View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UCT Computer Science Research Document Archive

A Flexible Approach to Web Component Packaging

HUSSEIN SULEMAN AND SIYABONGA MHLONGO
University of Cape Town

Web-based applications are rapidly making the transition to service oriented architectures, but this breaking up of single monolithic applications into multiple cooperating components creates complexity and management problems for system designers. This paper discusses a solution to one of the emerging complexity-related problems: how to package and distribute applications based on Web components for rapid and flexible deployment. To this end, a packaging framework was designed and reference implementations of package building and package installation tools were developed. The usability of such tools was then tested by a sample of users in the context of online information management systems, because of the availability of Web components and the inherent network-aware nature of such systems. The results are promising and demonstrate that the approach taken is reasonable and understandable and that such supporting technology for systems based on Web components is likely to positively impact the growth of advanced Web applications. In addition, the process of developing a proof-of-concept packaging system and the feedback from users has provided a useful set of requirements and guidelines for building such systems in future.

Categories and Subject Descriptors: H3.5 [**Online Information Services**]: Web-based services; H3.7 [**Digital Libraries**]: Systems issues; D2.9[**Management**]: Software configuration management
General Terms: Management, Design, Standardization, Languages
Additional Key Words and Phrases: Component, package, dependency, installation, automation

---

1. INTRODUCTION

Complex Web applications were historically deployed by systems administrators but, as they become more commonplace, the responsibilities have shifted to users who are not as technically trained. For example, many librarians working with digital collections software, such as DSpace [Tansley, et al., 2003], need to manage their own software installations, with only occasional support from IT staff. This is especially true in developing countries where highly specialised staff are not always accessible. In these environments it is necessary for software installation to be as simple as possible, thus being a technology enabler for a wider audience.

Alas, while Web applications are becoming more common, there is also a greater tendency to build applications as compositions of cooperating components, spurred on by the popularisation of service oriented architectures. This negatively impacts the ability of users to install Web applications, but the componentisation is necessary to support flexibility and configurability of modern software tools. One approach to address this potential conflict in requirements is to introduce developer tools to manage this complexity so there is no visible difference between the component-based approach and monolithic approaches to software deployment [Suleman, et al., 2005]. In keeping with this theme, this research investigated the effects of introducing a packaging solution for Web-based software systems, paying particular attention to the processes of encapsulating a Web-based software system as a single installable package and configuring and deploying the resultant package.

The remainder of this paper contextualises this work in the field of digital libraries, a study domain commonly associated with delivering Web-based services relating to information management to end-users. It then looks into current practices in Software Engineering aligned with Web-based applications. Subsequent sections introduce the design of a proof-of-concept solution, report on the evaluation thereof and highlight some of the work related to this research. The concluding section provides some views on the success of this research and highlights work that remains to be done in the near future.

2. BACKGROUND

Web-based applications vary according to audiences to whom services are offered and with the latest technology practices; there are but a few limitations on the nature of services which can be made available over the Web. In the Digital Library (DL) arena, popular examples include the ACM Portal (http://portal.acm.org/), CiteSeer (http://citeseer.ist.psu.edu/) and Wikipedia (http://en.wikipedia.org/), all of which appeal to academic communities and the general public. If these are considered typical digital library projects, digital libraries can be thought of as a study domain that addresses issues concerning information management, especially in a scholarly context.

Author Addresses:
H. Suleman, Department of Computer Science, University of Cape Town, Private Bag X3, Rondebosch, 7701, South Africa; hussein@cs.uct.ac.za.
S. Mhlongo, Department of Computer Science, University of Cape Town, Private Bag X3, Rondebosch, 7701, South Africa; mhlongo@cs.uct.ac.za.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, that the copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than SAICSIT or the ACM must be honoured. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
© 2006 SAICSIT

At the core of a digital library is an organised and managed information collection. Such management includes preserving, disseminating and retrieving this information and is made possible by one or more digital library tools. While most of the early systems were based on custom-built software, many current systems rely on pre-packaged tools such as Greenstone [Witten and Bainbridge, 2002], DSpace [Tansley, et al., 2003] and EPrints [University of Southampton, 2006], all of which attempt to mimic traditional libraries in that they have been designed to house and manage any type of digital content.

One of the prime disadvantages of the digital library systems mentioned above is that they are all relatively monolithic by nature of their design and follow the same trend as programming languages which first emerged as planar and, as need arose, adopted a more modular approach. The same is happening to the digital library world. Modular- or component-based approaches to building digital libraries have slowly begun to emerge over the past few years.

According to Bainbridge, et al. [2004], digital libraries need to be dynamic. They (Bainbridge, et al.) support this by emphasising the need for administrators to routinely add new collections, or new user interfaces or completely new kinds of services, to a digital library at runtime, that is, without bringing it to a complete halt. This has seen the realisation of the recent Greenstone 3 project, now an agent-based (component-based) digital library system, something that its predecessors were not. Other similar views were initially witnessed in efforts such as the OpenDLib project [Castelli and Pagano, 2003] and the Open Digital Libraries (ODL) project [Suleman and Fox, 2001]. The primary aim of the OpenDLib project was to create a system that manages digital library services by providing an infrastructure with which a digital library can be customised on-the-fly, hence making the digital library expandable. The ODL project, in contrast, developed a set of independent reusable components that communicated using Web-based protocols, for the construction of information management systems. Typical components in both systems include metadata repositories and search engines.

This transition from monolithic digital library systems to those that are based on more flexible component-based frameworks has been influenced by a number of factors. One of these factors is that breaking down a complex Web-based system into smaller manageable pieces can be motivated as being a good strategy for taking maximum advantage of the distributed nature of the networks within which the resultant component-based system will be housed. A more classical factor however is that it is regarded as good practice in the field of software engineering to break down a complex system into smaller manageable pieces. The following section briefly looks into current software engineering practices that are employed in deriving Web-based applications.

3. SOFTWARE ENGINEERING FOR WEB DL APPLICATIONS

Since its inception not four decades ago, the field of software engineering has grown to become one of the pioneering disciplines at the forefront of this revolutionary Information and Communication Technology (ICT) era. Software engineering has, throughout the years, picked up a variety of definitions but has comfortably been accepted as 'software manufacturing' or 'software development' by the general public, perhaps due to it being an engineering which is defined by Wang [2000] as the 'profession devoted to designing, constructing, and operating the structures, machines, and other devices of industry and everyday life.'

One of the more recent sub-areas of software engineering is Component-based Software Engineering (CBSE). CBSE emerged to address three core aspects related to components: the development of individual components, the aggregation and composition of components into systems and the modification of existing systems by replacement or upgrading of individual components [Crnkovic, 2001]. CBSE is particularly applicable in the context of distributed systems and Web-based applications where an application may be distributed across numerous servers and is network-linked. This setup is ideal if one of the servers fails, since only a portion of the application will be affected by that failure and other parts may still be functioning. Other known advantages of CBSE include decreases in development time, increases in the usability of the products and decreases in the production costs [Crnkovic, 2001; Johnson, 2002].

An obvious disadvantage associated with CBSE is the effort required to develop, maintain and provide support for these components. In general, CBSE has many obvious benefits while its problems stem largely from the management of components. If adequate facilities are available to improve management of components, it is expected that system designers can enjoy the benefits of CBSE, while successfully addressing or at least reducing the impact of the disadvantages.

4. A MOTIVATION FOR PACKAGING

It is to this end that this research project was conceived, to look into how components may be managed from a distribution and redeployment perspective. This project has not looked into how services may be discovered or components identified – while this is an important problem, it is sufficiently complex and was deemed out of scope. Related parts of the larger research project have investigated how service-oriented components may be configured and composed visually and how the user interface can be designed visually (itself as a single component) to communicate with this suite of components [Suleman, et al., 2005]. The end result of the latter design tools are a set of components

and a formal description of how the digital library system is to be linked together to function as a single entity [Eyambe, 2005]. Packaging can then be used to create a redistributable installation package.

The project was further motivated by the frustrations experiences by users of earlier versions of systems such as EPrints and DSpace, when installation involved a series of auxiliary activities which did not always go smoothly. An aim of this project was therefore to attempt to address the following shortcomings identified by users in existing production systems:

1. Dependencies were not always available and it was not clear which parts needed to be installed and which did not.
2. A lot of the installation was not automated.
3. Superuser or administrator access was almost always necessary at some point.
4. Multiple installations could not easily co-exist on a single machine.

5. TERMINOLOGY

A **component** is a collection of library files that is normally useful only as part of a larger software package – in the object-oriented sense this is equivalent to a class.
A **component instance** is a specific set of parameters and machine APIs based on the component libraries – in the object-oriented sense this is equivalent to an instance.
A component instance cannot be executed on a machine unless the component is installed on that machine.
A **dependency** of component X is any other component or software module that X cannot function without.
A **dependency checking script/application** is a script/application that checks if a given dependency module is installed on a machine.
A **software package** is made up of a set of components and configuration information to create instances from each component.
An **installed software package** is a set of instances, based on a set of components, all of which reside on a physical machine.

6. SYSTEM DESIGN

This research aimed to demonstrate that a component packaging solution can improve the manageability of a digital library software toolkit, without losing the inherent flexibility of a component-based solution. Eyambe and Suleman [2004], have already shown that it is preferable to design a component-based digital library using a visual interface. This research thus builds on those efforts to, firstly, build a package from its specification and, secondly, install and configure that package as required. This system has been designed to primarily interface with derivatives of the ODL components. However, similar behaviour is expected for other types of components – the machine interface and APIs are language and technology agnostic. Employing the cross-platform Java programming language in conjunction with XML for data presentation was a design strategy adopted to address the issue of heterogeneous operational platforms. In the following sections, the design of the package builder and package installer subsystems is discussed. First, however, the specification language for digital library systems is presented, as this forms the initial input to the packaging and installation tools.

6.1 Component Connection Language (CCL)

The Component Connection Language (CCL) is a simple XML-based language to specify the connections among pairs of component instances and the configurations of each instance in the system. It is similar in nature to 5SL [Gonçalves and Fox, 2002], but it seeks only to specify high-level relationships, without attempting to encode the detailed semantics of the individual components being referred to. The language also is meant purely as an exploratory vehicle for research into DL architectures so completeness was not a criterion in its design.

Figure 1 shows a snippet from a typical CCL representing a very simple searchable archive with associated Web-based user interface. At the top level there are multiple component instance elements – because of space constraints only one is shown in the figure. Each instance contains the configuration information necessary for that particular software component. This is followed by a list of connection tags, each of which specifies a directed relationship between 2 instances. In the example shown, the user interface (UI) is connected to the search engine (Search), which is in turn connected to the data archive (Union).

```
<CCL>
...
   <instance>
      <instanceDescription>
         <name>Search</name>
         <description>
            <blox:irdb>
               <repositoryName>ODL Search Engine</repositoryName>
               <adminEmail>smhlongo@cs.uct.ac.za</adminEmail>
               <database>DBI:mysql:test</database>
               <dbusername>root</dbusername>
               <dbpassword>root</dbpassword>
               <table>search</table>
               <archive>
                  <identifier>HUSPICS</identifier>
                  <url>
                     <!--URL_TO_CGI_LOCATION-->/ODL-DBUnion-1.2/DBUnion/evaluation/union.pl
                  </url>
                  <metadataPrefix>oai_dc</metadataPrefix>
                  <interval>86400</interval>
                           <interrequestgap>10</interrequestgap>
                  <overlap>86401</overlap>
                  <granularity>second</granularity>
               </archive>
            </blox:irdb>
         </description>
      </instanceDescription>
   </instance>
...
   <connection>
      <from>UI</from>
      <to>Search</to>
   </connection>
   <connection>
      <from>Search</from>
      <to>Union</to>
   </connection>

   <questions>
      <question>
         <description>
            The administrator's e-mail is that to which all correspondence about the
            digital library will be directed.
         </description>
         <text>Please Input the Administrator's e-mail Address</text>
         <answer> </answer>
         <default>administrator@domain.suffix</default>
         <locations>
            <location>/CCL/instance[0]/.../description/ui/adminEmail</location>
            <location>/CCL/instance[1]/.../description/dbbrowse/adminEmail</location>
            <location>/CCL/instance[2]/.../description/irdb/adminEmail</location>
            <location>/CCL/instance[3]/.../description/dbunion/adminEmail</location>
         </locations>
      </question>
   </questions>
```

Figure 1. Fragment of a CCL specification

At the end of the CCL is a list of question tags. These indicate the list of questions that any configuration application ought to ask of the user installing the software. The list of location tags associated with each question are XPath specifications corresponding to the exact positions within the CCL where the answers are to be inserted/replaced to update the CCL after gathering information from the user at install-time. The semantics of these questions are best determined by the designer of each component so they are obtained from the individual components. However, the packager may choose to change the questions or use a single response in multiple locations – this optimisation of the installation process is a task performed when the components are being packaged.

The questions section of the CCL file was added specifically to support the packaging process. Prior to this, the configuration information for a single component instance was specified only as an XML Schema. However, an extension of XML Schema would be necessary to encode the questions to be asked and the default answers. While such an extension is somewhat straightforward, the storing of answers from users at install-time has to be done outside of the CCL since the XSDs would represent only the type information for individual components or the conglomerate CCL. They would also not be able to cross boundaries of individual components.

6.2 Package Builder

The aim of the package builder is to build a digital library package according to an input specification. Figure 2 shows all the steps of the package building process.

The input to the package building process is a Component Connection Language (CCL) specification file, as discussed in the previous section.

Once the CCL file has been analysed, the package builder then gathers other resources that are defined by each of the contained components. These resources include dependencies and dependency checking scripts. Default installation options, such as the installation location and digital library name, can then be specified.

The most comprehensive part of the package builder subsystem is the questions interface. This interface allows for existing installation questions to be edited and for new installation questions to be specified. In keeping with the notion that constructing a digital library system should be a visual process, the specification/editing of each question, along with the default answers and CCL locations is performed using visual navigation tools, as depicted in Figure 3. The locations are selected using a tree-view of the CCL, thus eliminating any typographical errors from manual editing of XML.

The final step of the package building process involves the creation of installation scripts and bundling of all resources making up the package.

6.3 Package Installer

The input to the package installation process is an installation script that was constructed during the package building process. This script informs the package installer of the installation questions as well as all the default values. System checks such as OS version and dependencies are then performed. If the returned results are favourable and all the questions have been attended to, the components are installed and configured by the installer. This process is depicted in Figure 4.

The installer makes an assumption that the target location of the system may contain Web-executable scripts – i.e., that any application placed in the location may be addressed and executed via the Web server on the machine. This may violate the basic requirement that the system work on multi-user systems without administrator intervention. However, it is possible to set up Web servers such as Apache to execute all scripts located within a defined directory for all users, without further administrator effort – this is usually the *public_html* directory and its descendants. In addition, in a sister project, a Universal Web Service Server is being designed to not only give each user a directory for Web-executable applications, but provide the users with sandboxes in which to execute Web applications in any language [Maunder, et al., 2005]. Coupled with tools such as the package installer, this will allow users on a multi-user Linux/BSD system to simply drop applications into their personal workspaces and have working Web applications within minutes.

It is often the case that the package also depends on other packages in order for it to function as desired. Sometimes it is possible to bundle these dependency modules with the software in one package, especially if the dependency modules do not have any intellectual property rights restrictions. Software installations are often hampered due to licence restrictions which forbid bundling all required packages in a single distribution (e.g., the Java Virtual Machine from Sun Microsystems). Some licences even go as far as forbidding software to be mirrored, thus limiting the availability of the software to just a single site and even at that only site, downloaders often have to fill in a form before being granted the right to acquire the software. This makes it even more difficult to automate the process of obtaining such dependencies.

In this project, each component defines its own set of dependencies i.e., the software library includes a list of all libraries it depends on. Each dependency is then associated with applications to check for its existence under various Operating Systems (OSs), with the intention of maximising the degree of automation with the greatest degree of generality.

In addition to installation, the installer needs to be aware of post-deployment needs of users. The most trivial of the post-deployment needs of any software package is that it should be possible to remove it from the system in which it has been installed if the package is no longer needed. Surprisingly, many packages either do not offer this feature or offer it but through a very elaborate and non-trivial interface.

The package installer subsystem thus offers the ability to remove an installed digital library. Migrating data across systems (or installing a digital library system with preloaded data) has been looked into and implemented for ODL components. One of the many advantages of adopting a component-based approach for building software systems is that changes to various components can be done without necessarily jeopardising the system as a whole. Future work will entail implementing an infrastructure whereby an installed digital library system can be upgraded componentwise.

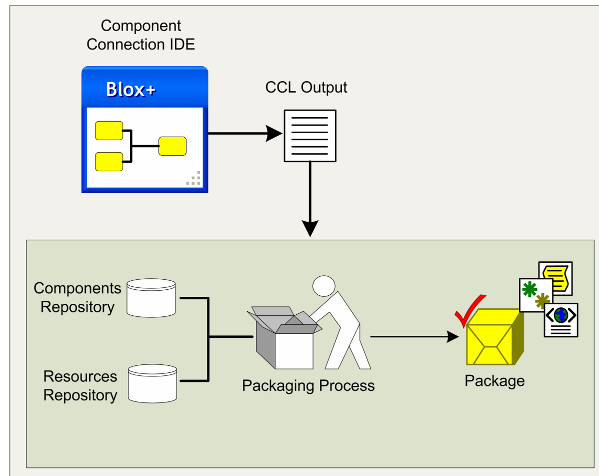

Figure 2. Package building process

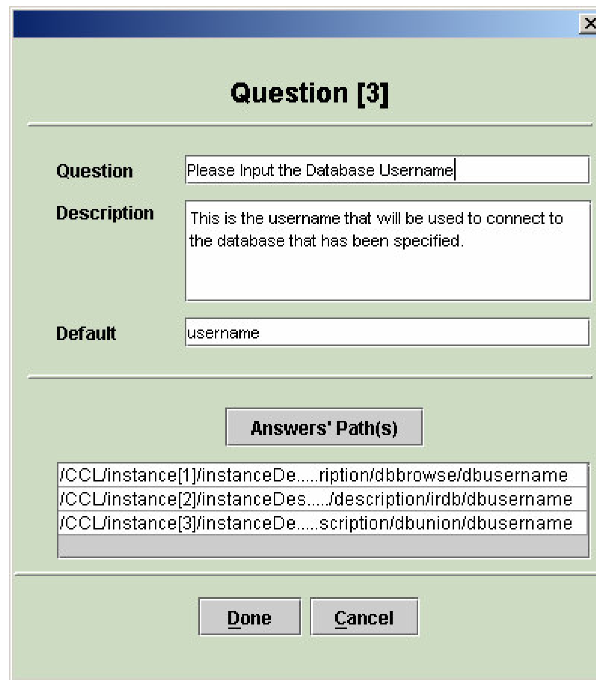

Figure 3. Question specification editing

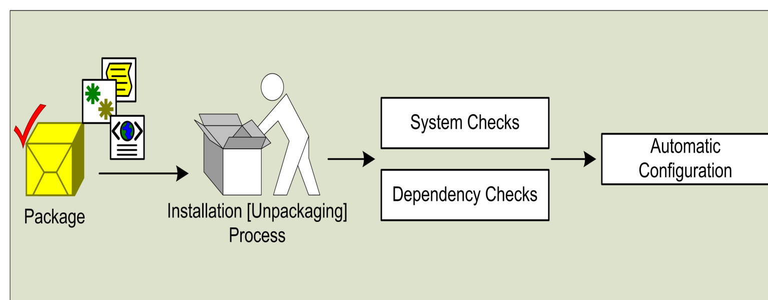

Figure 4. Package installation process

		Very Good	Good	Neutral	Poor	Very Poor	No Response
Textual Installation	Understanding	3	16	3	3	0	0
	Usability	4	6	11	3	0	1
Visual Packaging	Understanding	9	11	4	1	0	0
	Usability	10	14	1	0	0	0
Visual Installation	Understanding	7	12	5	1	0	0
	Usability	6	16	2	1	0	0

Table 1. Results from comparative user study

7. EXPERIMENT AND RESULTS

The packaging and installation system was subjected to a simple user-based evaluation to determine how developers would react to the system in isolation and when compared to a non-visual software installation process.

The experiment was made up of three parts. In the first part users installed a subset of components using the Linux command-line to configure the individual instances (the component-wise experiment). In the second and third parts, users were given a CCL specification (designed in advance) and asked to use this as the basis for creating a package visually, and subsequently installing it, also visually (the packaging experiment). The component-wise and packaging parts of the experiments were administered randomly. Users were asked about the level of their understanding of each of the three processes and how usable they thought the different approaches were. The aim of these experiments was primarily to determine how understandable and usable the system was. The results for some of the direct questions are listed in Table 1.

In general, users had a high degree of understanding of all of the three parts of the process.

For understanding of the visual packaging part, we can use a χ^2 test to demonstrate that the mean response is in fact normally distributed and centred on 'Good'. By not opting for the simpler t-test, it is possible to hypothesise a distribution of discrete values instead of just a mean. Following the recommendations for a χ^2 test, the expected set of frequencies were computed as {6, 8, 5, 5, 1}. Then, with a χ^2 value of 7.025, we cannot reject the null hypothesis that there is a normal distribution among the responses centred on "Good". Thus, we conclude that there is a normal distribution in the responses for understanding of visual packaging, with a mean of "Good".

Using a similar test, it can be concluded statistically that there is, on average, a "Good" understanding of the visual installation and textual installation processes.

In terms of usability, the results are not all similar. On average, users rated the usability of the textual process below that of the visual process, as is to be expected.

To confirm that users were not influenced by prior understanding of component-based systems (data for which was captured from each user), a χ^2 test of independence was performed. With a χ^2 value of 11.86, we can conclude that the users' understanding of the package installation process did not necessarily depend on a prior understanding of use of component-based systems. Similar tests support the conclusions that the experimental results are independent of any prior exposure to component-based systems.

(Note: The significance level was always 95% in all tests.)

Finally, given that users have a good understanding of all processes, they were asked to choose which approach they preferred and provide some feedback on their reasons. 92% of the users (23 out of 25) preferred the visual approach and provided reasons such as:

- 'The package installation process is a lot faster and it is less likely that mistakes are made during this process whereas with a componentwise installation process, one needs to know a lot more about the components.'
- 'The interface presented by package installation process is a lot easier to use and is more familiar while the componentwise approach presents a command-line-based installation process which is more error-prone and confusing.'
- 'The package had a better looking interface and was easier to follow while the componentwise approach did not have a friendly interface and required lots of unnecessary user input.'

The responses from the two participants who either preferred the textual method or were agnostic included:

- 'The componentwise approach is easier to work with and provides for a flexible configuration process whereas with the package approach, most processes are abstracted.'
- 'For power users, componentwise is the best option and for novice users, the package is a lot simpler.'

While these are valid issues raised by some users, it is clear that the vast majority of users preferred a visual tool to abstract away and manage the details for them. Also, the final comments can be addressed by the fact that the system can be modified as it is ultimately made up of components as well.

8. RELATED WORK

This work began with a study of many related packaging and popular tools, that provided a baseline set of features to simplify the processes of updating, upgrading and uninstalling software systems. With most current software management tools, the emphasis is still on packaging monolithic software applications, with little or no provision for those applications based on a component framework with well-defined interfaces for component configuration. Another major disadvantage with many non-commercial software management tools is that they tend to be specific to the OS which they offer solutions to. The following sections briefly discuss some of the most popular of these tools that are currently available.

8.1 Red Hat Package Manager (RPM)

RPM is the most widely used Linux package manager that many commercial Linux distributions are based on [Bailey, 2000; Gagne, 2000]. An entry point into building an RPM package is creating a formal recipe that RPM will understand and convert into a package according to the recipe's specifications. This recipe, the RPM specification file, has sections that address specific characteristics of the distribution package to be built. RPM also maintains a database of all the packages that are installed on a particular system to facilitate package management.

RPM is known to have inconsistencies when it comes to package names, package contents and dependency handling, but perhaps the most noticeable disadvantage is that only privileged users are able to install RPM based packages, since the RPM database earlier mentioned can be accessed only by privileged users. Our solution offers a localised manner for managing packages, local in a sense that each package has its own manager attached to it and that all the installations are local to the user installing the package. Another negative for RPM is that, at its core, it is command-line based and, as such, offers a steep learning curve for those interested in learning how it works. Another packaging system that is similar to RPM in many aspects is the Advanced Packaging Tool (apt) used in Debian and Debian-based Linux distributions.

RPM was used in initial tests to create a package of digital library system components. However, this package could only be installed once and by an administrator. In addition, it was not possible to specify dependency tests or other actions that need to be performed in order to activate a software installation. At the very outset RPM was thus abandoned as a possible solution framework.

8.2 emerge

emerge is a utility command that is used to access most of the functions provided by the Gentoo Linux package management system, Portage [Cowie, 2005]. Once issued appropriate commands, the emerge tool will download the desired application together with all its dependencies as source files, compile all the downloaded source files and install everything in a sandbox environment.

Applications are downloaded as source files in order to exploit full functionality of the emerge utility. It is possible for emerge to obtain just the binaries of an application for optimisation purposes. Like RPM, Portage is a global package management system and, as such, the emerge utility can be used only by users with specific privileges.

8.3 Windows Installer

The Windows Installer is not an installation program but a basic installation tool with which applications are installed on Microsoft Windows systems. Many applications, such as Wise and InstallShield, that install software on Microsoft Windows systems use the Windows Installer engine in order to maintain consistency in the internal database that Windows maintains. This ensures reliable operation of important installation features such as rollback and software versioning. Another powerful feature inherent in using the Windows Installer is the automatic generation of the uninstallation sequence for a particular application.

The Windows Installer tool does not install dependencies, but any installation tool that uses the Windows Installer could address this.

8.4 InstallShield

InstallShield is the most comprehensive cross-platform software management solution available [Macrovision Europe Ltd., 2005]. With InstallShield, publishers are able to create packages of complex software applications for deployment on multiple OSs, as Web Services or even on mobile devices.

InstallShield is a commercial tool and therefore is mainly targeted at corporate institutions. It too, however, only has limited support for the management of dependencies and is primarily aimed at software developers building applications without an explicit component model rather than composing applications from components.

9. ANALYSIS AND CONCLUSIONS

This work has largely been driven by the construction of a proof-of-concept prototype to package a system composed of components, thereby improving its manageability. This prototype was tested by a small group of budding software developers and found to be both usable and understandable and preferable to non-visual approaches to component installation. This confirms the relative usefulness of a component management framework and the applicability of such technology to digital library and Web application system design and implementation. While the system development and experiments were done in the context of digital library systems, there is no fundamental difference between digital library systems and Web-based systems in general – the components used in these experiments were selected primarily because of availability. Related efforts have demonstrated that Web-based search engines and Web-based bulletin boards can easily be cast into the system of ODL components and instances.

The shortcomings identified earlier were addressed in that: dependencies were included in the final packages and the user did not need to select among them; the degree of automation was higher than non-packaged solutions; none of the software tools required administrator access to the machine; and multiple packages could be installed at once on a single machine.

A more subtle, yet possibly more important outcome of the project has been the derivation of a set of requirements (many of which were tested through the prototype) for packaging systems for component-based digital library systems. Notable among those are the following:

- A general packaging system must be platform and/or language independent.
- Dependencies must be catered for – this includes the testing, packaging and installation of such. Intellectual property rights may need to be dealt with for some dependencies, possibly in the form of embedded machine-readable rights descriptions.

Equally important is a set of requirements for the machine interfaces of individual components. Notable among those are the following:

- Every component must have self-description capabilities i.e., it must reflect its interfaces, parameters and capabilities.
- It must be possible to create and configure instances of each component without user interaction.
- Sequential configuration information, necessary to power configuration wizards, may be specified at the level of the system or each component – however, if specified at the component level, a higher degree of information hiding can be achieved.
- Components and interfaces must be appropriately named, with strong conventions for version numbering. This is important to support dependency resolution.

An early aim of this project was to design a packaging system that would not require administrator privileges on a multi-user system. The prototype sufficiently demonstrates that this is possible. Some degree of consistency is lost with multiple copies of components on a single machine. However, this allows for a clean separation among users sharing the system. This functionality is particularly useful in developing countries where adoption of digital library systems is still hampered by the scarcity of equipment for experimentation and production use. In such environments, the lack of skilled administrative support makes it even more crucial to concentrate on a few large multi-user systems rather than a lot of mostly single-user machines, each administered by its owner.

Ultimately, a major outcome of this research is that flexible packaging methodologies are indeed feasible for at least some Web-based component-based applications.

10. FUTURE WORK

First and foremost, it is hoped that the proof-of-concept prototype and evaluations thereof serve as an indicator for the type of software architecture that can be used in future versions of prepackaged digital library system software. The next version of Greenstone [Bainbridge, et al., 2004] promises to adopt some of the ideas that have resulted from earlier research on components and the next version of DSpace could do likewise [Tansley, 2004].

From a research perspective, much work remains to be done in determining how best to encode, comprehend and act upon rights expressions embedded in software aggregations. The trust issues need to be addressed so that users can install software without requiring high speed Internet connections that many simply do not possess. Also, information may need to flow back to component sources from click-through agreements and registration forms.

After a package has been installed, the system can be maintained by exchanging components for ones that have a higher level of efficiency. Sometimes it is simply not possible to squeeze additional functionality out of an ageing system – in this case the solution is to purchase a faster system to replace the existing one. In the developing world, there are typically fewer resources to do this, so a move towards cluster computing is more favourable. As such, a package being deployed on a single system should as easily work on a cluster of machines if they are available. A framework to support this at a high level is currently being developed as a natural extension of this work.

While the Flexible Digital Library project developed multiple nominally independent (bar the CCL) tools for visually building digital library systems, an integration of these tools in production systems may make it even simpler to design and create digital library tools. Ultimately, it should be possible for a designer who so desires to put together the

search facilities of Greenstone with the workflow management of EPrints and the user interface of DSpace and create a new package for redistribution, all without a single line of coding!

11. ACKNOWLEDGEMENTS

Thanks are due to the NRF and the Telkom/Siemens/THRIP Centre of Excellence for funding that supported this research.

12. REFERENCES

BAILEY, E. C. 2000. Maximum RPM. Red Hat Inc. http://www.rpm.org/max-rpm/.

BAINBRIDGE, D., DON, K. J., BUCHANAN, G. R., WITTEN, I. H., JONES, S., JONES, M., AND BARR, M. I. 2004. Dynamic Digital Library Construction and Configuration. In *Proceedings of Research and Advanced Technology for Digital Libraries: 8th European Conference (ECDL2004)*, Bath, UK, 12-17 September 2004, LNCS 3232, Springer.

CASTELLI, D., AND PAGANO, P. 2003. A System for Building Expandable Digital Libraries. In *Proceedings of the Third ACM/IEEE-CS Joint Conference on Digital Libraries,* Houston, USA, May 2003, ACM Press, New York, NY, 2003, 335-345.

COWIE, A. 2005. Gentoo for all the unusual reasons. *Linux Journal 130*, February 2005.

CRNKOVIC, I. Component-based Software Engineering – New Challenges in Software Development. *Software Focus 2(4)*, November 2001, 127-133.

EYAMBE, L. 2005. *A Digital Library Component Assembly Environment*. MSc Thesis, Department of Computer Science, University of Cape Town.

EYAMBE, L., AND SULEMAN, H. 2004. A Digital Library Component Assembly Environment. In *SAICSIT '04: Proceedings of the 2004 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, Stellenbosch, South Africa, ACM Press, 15-22.

GAGNE, M. 2000. Cooking with Linux: Rapid Program-Delivery Morsels, RPM. *Linux Journal 73*, May 2000.

GONÇALVES, M. A., AND FOX, E. A. 2002. 5SL: a language for declarative specification and generation of digital libraries. In *Proceedings of Joint Conference on Digital Libraries 2002*, Portland, USA, 263-272.

JOHNSON, P. D. 2002. Mining Legacy Systems for Business Components: An Architecture for an Integrated Toolkit. In *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, Washington DC, USA, 2002, IEEE Computer Society, 563-571.

MACROVISION EUROPE LTD. 2005. Installshield: The Industry's Leading Installation-Authoring Solution. http://www.installshield.com/products/installshield/.

MAUNDER, A., VAN ROOYEN, R., AND SULEMAN, H. 2005. Designing a Universal Web Application Server. In *Proceedings of SAICSIT 2005*, 20-22 September, White River.

SULEMAN, H., FENG F., MHLONGO, S., AND OMAR, M. 2005. Flexing Digital Library Systems. In *Proceedings of the 8th International Conference of the Asian Digital Library*, Bangkok, Thailand, 12-15 December 2005, LNCS, Springer-Verlag.

SULEMAN, H. AND FOX, E. A. 2001. A Framework for Building Open Digital Libraries. *D-Lib Magazine 7(12)*, December 2001.

TANSLEY, R. 2004. DSpace 2.0 Design Proposal, presented at DSpace User Group Meeting, 10-11 March, Cambridge, USA, 2004. http://wiki.dspace.org/DspaceTwo.

TANSLEY, R., BASS, M., STUVE, D., BRANCHOFSKY, M., CHUDNOV, D., MCCLELLAN, G., AND SMITH, M. 2003. The DSpace Institutional Digital Repository System: Current Functionality. In *Proceedings of Joint Conference on Digital Libraries 2003*, Houston, TX, May 27-31, 2003, ACM Press, New York, NY, 87-97.

UNIVERSITY OF SOUTHAMPTON. 2006. EPrints.org. http://www.eprints.org/.

WITTEN, I. H., AND BAINBRIDGE, D. 2002 *How to build a digital library*. Morgan Kaufmann, San Francisco, CA.