical software artefact has 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. Further, the preservation of software is frequently seen as a secondary activity and one with limited usefulness.

Software preservation is thus a relatively underexplored topic of research and there is little practical experience in the field of software preservation as such. The work in this report, and project, builds on the outputs of a JISC sponsored study into the significant properties of software for preservation[1], which looked at a number of software repositories and other groups engaged in maintaining software over the long term (Matthews et. al. 2008). Given the relative immaturity of the field, the project has devloped a framework to express the notion of software preservation and set out some baseline concepts of what it means to preserve software. The framework has been developed further and deeper, bringing in notions from the OAIS reference model[2], and also developed tool support.

1.1 What do we mean by preservation?

There is a large overlap between preservation for the future and good practice for software maintenance in the here and now. Although the word "preservation" implies for future generations, in software terms this can mean a much shorter time frame – much software has a working life (without additional modifications/rewriting) of five years at best and so curation can be considered to be quite a short term undertaking in the software domain. It is quite possible for the underlying data to have been created much earlier in time than the software used to manipulate it. It is the art of maintaining the underlying data by the way software systems are curated and migrated which is important in this sphere. Whilst this project is concerned with software generated as a result of research rather than other endeavours, it is important to consider the wider landscape before teasing out research output issues.

1.2 Definition of Software

Software is defined as: "*a collection of computer programs, procedures and documentation that perform some task on a computer system*."[3] Computer programs themselves are sequences of

[1] Joint Information Systems Committee (JISC) study into the *Significant Properties of Software* (2007).
[2] http://public.ccsds.org/publications/archive/650x0b1.pdf
[3] http://en.wikipedia.org/wiki/Software

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 *product*[4].

The term software 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 report, such content is considered a data format for a different digital object type, and is thus out of scope.

1.3 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 systems coded by one individual for their own use (typical in research), open-source systems, to commercial products. We can classify this diversity along a number of different axes, which impact on preservation requirements.

- **Diversity of application.** Software is used in almost every domain of human activity**.** Thus there are software products 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 and needs to classify 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 computers, clusters). In order to recreate the functionality of system, 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

[4] Other alternative terms are also used, including software *system* (which could be confused with a complete assemblages of different hardware software items),and software *package* (which in the context of preservation, could be confused with the OAIS notion of *information package*). We chose the term *product* as a neutral term, and does not imply that the software product is being provided on a commercial basis.

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.

- **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 products which provide particular set of library functions, such as the Xerces XML processor; major application products, 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 products 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 product 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 within attribution and licensing conditions.
- **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 products 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. Others 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 a general framework for software preservation can be defined which is applicable to a wide range of different software products.

2 Why Preserve Software?

A key question with respect to preservation of software is why it is a useful thing to do. 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: hardware, operating system, versions of systems (e.g. programming languages and compilers) and configuration change. When the environment changes, software notoriously stops working, crashes taking down vital pieces of data, or works but not as originally intended, with missing or differing functionality. The last case can be particularly damaging, as the software may seem to work but actually produces subtly different results. For example, compiling with a different floating point module may produce quite different results in the analysis. The complexity of the software makes it difficult to make the required adjustments so that is functions correctly in the new environment.

Software is *disposable* as in the face of environment change and the complexity of large-scale systems, developers often throw away 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 wrestle with legacy, 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.

Together, these make the preservation of software appear both difficult and unnecessary. However, there are also good reasons to preserve software, especially in a research and teaching environment. Some of these reasons are as follows.

Preserve 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 of the hypothesis of the research - if you can't implement it 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 test 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 products 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.

Preserving the data

Related to the previous point, is the reproduction and verification of research which has generated and analysed data, and published the results. In order to verify the asserted claims of a research project, then it should be reproducible. In many circumstances it may be 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, and they should be judged on the results as they saw 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 is expensive; other data which is recording specific events, such as environmental conditions at particular times and places, is non-reproducible. In these circumstances, it is desirable to preserve the data and reuse it in order to maximise its scientific potential, and to do this it is often necessary to preserve some supporting software to process the data format, and to provide the appropriate data analysis. This reason 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 keep the content accessible to future users.

Thus software also needs to be preserved 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 in itself, so it may be suitable not to ensure that that software is reproduced in its exact form, but only sufficiently well preserved to process the target data accurately. Thus we introduce the key notion of adequacy, to provide "good enough" preservation, with key properties of the software preserved and others disregarded.

Handling Legacy

Perhaps the prime motivation to preserve software for most organisations is to save effort in recoding. Legacy code 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 the maintenance of most existing software repositories, and a significant part of the effort which is undertaken by software developers both in-house within end-user organisations, 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, similar considerations to software preservation are within the best practice on

software maintenance and reuse, a long recognised part of good software engineering. If you can find an existing product or library routine, why bother rewriting it? Of course in these circumstances you need assurance that the software will run in your current environment and provide the correct functionality

Museums and Archives

A small but significant constituency of software preservation is those museums and archives which specialise on preserving aspects of the history of computing and its influence on the wider course of events. These institutions wish to preserve important software artefacts as they were 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 their development over time.

Such museums themselves often concentrate on preserving hardware. For example, Bletchley Park[5] and the National Museum of Computing[6] preserve or rebuild historic machines, including early code-breaking machines from WWII, as does the Science Museum[7], the Museum of Science and Industry in Manchester[8], and the Computer History Museum[9] in California. These machines are often kept in working operation, so there is a need to preserve the software to demonstrate the function of the machine.

Others archives are interested in preserving the software alone, typically via a web presence. Examples include: the Chilton Computing site[10], which includes the Atlas Basic Language Manual describing the software architecture of the Atlas computer from 1965; the Multics History Project[11], which preserves the code for the Multics operating system developed in the 1960s; and Bitsavers[12], which preserves documentation and software for minicomputers and mainframes from the 50's to the 80's.

For example, the Multics History Project is a effort to locate and engage the original experts on designing and using the system to capture their knowledge before they die. The project seeks to preserve the binary, to "preserve the bits", and document the formats. It also has an emphasis on capturing the implicit knowledge of the development organization and process, and creating a "map" of the software describing different views on it, via for example, capturing its source code, the coding interfaces, and its functions. It further seeks 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 for active use, and most of the functionality could be emulated elsewhere and the data generated using it processed on different operating systems. Nevertheless, Multics is an object lesson in software engineering (good and bad), and is undoubtedly valuable case study for future generations of computing engineers.

[5] http://www.bletchleypark.org.uk/
[6] http://www.tnmoc.org/index.htm
[7] http://www.sciencemuseum.org.uk
[8] http://www.msim.org.uk/
[9] http://www.computerhistory.org/
[10] http://www.chilton-computing.org.uk/
[11] http://www.multicians.org/mhp.html
[12] http://www.bitsavers.org

Other groups wish to preserve hardware and software as research interests or private enthusiasms, for example the Computer Conservation Society[13], a specialist interest group of the BCS, the Software Preservation Group[14] supported by the Computer History Museum, or a number of groups such as the Software Preservation Society[15] 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 (Zabolitsky, 2002). However, this is largely limited to preserving historic software as a unit with the historic hardware, so the major concern is preserving the storage of the code on some physical media, with appropriate backup and replication strategies; these preservation actions are similar to those for other digital artefacts. The problem of preserving the usage of the software in a future context is not considered in detail. (Computer History Museum 2006) gives an overview of approaches to software preservation being undertaken in museums and archives. Major concerns are how to collect important software products, especially with a variety of licensing constraints, and how to interpret and display them to the public.

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 virtual machine code, which can be emulated on more modern platforms.

3 The Principles of Software Preservation

Software preservation has four major aspects.

- **Storage**. A copy of a software "product" needs to be stored for long term preservation. Software is a complex digital object, with potentially a large number of components constituting a product (c.f. an *information package* as in OAIS); what is actually preserved is dependent on the software preservation approach taken. Whatever the exact items stored, there 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 product 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 product 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

[13] http://www.computerconservationsociety.org/index.htm
[14] http://www.softwarepreservation.org/
[15] http://www.softpres.org/

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.

4 Software Preservation Approaches

Various approaches to digital preservation have been proposed and implemented, usually as applied to data and documents. However, they do usually refer 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 (Cedars, 2002) defines three main strategies, which we give here, and consider how they are applicable to software.

- **Technical Preservation. (techno-centric).** Maintaining the original software (typically a *binary*), and sometimes hardware, of the original operating environment. Thus this is similar to the use case for software preservation arising from museums and archives where the original computing hardware is also preserved and as much of the original environment 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.

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 accuracy of the preservation.

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 becomes 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 (e.g. 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. Nevertheless, emulation is being applied in several projects, notably within the European project PLANETS[16].

The migration approach does not seek to preserve all the properties of the original, or at least not exactly, but as observed in the European project CASPAR[17], only those up to the interface definition, 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

[16] http://www.planets-project.eu/
[17] http://www.casparpreserves.eu/

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 key but there is a tolerance to change of other characteristics.

These three different preservation strategies thus require different levels of detail of preservation information for 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.

5 Performance Model and Adequacy

Closely related to the preservation approach is the notion of how a sufficient level of *performance* adequately preserves the required characteristics of software. Performance as a model for the preservation of digital objects was defined by the National Archives of Australia in (Heslop et. al. 2002) 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 1. 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 considered 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.

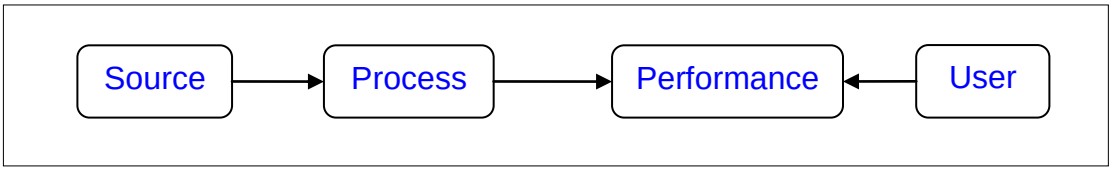

Figure 1: NAA Performance Model

The notion of performance has been developed in the context of traditional archival records and has been adopted in studies into the significant properties of different media types (see [5], [2]) which compare 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 itself.

In the case of software, the performance is the execution of binary files on some hardware platform configured in some architecture to provide the end experience for the user. However,

the processing stage depends on the nature of the software artefacts preserved which have differing reconstruction and replay requirements. This is illustrated in Figure 2.

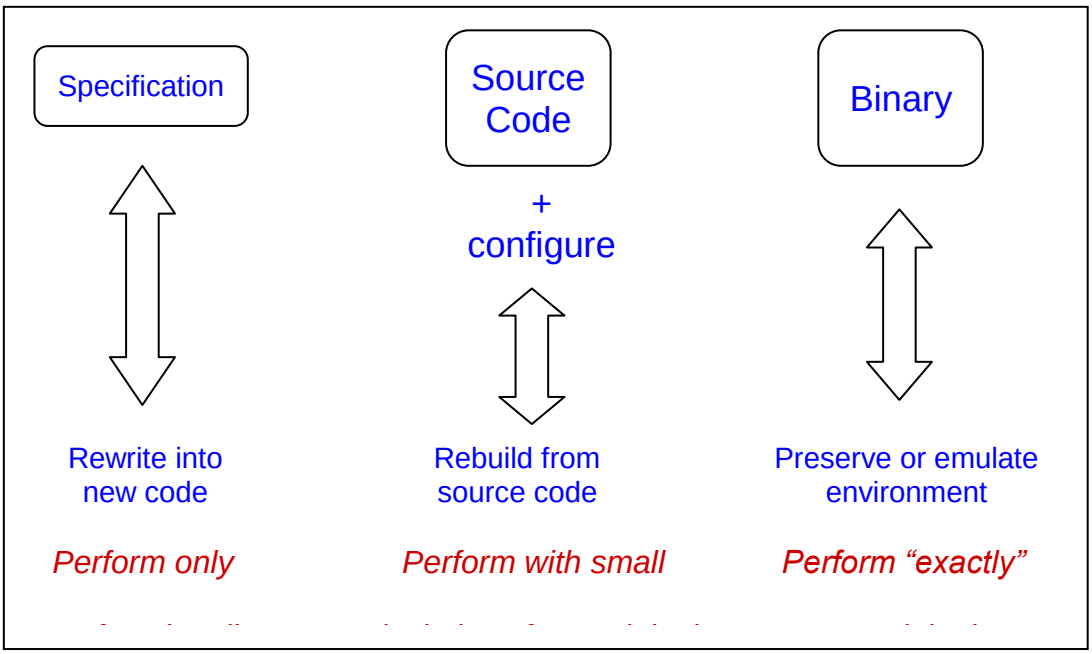

Figure 2: Software Performance models from different sources

- In the case where binary is preserved, the process of generating 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 products 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.

A 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, we define the notion of ***Adequacy***.

> *A software product (or indeed any digital object) can be said to perform adequately relative to a particular set of features ("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*.

Admittedly, this notion of adequacy is usually viewed as an aspect of 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). However, we feel that it is useful to separete these two notions in order to establish a more lucid requirement specification of long-term preservation of software. For this, we use the premise that the term ***Authenticity*** in long-term preservation essentially signifies the level of **trust** between a preserved software product and its future end users. From the preserpective of an end user of a software product, this trust is primarily associated with the ability to trace the provenance (e.g. history of origin, custodianship etc.) and verify the fixity information (e.g. checksum) of the software. For example, a preserved software with comprehensively documented provenance history and verifiable fixity information might establish a sense of trust for the body responsible for its preservation in its users. But this "trusted preservation" does not gurantee a reliable behaviour from the software once reconstructed in future; it might incur a loss of some of its original features during its reconstruction process. However, the software could still be used for the remaining features retained after reconstruction, which could be sufficient to extract an acceptable level of performance from the software. An example of such a software is the emulated version of the 1990's DOS-based computer game Prince of Persia[18]. While some of the instructions do not always work on the emulator and the original appearance of the game is also somewhat lost, it is possible to run the emulator to play the complete game on a contemporary computer platform. The term ***Adequacy*** introduced here is intended to represent this particular concept. In effect, by measuring the adequacy of the performance, we can thus determine how well the software has been preserved and replayed.

5.1 Performance of software and of data

A further aspect of the performance model for software is that the measure of adequacy of the software is closely related to the performance of its input *data*. The purpose of software is (usually) to process data, so the *performance* of a software product becomes the *processing* of its input data. This relationship is illustrated in Figure 3. Note that we have reversed the arrow between performance and user to reflect the information flow. Further, there is an interaction between the user and the software performance, reflecting the user's interaction with the software product during execution, changing the data processing and thus the data performance.

[18] http://www.bestoldgames.net/eng/old-games/prince-of-persia.php

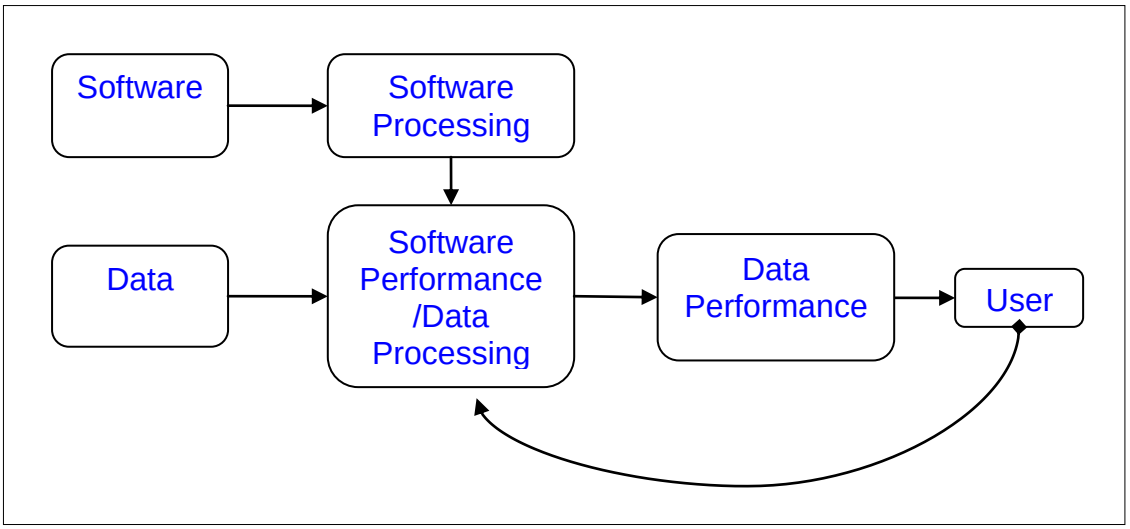

Figure 3: Performance model of software and its input data

So for example, in the case of a word processing product which is preserved in a binary format, which is processed via operating system emulation, the performance of the product 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. The user can then interact with the processing (via for example entering, reformatting or deleting text) to change the data performance. Thus 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 thus how well it preserves the significant properties of its input data, and also perserving a known change in the data performance which results from user interaction with the processing.

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 need to be preserved, as in Figure 4. In this case, the performance of software is the processing of the application binaries or source code, which in turn are measured by its adequacy in processing its intended input data.

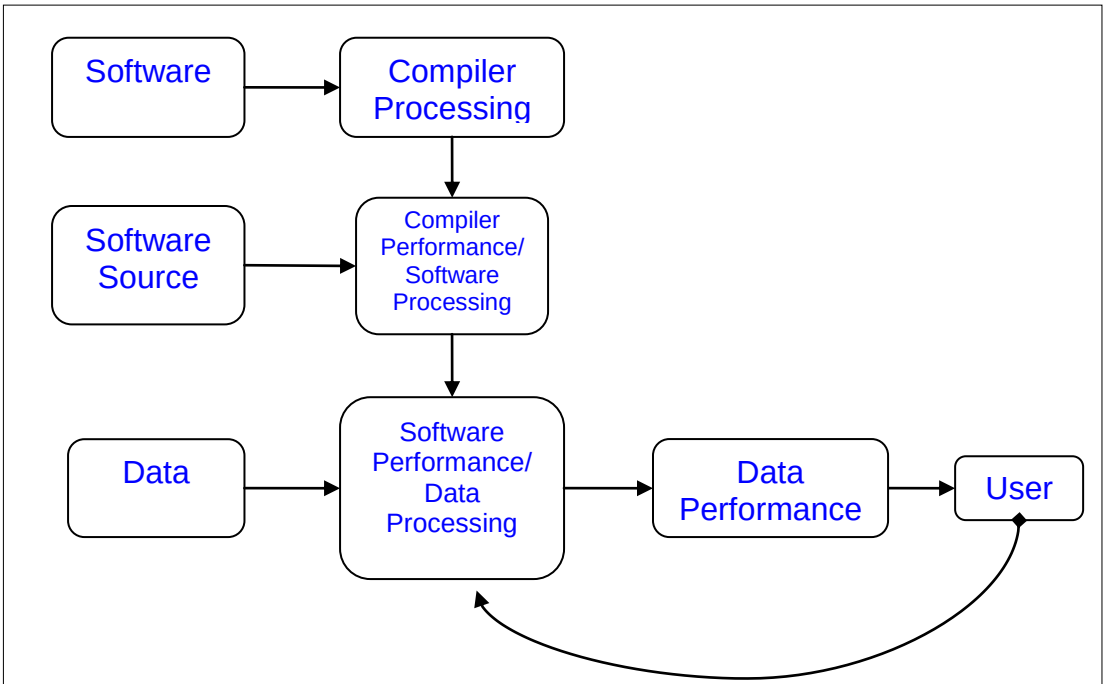

Figure 4: Performance model of software compilers

Thus the adequacy of different preservation approaches is dependent upon the performance of the end result on the end use on *data*. As the software has to be able to produce an adequate performance for any valid input date, the adequacy can be established by performing trial executions against representative test data covering the range of required behaviour (including error conditions). This is further illustrated in the table below that outlines some probable adequacy determining factors for a number of different types of software product:

Software Category	"Adequacy" Factor(s)	Examples
Scientific Data Processing Software	The adequacy of the behaviour of this type of software after it has been reconstructed, may be measured by: • Running the software to process some pre-specified test input data • Comparing the output of the test run with the corresponding pre-specified test result; • Checking if the output exceeds the acceptable level of error tolerance for the software.	Starlink, NAG Software Library. For example, the NAG library publishes test cases and a specification of the required accuracy, in terms of number of significan figures for its mathematical software routines. Such accuracy is

		highly dependent on the mathematical libraries and coprocessor used.
Games	The adequacy of the behaviour of a reconstructed game may be measured by: • Comparing its UI with the screen capture of its original UI. • Comparing its performance against some pre-defined use cases. For example, the completion time of a paritcular level can be compared against the average completion time for that level in the original game.	The 1990's DOS-based version of Prince of Persia
Programming Language Compilers	A compiler may be said to have been preserved adequately, if, after reconstruction: • it covers all features of the programming language that it supports, e.g. concurrency(i.e. threads), polymorphism, etc. For some programming languages (e.g. Fortran, C, C++ etc.), there exist ISO standards[19] which describe the correct behaviour of a software written in these languages. These standards also provide test programs that may be used to assess the adequacy of a compiler for rendering all features of the programming language that it supports. • the application resulting from compiling its source code (written in a language supported by the compiler) using the compiler yields the expected behaviour.	Java Compiler, C compiler
Word Processor	The adequacy of a word processor may be	OpenOffice Word[20],

[19] http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_tc_browse.htm?commid=45202
[20] http://www.openoffice.org/

	measured based on its ability to: • render existing supported word documents with an acceptable level of error tolerance. For example, a word processor may be regarded as adequate as long as it clearly displays the contents (e.g. test, diagram, etc.) of a word document, even if some of the features of the document content, such as font colour and size, may have been rendered incorrectly or even lost completely. • enable editing (e.g. add/change/remove text, change font) and saving existing word documents • enable creation and saving of new word documents	Corel WordPerfect[21]

Thus, the adequacy of preservation of a particular significant property can be established by testing against pre-specified suites of test cases with the expected behaviour, and pre-specified user interactions to change the data performance in known ways.

6 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.

6.1 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

[21] http://www.corel.com/servlet/Satellite/us/en/Product/1207939618939#tabview=tab0

[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.

6.1.1 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 5.

This model has four major conceptual entities, which together describe a complete ***Software System***. These are *Product, Version, Variant* and *Instance.* 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.

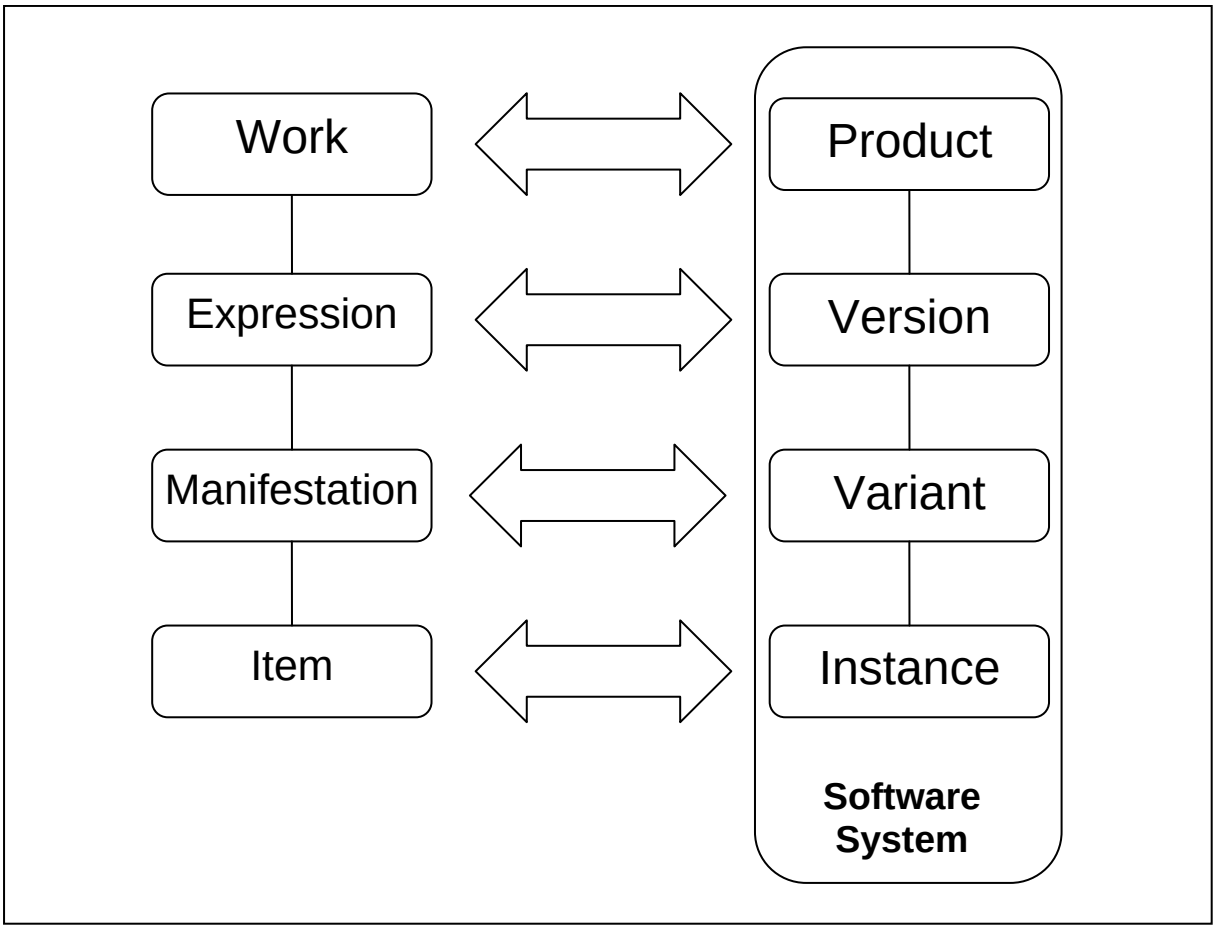

Figure 5: Conceptual model for Software and relationship to FRBR

Product. The product[22] is the whole top-level conceptual entity of the system, and is how the system may be commonly or informally referred to. Products 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-products, with independent provenances (e.g. Linux). Thus examples would be "Windows", "Word", "Starlink", "Xerces". Products themselves can be composite objects (a software library or framework) and may have a number of other sub-products within them.

Products are characterised by the following main features.

- A product 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;

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

    - "Gram–Schmidt orthogonalisation of n vectors of order m" for function identifier F05AAF in the NAG library. This function can be regarded as a product in its own right, but it also a sub-product of the whole NAG library (which is also a product).
- A product 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-products within the product. Software often changes ownership, but then it should change as a product - 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 product 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 product, 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) product entirely, although maintaining many of the same components and sub-products.

Version. A version of a software product is expression of the product which provides a single coherent presentation of the product 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 products, the sub-products will themselves have a number of versions which will be related to versions of the complete product. 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 product 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.*

Instance. An actual physical instance of a software product which is to be found on a particular machine is known as an *Instance*. It may be also referred to as an installation, although there is no necessity for the product 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 instance and user.
- Usage of particular hardware and peripheral devices as appropriate.
- It may also be necessary to record a MAC or URLs etc identifying particular locations or machines to which the licence for a particular software instance is bound.

6.1.2 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-products. All systems however, will be constructed out of many individual *components*[23]. A component is a storable unit of software

[23] 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 *Instances* in our model. We make the distinction to handle the inherently composite nature 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 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.

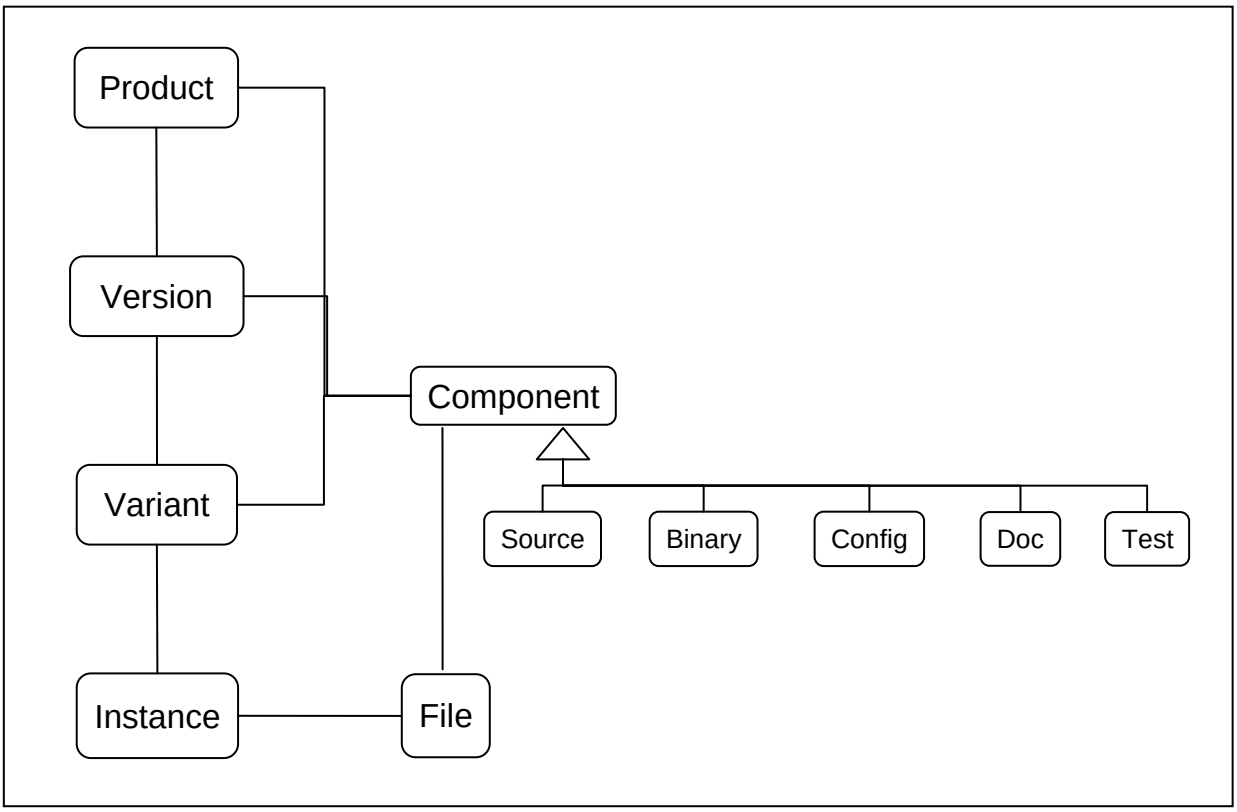

Figure 6: The Software Component Conceptual Model

Software components are thus associated with a product, version or variant in the conceptual model of software as in Figure 6.

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 product, 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 product and expected behaviour arising from operation of the product. Produced to test the conformance of the product 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 product/version/variant and as noted the significant properties of a component may be of a different digital object type.

7 The OAIS Reference Model and Software Preservation

The Reference Model for an Open Archival Information System (OAIS) is an ISO standard that is primarily concerned with the long-term preservation of digitally encoded information. In essence, the underlying notions of the OAIS reference model should be applicable to the long-

term preservation of software artefacts as fundamentally (i.e. at bit level) they are in fact digitally encoded information. This is further analysed in this section.

7.1 The OAIS Information Model and the Preservation of Software

The OAIS reference model describes a number of conceptual models in order to aid formulation of a suitable preservation strategy for a digital object. In terms of the relevance to the framework for software preservation (Section 6), the most important of these models is the Information Model that broadly describes the metadata requirements associated with retaining a digital object over the long-term.

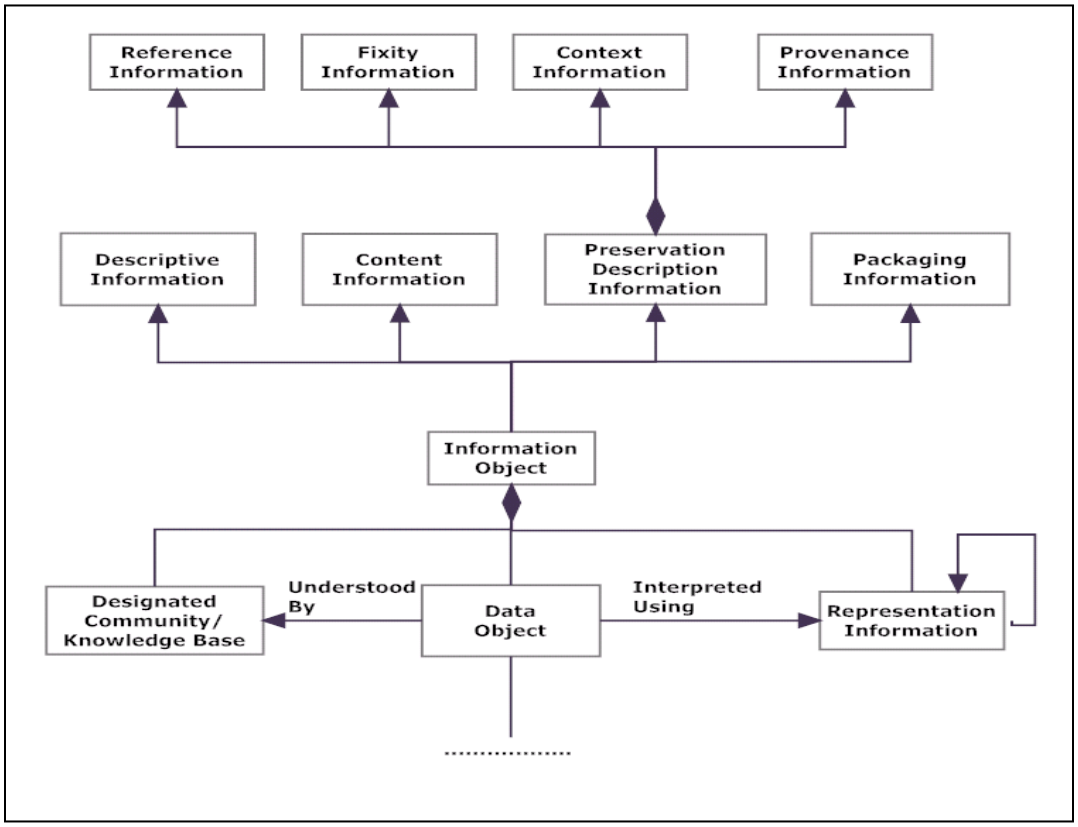

Figure 7: The OAIS Information Model

As illustrated in Figure 7, the information model embedded in the OAIS framework consists of a number of components. Here we consider the underlying notions of these components in the context of long-term preservation of software.

7.1.1 Content Information

This is essentially the digital object that needs to be preserved over the long-term. In the case of software preservation, it should be a copy of the most recent version of the software, which at the most fundamental level is a sequence of bits, i.e. a digital object. In terms of the conceptual model for software components presented earlier in the report (section 6.1) this may be compared to an *Instance* of a software product.

7.1.2 Preservation Description Information (PDI)

This is a set of information that is needed to efficiently manage and preserve the software product with which it is associated, over an indefinite period of time. In order to ensure effective preservation of a software product, the OAIS reference model identifies four different types of PDI to be recorded and preserved along with the software product to which they correspond:

- ***Reference Information***: This information enumerates identifiers assigned to a software product such that it can be referred to unambiguously, both internally and externally to the preservation archive. If necessary, this may also be used to describe mechanisms used to assign unique identifiers to a software product. Examples of unique and persistent identification schemas for software include ARK[24], DOI[25], etc.
- ***Provenance Information***: This is intended to record information needed to sufficiently trace and verify the history of a preserved software product. The provenance information about a software product may include its vendor information, chain of custody, preservation actions and effects and so on. This notion is also captured in the software preservation framework that identifies the history of changes of ownership of a software product and any other changes that it has undergone during its lifecycle as its significant properties (Section 6.1.2, 8.1).
- ***Context Information***: This documents the relationship(s) of a software product under preservation with other digital objects in the same and/or other preservation archive(s). This could be a data object that the software is used to render and/or another software product that the software interacts with to aid rendering a data object. For example, the context information of a NetCDF[26] reader might be a NetCDF file or a more complex software suite for, say, producing Plume[27] imagery that uses the NetCDF reader for reading a NetCDF file into memory.
- ***Fixity Information***: This describes the mechanisms (e.g. checksum, digital signature etc.) used to verify that the software product has not been subjected to any unauthorised or undocumented modification(s).

[24] Archival Resource Key - http://www.cdlib.org/inside/diglib/ark/
[25] The Digital Object Identifier System - http://www.doi.org/
[26] http://www.unidata.ucar.edu/software/netcdf/
[27] http://www.mantleplumes.org/PlumeDLA.html

In view of future re-use of a preserved digital object, the PDI in the OAIS framework is specifically intended to aid futures users in verifying the authenticity of the digital object. Although the original OAIS model does not deal with authenticity very thoroughly but the working draft revision[28] of the model defines ***Authenticity*** as:

> *"the degree to which a person (or system) may regard an object as what it is purported to be. The degree of Authenticity is judged on the basis of evidence".*

The key term in the aforementioned OAIS definition of authenticity is "evidence". And as underlined earlier in the report (section 5), the authenticity of a software product has two central aspects: "Trust" (i.e. trusted preservation) and "Adequacy" (i.e. reliability of behaviour in future). In terms of the OAIS model, the evidence on which the judgment of trusted preservation i.e. if a software product has been preserved by a trusted preservation body and has not been altered in an unauthorised manner, may consist of the provenance and fixity information of the software. Verification of the adequacy of a preserved software product in future, on the other hand, may require further information, which is not specified (at least not in direct terms) in the OAIS model. At minimum, the information required to verify the reliability of the functions of a preserved software product may consist of some pre-defined test routines and their expected results. For example, this information for a software product used for converting a NetCDF file to a GML[29] format would be an example NetCDF file and its corresponding GML file.

As discussed earlier in this report, this type of "Adequacy" related information for a software product may be considered amongst the ***Preservation Description Information*** of software for demonstrating the satisfaction of significant properties, and thus viewed as an additional component of the OAIS information object in the context of long-term software preservation. The notion of "Test Suite" in the conceptual model for software components (section 6.1.2) is intended to provide evidence of adequacy of the behaviour of software in future.

7.1.3 Descriptive Information

The information needed to facilitate efficient discovery and access to a preserved software product, typically through search and retrieval facility provided by the long-term preservation archive. Descriptive information about a software product may be derived from its PDI and significant properties. This information for a software product may include its name, a brief description of its features and so on. The software preservation framework also defines a number of preservation properties of software to capture descriptive information of a software product, such as its functional purpose (Section 6.1.2).

7.1.4 Representation Information (RI)

This is intended to facilitate proper rendering, understanding and interpretation of a preserved digital object on in future. In terms of software preservation, this is equivalent to the information required to understand the reconstruction process of a preserved software product and reconstruct it on a future technological platform. This representation information can be recursive until there

[28] Based on personal communication with Dr. David Giaretta, one of the editors of the OAIS Reference Model.
[29] Geography Markup Language - http://en.wikipedia.org/wiki/Geography_Markup_Language

is sufficient information available to rebuild the software. The OAIS framework identifies two different categories of representation information:

- ***Structure Information***: For a software product, this may include the name of compiler used to build it from source code, the name of the virtual machine, configuration information, installation instructions, dependent library, and programming language. This type of structural representation information for a software product is also identified within the software preservation framework as preservation properties of software, such as software environment and software architecture (Section 8.1).
- ***Semantic Information***: The additional information required to properly understand the intended meaning and purpose of technical representation information. For example, a Linux-based software product might require the name of the operating system, i.e. Linux included in its representation information. However, it might also be necessary to include detailed information about Linux, e.g. description, installation instructions, tutorial, etc. to ensure proper understanding of the term "Linux" in future. It should be noted that the level of granularity of semantic representation information should depend on the knowledge base of the designated community associated with the software. This is discussed in more detail later. The notion of semantic representation information is not addressed directly in the software preservation framework presented in this report. However, the framework does provide scope for recording reference(s) to further information about a preservation property. Therefore, the value of the preservation property, *operating system* for the aforementioned Linux-based software could be *Linux* and an URL to the Linux website or, in the case where a copy of Linux operating system is also being preserved (either in the same or different archive), a reference to its preserved version which would have its own (structural and semantic) representation information.

7.1.5 Packaging Information

The information that is used to bind the software product and its associated metadata, such as PDI and Descriptive Information into an identifiable unit or package for preservation. For example, if a software instance is compressed before being ingested into the archive, the packaging information for this software would be information about the underlying structure of its compressed form.

7.1.6 Designated Community

This encompasses all identified potential consumers (e.g. human, software application etc.) to whom the preserved software product is beneficial in terms of accurate interpretation and proper utilisation of the software. The level of recursion for a particular element of representation information about a software product is likely to depend on the level of knowledge that the designated community has about that element. For example, if the designate community has considerable understanding of Java, then the representation information of Java-based software

need only include the name of the programming language, i.e. Java. Conversely, if the designated community has no understanding of Java, then the representation information of such a software product might have to include detailed information about Java Virtual Machine (JVM) needed to compile and run the program as well as other Java-related information. Ideally, the task of defining an appropriate designated community for a digital object should be a part of the overall preservation strategy. Therefore, it is not considered within the remit of the framework for software preservation.

7.2 The OAIS Information Model and Software Performance

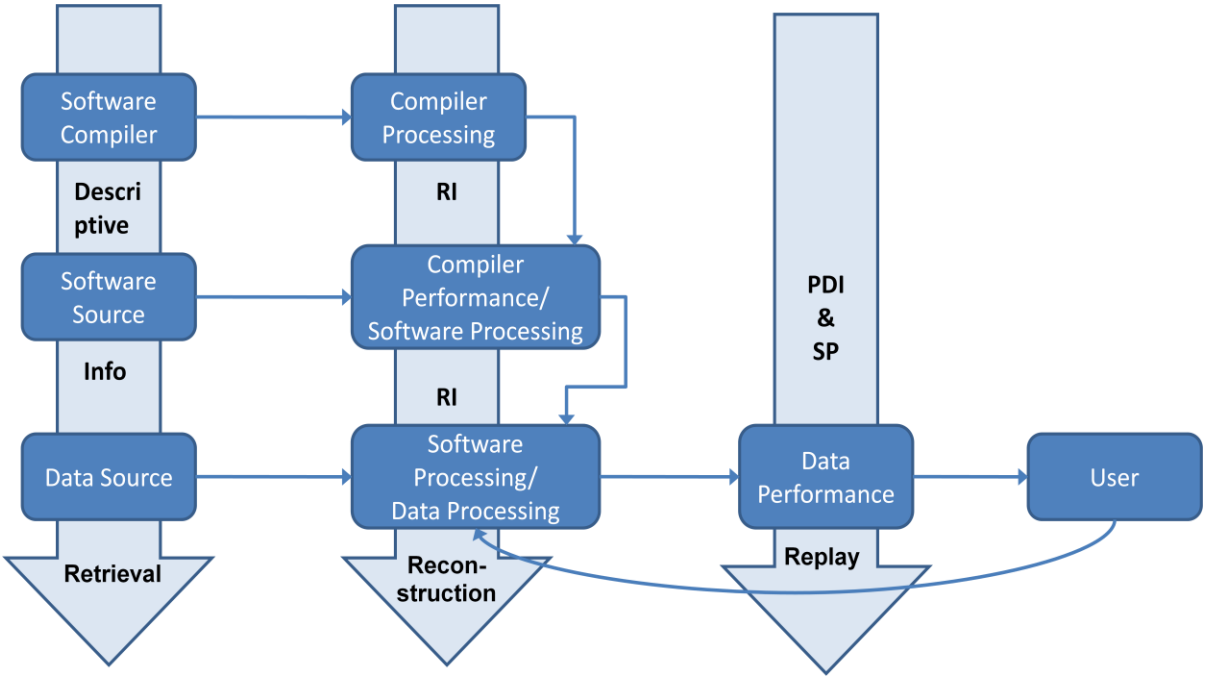

Figure 8: The Relationship between the OAIS Information Model and the Software Performance Model

Considering the relationship of the OAIS information model with the Software Preservation Framework and its applicability to long-term software preservation, it is possible to apply the model to the conceptual model of performance software discussed in Section 5.1. As illustrated in Figure 8 and outlined below, the OAIS information model can be applied to the process for rendering a preserved Data Source on a future technological platform, where the rendering of the data requires the use of a particular software, which in turn requires a specific complier, to be rebuilt from its preserved state:

- ***Descriptive Information*** about the Data Source is used to locate it in the archive.
- ***Representation Information(RI)*** about the Data Source is used to determine the name of the software required to process it.
- ***Descriptive Infromation*** about the sofware record in the RI of the DataSource is used to locate the source code of the software product in the archive.
- ***Representation Information(RI)*** of the software is used to determine the name of compiler required to re-build the software. It may also be necessary to use the Packaging Information of the software for its reconstruction, especially if it needs to be reconstructed in the form in which it was originally ingested into the preservation archive. For example, if the software was compressed (i.e. zipped) before being ingested into the preservation archive, verification of its fixity information after reconstruction would need to be done against its original compressed form in order to ensure accuracy.

Therefore, the reconstruction of software in future would need to retain its original comrpessed form.

- ***Descriptive Information*** about the compiler is used to locate and access it in the archive or elsewhere.
- ***Representation Information(RI)*** of the software is used to determine the instructions for re-building the software from source code using the compiler and subsequently re-build it.
- ***Preservation Description Information (PDI)***, such as provenance and fixity information of the software, is used to verify the integrity of the re-constructed instance of the software.
- ***Representation Information(RI)*** of the Data Source is used to process it using the re-built software
- ***Significant Properties(SP)*** is used to measure the adequacy of the software in processing the Data Source, which in turn measures the performance of the compiler in re-building the software from its source code.

8. Preservation Properties of Software

8.1 Categories of Properties

In considering what preservation properties are needed for 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. In terms of the OAIS information model, software functionality related information may be regarded as ***Descriptive Information*** (Section 7.1.3) of the software that can aid efficient discovery and accessibility of the software in future.
- **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. In the context of long-term software preservation, this type of information should be useful for rebuilding and reusing the software in future. Therefore, it serves the same purpose as that of the ***Representation Information*** in the OAIS information model (Section 7.1.4).

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. In addition, detailed history of significant changes that a software product has undergone facilitates verification of its authenticity in future. In view of the OAIS information model, software version history is a type of the ***Preservation Description Information (PDI)*** (Section 7.1.2).

- **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. In the OAIS information model, this directly corresponds to the ***Provenance Information*** category of ***Preservation Description Information (PDI)***
- **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. Of particular note here is that sufficient documentation about the intended user interaction with a software product should also contribute towards the assessment of adequacy of the functionality of the software in future.
- **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 products, 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. In essence, this information about the environment in which a software product operates, may be categorised as the ***Representation Information*** of the software.
- **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. In common with the software environment related information, the information about the underlying architecture of a software product may also be viewed as a part of the ***Representation Information*** of the software.
- **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. Therefore, this contributes towards the information needed to measure the overall adequacy of software preservation in future (Section 5.1 and 7.1.2). 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.

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. We also try to demonstrate the relationship of each of these properties to the relevant OAIS information entity. Note that as specified earlier, we do not give details on the significant properties of the user interaction.

8.1.1 Product Properties

Products 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 product. We would also expect a general classification of the system within a controlled vocabulary to refer to a product. The following properties are associated with a Product.

Property Category	**Software Property**		**Equivalent OAIS Terminology**
Functionality	**Purpose**	Description of overall functionality of software system	**Descriptive Information**
	Keyword	Classification of software under a specified controlled vocabulary	**Descriptive Information**
Provenance and Ownership	**product_name**	Name of the product	**Descriptive Information**
	Owner	Owner of the product, with contact details	**Provenance Information**
	Licence	Overall licensing agreement	**Provenance Information**
	Location	URL of website of software	**Reference Information**
Software Architecture	**Overview**	Overview of software architecture	**Descriptive Information**
Software Composition	**software overview**	Documentation on the overview of the software	**Descriptive Information**

	Tutorials	Teaching material on the system.	**Representation Information**
	requirements	requirements of product	**Representation Information**

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**		**Equivalent OAIS Terminology**
Functionality	**functional_description**	Description of relationship of between inputs and outputs of the version.	**Descriptive Information**
	release_notes	Description of changes of this version from other versions.	**Provenance Information**
	algorithm	Description of the algorithm used.	**Representation Information**
	input_parameter	Details of names and formats of inputs	**Representation Information**
	output_parameter	Details of names and formats of outputs	**Representation Information**
	interface	API description	**Representation Information**
	error_handling	Description of how errors are handled.	**Representation Information**
Provenance and Ownership	**version_identifier**	Identifier for this particular version	**Reference Information**
	licence	Licence specific to this version.	**Provenance Information**
Software Environment	**programming_language**	Programming language used for this version.	**Representation Information**
	hardware_device	Category of hardware device	**Representation Information**

		which the software version depends upon.	
Software Architecture	**detailed_architecture**	Detailed description of architectural dependencies of the version.	**Representation Information**
	dependent_product	Dependency on another software product being installed.	**Representation Information**
Software Composition	**source**	Source code modules for this version.	**Representation Information**
	manual	Usage instructions for this version	**Representation Information**
	installation	Installation, build and configuration instructions for this version.	**Representation Information**
	test_cases	Test suite for this version.	**Significant Properties (Not addressed in the OAIS Model)**
	specification	Specification of this version	**Representation Information**

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**		**Equivalent OAIS Terminology**
Functionality	**variant_notes**	Description of the variations in behaviour specific to this variant.	**Descriptive Information**
Provenance and Ownership	**Licence**	Licence specific to this variant.	**Provenance Information**
Software	**Platform**	Target hardware	**Representation**

Environment		machine architecture of version.	**Information**
	operating_system	Version of operating system	**Representation Information**
	Compiler	Version of compiler used to construct this variant.	**Representation Information**
	dependent_library	Version of dependent software libraries used.	**Representation Information**
	hardware_device	Specific auxiliary hardware devices supported by the variant.	**Representation Information**
Software Architecture	**dependent_product**	Dependency on another software product being installed.	**Representation Information**
Operating Performance	**processor_performance**	A specification that a specific speed of processor is required.	**Significant Property (Not addressed in the OAIS Model)**
	memory_usage	Minimal/typical memory usage for RAM and disk of the variant.	**Significant Property (Not addressed in the OAIS Model)**
	peripheral_performance	Performance of specific peripheral hardware, for example screen or colour resolution, audio range.	**Significant Property (Not addressed in the OAIS Model)**
Software Composition	**Binary**	Machine executable code for this version.	**Representation Information**
	Source	variants of source modules for this version	**Representation Information**
	Configuration	installation and configuration instructions for this variant	**Representation Information**

8.1.4 Instance Properties

An instance of a software product 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 instance:

Property Category	Software Property		Equivalent OAIS Terminology
Provenance and Ownership	**Licensee**	Named licensee of the instance	**Provenance Information**
	Conditions	Local conditions of use of this instance.	**Representation Information**
	licence_code	Licence key value	**Representation Information**
Software Environment	**environment_variable**	Specific settings for environmental variables.	**Representation Information**
	hardware_address	Specific MAC address (or equivalent) identifying a specific machine.	**Representation Information**
Software Composition	**File**	Names and addresses of specific files in the instance.	**Representation Information**

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. Note the components properties may be viewed as a subset of the version properties and therefore have the same associations with the OAIS information model as those of the version properties.

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_product	Dependency on another software product being installed.

In our above analysis of different categories of preservation properties, we also underline a considerable commonality between the framework for software preservation and the information model embedded within the OAIS reference model. In essence, the software preservation framework attempts to articulate the OAIS information model for the specific task of long-term software preservation. In doing so, it introduces the notion of "***Adequacy***" of software behaviour in order to facilitate assessment of the efficiency of preservation in future; an area that is not comprehensively addressed in the OAIS reference model. The framework also identifies a number of properties of software, termed as ***Significant Properties*** against which the adequacy of software behaviour may be measured in future. The underlying notions of these significant properties of software are also not captured within the OAIS information model. Therefore, the conceptual framework for software preservation presented in this report may be regarded as a specialisation of the OAIS Information Model for the long-term preservation of software artefacts.

9 Conclusions

In this report we have developed a conceptual framework to express a rigorous approach to software preservation. This is a development on the approach given in (Matthews et. al. 2008) as it: develops and extends the notion of performance and emphasises the notion of adequacy and relates it to authenticity; narrows the notion of significant property to those properties which are testable within a performance; and considers the concepts introduced within the OAIS model, and uses them within the framework to categorise the preservation properties identified within the model. Thus this framework can be seen as a specialisation of the OAIS model to handle the case of software preservation.

We believe that this is a general and principled approach which can cover the preservation needs of a wide range of different software products, including modern distributed systems and service oriented architectures, which are typically built of pre-existing frameworks (e.g. Eclipse) and have a large number of dependencies on a widely distributed network of services, many of which are outside the control of the typical user (e.g. DNS services, proxies, web services provided by external organisations (e.g. Amazon Web Services).

We also believe that the performance model presented here, which has a notion of user feedback to influence the performance represents an approach to preserving the user interface and the user interaction model, although work is required to further develop that notion.

Further work is required to test this model and to provide tooling. Within the JISC sponsored Tools and Guidelines for Preserving and Accessing Software Research Outputs, some initial tooling work has been undertaken to integrate the capture of preservation properties of software within a software development process, and also to use the framework within case studies. Further work on case studies is required, especially across a range of software types to cover the diversity of software considered above, and to consider how to support the preservation of legacy software, both methodologically, and also using suitable tools, including tools which have been developed to support OAIS (such as those developed within the CASPAR project), including tools to express representation information, and assess the authenticity of a preservation package.

Acknowledgements

We would like to thank our colleagues David Giaretta, Esther Conway, Steven Rankin and other members of the Digital Curation Centre and CASPAR projects for their advice and discussions. The work was carried out under the JISC study into the Significant Properties of Software and also the JISC sponsored Tools and Guidelines for Preserving and Accessing Software Research Outputs.

References

The Cedars Project. (2002). *The Cedars Guide to Digital Preservation Strategies*. Retrieved July 29, 2008, from http://www.leeds.ac.uk/cedars/guideto/dpstrategies/dpstrategies.html (2002)

Computer History Museum. (2006). *The Attic & the Parlor: A Workshop on Software Collection, Preservation & Access*, proceedings May 5, 2006. Retrieved July 29, 2008, from http://www.softwarepreservation.org/workshop/

Heslop, H., Davis, S., Wilson, A. (2002). *An Approach to the Preservation of Digital Records*, National Archives of Australia, 2002. Retrieved July 29, 2008, from http://www.naa.gov.au/Images/An-approach-Green-Paper_tcm2-888.pdf

Matthews, B.M., McIlwrath, B., Giaretta, D., Conway, E. (2008). *The Significant Properties of Software: A Study*. JISC, draft report, 2008. Retrieved July 29, 2008, from http://sigsoft.dcc.rl.ac.uk/twiki/pub/Main/SigSoftTalks/SignificantPropertiesofSoftware.doc

Zabolitzky, J.G. (2002). Preserving Software: Why and How. *Iterations: An Interdisciplinary Journal of Software History, 1*. Retrieved July 29, 2008, from http://www.cbi.umn.edu/iterations/zabolitzky.html