Binary Change Set Composition

Tijs van der Storm

Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB
Amsterdam, The Netherlands
`storm@cwi.nl`

Abstract. Binary component-based software updates that are efficient, safe and generic still remain a challenge. Most existing deployment systems that achieve this goal have to control the complete software environment of the user which is a barrier to adoption for both software consumers and producers. Binary change set composition is a technique that can be applied to deliver incremental, binary updates for component-based software systems in an efficient and non-intrusive way. This way application updates can be delivered more frequently, with minimal additional overhead for users and without sacrificing the benefits of component-based software development.

Keywords: deployment, update management, component-based software engineering, software configuration management.

1 Introduction

An important goal in software engineering is to deliver quality to users frequently and efficiently. Allowing users of your software to easily take advantage of new functionality or quality improvements can be a serious competitive advantage. This insight seems to be widely accepted [10]. Software vendors are enhancing their software products with an automatic update feature to allow customers to upgrade their installation with a single push of a button. This prevents customers from having to engage in the error-prone and time consuming task of deploying new versions of a software product. However, such functionality is often proprietary and specific to a certain vendor or product, thereby limiting understanding and broader adoption of this important part of the software process.

The aim of this paper is to maximize the agility of software delivery without sacrificing the requirement that applications are developed as part of a component-based product line. While it may not be beneficial to force the user environment to be component-based, it certainly can be for the development environment. One would like to develop software in a component-based fashion, and at the same time allow users to transparently deploy an application as a whole.

If certain actions are tedious, error-prone or just too expensive, they tend to be performed less frequently. If the effort to package a software product in such a way that it is ready for deployment is too high, releases will be put out less frequently. Similarly, if deploying a new release is a time consuming activity with a high risk of failure, the user probably will not upgrade every day. Therefore, if we want to optimize software delivery this can be achieved by, on the one hand, reducing the cost of release, and on the other hand, by reducing the cost of deployment.

How would one optimize both release and deployment in a platform and programming language independent way, when many products composed of multiple shared components have to be released and deployed efficiently? In this paper I present a technique, called *binary change set composition*, which provides an answer to this question. It can be used to implement lightweight incremental application upgrade in a fully generic and platform indepedent way. The resulting binary upgrades are incremental, making the upgrade process highly efficient.

Contributions The contributions of this paper are summarized as follows:

1. A requirements analysis of application upgrade and a survey of related work.
2. The design of a lightweight, efficient and platform independent method for application upgrade.
3. The implementation of this method on top of Subversion.

Organization This paper is organized as follows. Section 2 provides some background to the problem of application upgrade by identifying the requirements and discussing related work. Section 3 forms the technical heart of this paper. I describe how to automatically produce releases and deliver updates in an incremental fashion. The implementation of the resulting concepts is then discussed in Section 4. Then, in Section 5, I evaluate the approach by setting it out against the requirements identified in Section 2. Finally, I present a conclusion and list opportunities for future work.

2 Background

2.1 Requirements for Application Upgrade

Application upgrade consists of replacing a piece of software that has previously been installed by a user. The aim of an upgrade for the user is to be able to take advantage of repaired defects, increased quality or new functionality. The business motivation for this is that customer satisfaction is increased. To achieve this goal, the primary requirement is that upgrades *succeed*. Nevertheless, there are additional requirements for application upgrade. In the paragraphs below I discuss four requirements: *lightweightness*, *efficiency*, *genericity* and *safety*.

For an software deployment method to be lightweight, means that (future) users of a software product should not be required to change their environment to accomodate the method of deployment of the product. Reasoning along the same lines, the method of creating deployable release should not force a development organization to completely change their development processes. Furthermore, the effort to create a release on the one hand, and the effort to apply an upgrade on the other hand, should require minimum effort.

Efficiency is the third requirement. If the aim is to optimize software delivery, both release and upgrade should be implemented efficiently. If deploying an upgrade takes too much time or consumes too much bandwidth, users will tend to postpone the possibly crucial update. Again, also the development side gains by efficiency: the storage requirements for maintaining releases may soon become unwieldy, if they are put out frequently.

To ease the adoption of a release and deployment method, it should not be constrained by choice of programming language, operating system or any other platform dependency. This genericity requirement mostly serves the development side, but obviously has consequences for users: if they are on the wrong platform they cannot deploy the application they might desire.

The final requirement serves primarily users: safety of upgrades. Deployment is hard. If it should occur that an upgrade fails, the user must be able to undo the consequences quickly and safely. Or at least the consequences of failure should be local.

2.2 Related Work

Related work exists in two areas: update management and release management,—both areas belong to the wide ranging field of software deployment. In this field, update management has a more user oriented perspective and concerns itself with the question how new releases are correctly and efficiently consumed by users. Release management, on the other hand, takes a more development-oriented viewpoint. It addresses the question of how to prepare software that is to be delivered to the user.

In the following I will discuss how existing update and release tools for component-based software deployment live up to the requirements identified in Section 2.1.

Research on software deployment has mostly focused on combining both the user and development perspectives. One example is the Software Dock [11], which is a distributed architecture that supports the full software deployment life cycle. Field docks provide an interface to the user's site. These docks connect to release docks at producer sites using a wide area event service. While the software dock can be used to deploy any kind of software system, and thus satisfies the genericity requirement, the description of each release in the Deployable Software Description (DSD) language presents significant overhead. Moreover, the Software Dock is particularly good at deploying components from different, possibly distributed origins, which is outside the scope of this paper. The same can be said of the Software Release Manager (SRM) [18].

Deployment tools that primarily address the user perspective fall in the category of software product updaters [12]. This category can be further subdivided into monolithic product updaters and component-based product updaters. Whereas product updaters in general do not make assumptions on the structure of the software product they are updating, component (or package) deployment tools are explicitly component-based.

JPloy [14] is a tool that gives users more control over which components are deployed. The question is, however, whether users are actually interested in how applications are composed. In that sense, JPloy may not be a good match for application deployment in the strict sense.

Package deployment tools can be further categorized as based on source packages or binary packages. A typical example of source-based package deployment tools is the FreeBSD ports system [15]. Such systems require users to download source archives that are subsequently built on the user's machine. Source tree composition [6] is another approach that works by composing component source distributions into a so-called *bundle*. The tool performing this task, called AutoBundle, constructs a composite build interface that allows users to transparently build the composition. Source-based deployment, however, is relatively time-consuming and thus fails to satisfy the efficiency requirement.

Binary package deployment tools do, however, satisfy the efficiency requirement. They include Debian's Advanced Package Tool (APT) [16], the Redhat Package Manager (RPM) [3], and more recently AutoPackage [2]. These tools download binary packages that are precompiled for the user's platform. Both APT and RPM are tied to specific Linux distributions (Debian/Ubuntu and Redhat/SuSe respectively) whereas autopackage can be used across distributions. Nevertheless AutoPackage only works under Linux. Although these deployment tools are independent of programming language, they are not generic with respect to operating system.

The deployment system Nix [7] supports both source and binary deployment of packages in such a way that it is transparent to the user. If no binary package is found it falls back to source deployment. It features a store for non-destructively installing packages that are identified by unique hashes. This allows side-by-side installation of different versions of the same package. Nix is the only deployment tool that is completely safe because its non-destructive deployment model guarantees that existing dependencies are never broken because of an update. Furthermore, it is portable across different flavors of Unix and does not require root access (which is the case for all package deployment tools except AutoPackage).

One problem in general with package deployment tools is that they are invasive with respect to the environment of the user. For instance, the value of these tools is maximum when *all* software is managed by it. This explains why most such tools are so intertwined with operating system distributions, but it is a clear violation of the lightweightness requirement.

While some systems, such as Nix, AutoPackage and JPloy, can be used next to the 'native' deployment system, they still have to be able to manage all dependencies in addition to the component that the user actually wants to install. In the worst case this means that a complete dependency tree of packages is duplicated, because the user deployed her application with a deployment tool different from the standard one. Note that this is actually unavoidable if the user has no root access. Note also that the user is at least required to install the deployment system itself, which in turn may not be an easy task.

2.3 Overview of the Approach

The motivations for component-based development are manyfold and well-known. Factoring the functionality of an application in separate components, creates opportunities for reuse,—both within a single product or across multiple products [17]. Similarly, productivity is increased because components can be developed in parallel. In this paper components are interpreted as groupings of files that can be versioned as a whole.

Components are not stand-alone applications. This means that a component may require the presence of other components to function correctly. Such dependencies may be bound either at build-time or at runtime. Applications are then derived by composing constituent components.

In the following I assume a very liberal notion of dependency, and consequently of composition. When one component requires another component it is left unspecified what the concrete relation between the two components amounts to. Abstract dependencies thus cover both build-time and runtime dependencies. Under this interpretation, composition is loosely defined as merging all files of all related components into a single directory/archive.

When a component has been built, some of the resulting object files will contribute to the composed application. This set of files is called the (component) distribution. To distribute an application to users, the relevant component distributions are composed before release, resulting in a single application distribution. Thus, an application is identified with a certain root node in the component dependency graph and its distribution consists of the transitive-reflexive closure of the dependencies below the root.

In the next section I will present a technique to efficiently create and deliver such application releases, called *binary change set composition*. We will see that continuous integration of component-based software extends naturally to a process of automatic continuous release. A component will only be built if it has changed or if one of its dependencies has changed. If a component has been built it is released automatically. The results of a build are stored persistently so that components higher up in the dependency graph may reuse previous builds from components lower in the dependency graph.

Apart from the files belonging to a single component, the composition of these sets of files is also stored. The space requirements for this can quickly become unwieldy, therefore these application distributions are stored differentially. Differential storage works by saving the changes between files. Instead of composing sets of files, one can now compose sets of change sets. In addition to storing many releases efficiently, binary change set composition yields an efficient way of updating user installations.

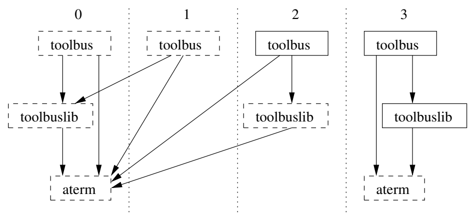

Fig. 1. Incremental integration

3 Binary Change Set Composition

3.1 Incremental Integration

Tools like *make* optimize software builds because it only updates targets when they are out of date. It is possible to lift this paradigm from the level of files to the level of components. Hence, a component is only built if it is out of date with respect to some saved state, or when one of its dependencies is out of date. If built artifacts are stored persistently they can be reused. Sharing of builds is particularly valuable when a software product is continuously integrated [9]. Traditionally this involves building the complete application as soon as someone commits changes to the source control system. However, building large systems from scratch may not scale.

Consider an example that derives from three real-world components, *toolbus*, *toolbuslib* and *aterm*. The Toolbus is a middleware component that allows components ("tools") to communicate using a centralized software bus. Tools implemented in C use the *toolbuslib* component for this. Using the Toolbus, tools exchange data in a tree-like exchange format called Annotated Terms (ATerms) this datastructure is implemented by the *aterm* component. Obviously, *toolbus* requires both the connection and the exchange format libraries, whereas the connection library only requires the exchange format. All three components are used with the ASF+SDF Meta-Environment, a component-based application for language development [4].

Figure 1 shows four build iterations. The dashed boxes indicate changes in that particular component. In the first iteration every component has been built. At the time of the second iteration, however, only the top-level toolbus component has changed, so it is built again but this time reusing the previous builds of *toolbuslib* and *aterm*. Similarly, in the third iteration there has been a change in the *toolbuslib* component. Since *toolbus* depends on *toolbuslib* a new build is triggered for both *toolbuslib* and *toolbus*. Finally, in the last iteration changes have been committed to the *aterm* component and as a result all components are rebuilt.

An implementation of incremental continuous integration, called Sisyphus, has been described in [19]. This system works as follows. Every time a commit to the source control system occurs, Sisyphus checks out all components. It does this by starting with a root component, and reading a special file contained in the source tree that describes the dependencies of this component. This process is repeated for each of the dependencies. Meanwhile, if the current version of a component has not been built before, or one of its dependencies has been built in the current iteration, a build is triggered. Results are stored in a database that serves as saved state.

3.2 Build and Release Model

The build and release model presented in this section is loosely based on the model presented in [19]. It can be seen as the data model of a database for tracing change, build and release processes. The state of a component at a certain moment in time is identified with its version obtained from the source control system. Each version may have been built multiple times. The model records for every build of a component version which builds were used as dependencies. A set of built artifacts is associated to each build. Finally, a release is simply the labeling of a certain build; the set of releases is a subset of the set of builds.

In the context of this paper two sets are important: *Build*, the set that represents component builds, and *Use* defined as a binary relation between builds (i.e. $\mathit{Use} \subseteq \mathit{Build} \times \mathit{Build}$). The set of built artifacts contributed by a build b is given by $\mathrm{files}(b)$.

The extent of a build is defined as the set of builds that have participated in a build. The extent of a build b is computed by taking right image of b in the transitive-reflexive closure of the *Use* relation:

$$\mathrm{extent}(b) = \mathit{Use}^*[b]$$

The extent of a build contains all builds that will make up an application release. The set of files that will be part of a release is derived from the set of files that each component in the extent contributes. This is discussed in the next section.

3.3 Prefix Composition

When a component has been built some of the resulting object files will contribute to the composed application. The set of files that is distributed to the user is called the application distribution, and it is composed of component distributions.

Figure 2 shows how the files contributed by each component to the toolbus application are taken together to form a single application distribution. On the left is shown that all installable files of each component first end up in a component specific directory,—in the example this could have been the result of issuing *make install*. To release the *toolbus* as an application, these sets of files and directories are merged, resulting in a single application distribution, as shown on the right.

I call this way of composing components "installation prefix composition" since the component directories on the left correspond to path prefixes passed

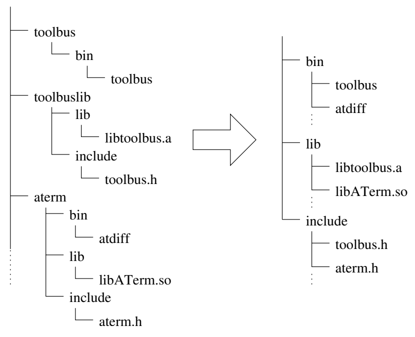

Fig. 2. Prefix composition

to *./configure* script that is generated by AutoConf [1], a tool to configure build processes that is widely used in open source projects. Among other things, it instructs *make install* to install files to a Unix directory hierarchy below the prefix.

Since components are composed by merging sets of files and directories we must ensure that no component overwrites files of another component. Formally, this reads:

$$\forall b \in \mathit{Builds} : \bigcap_{b' \in \mathrm{extent}(b)} \mathrm{files}(b') = \emptyset$$

In other words, this ensures that making a distribution is compositional. Instead of explicitly creating a global application distribution one can compose individual component distributions to achieve the same effect. What the property effectively states is that building a component, viewed as a function, distributes over composition.

There is one technicality which has to be taken care of: the distributed files should be relocatable. Because builds happen at the developer's site one must ensure that no (implicit) dependencies on the build environment are bound at build time. For instance, if a Unix executable is linked to a dynamic library that happens to be present at build time, then this library should also be present on the user's machine,—even on the same location. Since we do not want to require that users should reproduce the complete build environment, care must be taken to avoid such "imported" dependencies. I elaborate on this problem in Section 4.3.

3.4 Change Set Delivery

If the compositionality property holds the composition is defined by collecting all files that are in the extent of a build:

$$\mathrm{files}^*(b) = \bigcup_{b' \in \mathrm{extent}(b)} \mathrm{files}(b')$$

The function files* computes the set of files that eventually has to be distributed to users. An update tool could transfer these files for every build that is released to the users of the application. If a user already has installed a certain release, the tool could just transfer the difference between the installed release and the new release. Let $F_{1,2} = \text{files}^*(b_{1,2})$. Then, the change set between two releases b_1 and b_2 is defined as:

$$\{\varDelta(F_1 \cap F_2), +(F_2 \backslash F_1), -(F_1 \backslash F_2)\}$$

Change sets have three parts. The first part, indicated by $\varDelta$ contains binary patches to update files that are in both releases. The second and third part add and remove the files that are absent in the first or second release respectively.

Upgrade	Change set delivered to user
$0 \rightarrow 1$	$\{\varDelta_1^0$bin/toolbus$\}$
$1 \rightarrow 2$	$\{\varDelta_2^1$bin/toolbus, $\varDelta_2^0$lib/libtoolbus.a$\}$
$2 \rightarrow 3$	$\{-$bin/atdiff$\}$

Table 1. Change set delivery

If we turn our attention once again to Figure 2, we see on the right the composed prefix for the *toolbus* application. Let's assume that this is the initial release that a typical user has installed. In the meantime, development continues and the system goes through three more release cycles, as displayed in Figure 1. The sequence of change sets transferred to our user, assuming she upgrades to every release, is listed in Table 1.

The second iteration only contains changes to the *toolbus* component itself. Since the only installable file in this component is *bin/toolbus*, a patch is sent over updating this file at the user's site. In the next iteration there is a change in *toolbuslib* and as a consequence *toolbus* has been rebuilt. Updating to this release involves transferring patches for both *bin/toolbus* and *lib/libtoolbus.a*. There must have been a change in the *bin/toolbus* since *libtoolbus.a* is statically linked. In the final iteration the changes were in the *aterm* component. However, this time neither *toolbuslib* nor *toolbus* are affected by it—even though they have been rebuilt—because the change involved the removal of a target: the *bin/atdiff* program appears to be no longer needed. Neither *toolbus*, nor *toolbuslib* referenced this executable, hence there was no change in any of the built files with respect to the previous release. As a result, the change set only contains the delete action for *bin/atdiff*. Note that these change sets can be easily reverted in order to support downgrades.

3.5 Change Set Composition

Until now we have assumed that every application release was completely available and the change sets were only used to optimize the update process. From

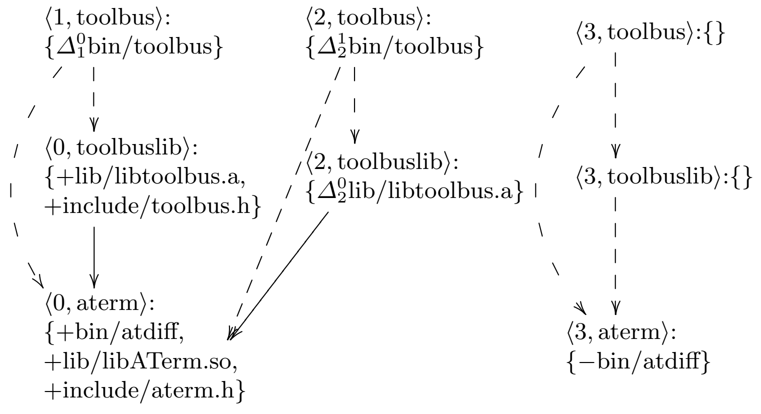
$\langle 1, \text{toolbus}\rangle$: $\{\varDelta_1^0\text{bin/toolbus}\}$
$\langle 2, \text{toolbus}\rangle$: $\{\varDelta_2^1\text{bin/toolbus}\}$
$\langle 3, \text{toolbus}\rangle$:{}
$\langle 0, \text{toolbuslib}\rangle$: {+lib/libtoolbus.a, +include/toolbus.h}
$\langle 2, \text{toolbuslib}\rangle$: $\{\varDelta_2^0\text{lib/libtoolbus.a}\}$
$\langle 3, \text{toolbuslib}\rangle$:{}
$\langle 0, \text{aterm}\rangle$: {+bin/atdiff, +lib/libATerm.so, +include/aterm.h}
$\langle 3, \text{aterm}\rangle$: {−bin/atdiff}

Fig. 3. Change set composition

the use of change sets to update user installations, naturally follows the use of change sets for storing releases. Figure 3 shows how this can be accomplished.

Once again, the three integration iterations are shown. In the first iteration, only the *toolbus* had changed and had to be rebuilt. This resulted in an updated file *bin/toolbus*. The figure shows that we only have to store the difference between the updated file and the file of the previous iteration. Note that initial builds of *aterm* and *toolbuslib* (from iteration 0) are stored as change sets that just add files.

The second iteration involves a change in *toolbuslib*; again, patches for *toolbus* and *toolbuslib* are stored. However, in the third iteration, the change in the *aterm* component did not affect any files in *toolbus* or *toolbuslib*, so no change sets need to be stored for these components. But if users should be able to update their installation of the toolbus application, still the toolbus should be released. So there really are four toolbus releases in total, but the last one only contains changes originating from *aterm*.

I will now describe how this scheme of binary change set composition can be implemented on top of Subversion.

4 Implementation using Subversion

4.1 Composition by Shallow Copying

Subversion [5] is a source control system that is gaining popularity over the widely used Concurrent Version System (CVS). Subversion adds many features that were missing in CVS, such as versioning of directories and a unified approach to branching and tagging. Precisely these features prove to be crucial in the implementation of binary change set composition on top of Subversion.

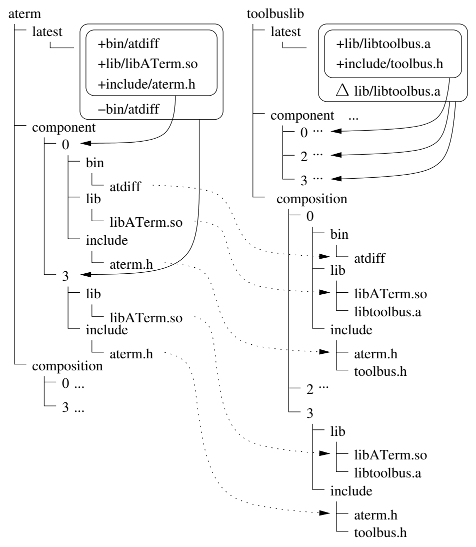

Fig. 4. Composition by shallow copying

Next, I will describe how Subversion repositories can be used as release repositories that allow the incremental delivery of updates to users. The release process consists of committing the component distributions to a Subversion repository, and then use branching to identify component releases. Such component-release branches are the unit of composition, which is also implemented by branching.

The crucial feature of Subversion that makes this work, is that branching is implemented by copying. So, for instance a branch is created for some repository location—file or directory—by copying the tree to another location. At the new location, Subversion records a *reference* to the source of the copy operation. The copy operation is a constant-space operation and therefore a feasible way to implement sharing.

Figure 4 shows a snapshot of a Subversion repository containing *aterm* and *toolbuslib* releases based on the change set graph displayed in Figure 3. For the sake of presentation releases of the *toolbus* have been omitted. On the left we see the Subversion tree for *aterm*, and on the left the tree for *toolbuslib*. The trees have subtrees indicated *latest*, *component* and *composition*. The *latest* tree is where component distributions are stored. The rounded boxes contain the change sets from Figure 3. The *component* tree and the *composition* tree contain shallow copies of versions of the latest tree; these are the releases proper. Solid arrows indicate copy relations the context of a single component,—dotted arrows indicate cross component copying (i.e. composition relations).

After every build the changes in the distributions are commited to the *latest* tree. The state of the *latest* tree at that time is then copied to a branch identifying this particular build; such branches are created by copying the files from latest to a separate directory under *component*. Note that since the change set for *toolbuslib* in iteration 3 was empty, *toolbuslib* release 3 is created from the state of the latest tree at iteration 2.

The tree below *composition* contains releases for compositions. This works by, instead of just copying the files belonging to a single build, copying the files in the extent of the build. In the example, this means that, next to the files contained in *toolbuslib* releases also the files in *aterm* releases are copied. If we compare *toolbuslib* composition 0 and 3, one can see in the figure that composition 0 is composed with release 0 of *aterm*, whereas composition 3 is composed with release 3 of *aterm*, exactly as in Figure 3.

4.2 Upgrade is Workspace Switch

Assuming the proper access rights are in place, the Subversion repository can be made publicly accessible for users. A user can now *check out* the desired subtree of *compositions*; this can easily be performed by a bootstrap script if it is the initial installation. She then obtains the composed prefix of the application.

Now that the user has installed the application by checking out a repository location, it is equally easy to down- or upgrade to a different version. Since the subtrees of the *composition* tree contain all subsequent releases of the application, and the user has checked out one of them, up- and downgrading is achieved by updating the user's local copy of the composed prefix to another release branch. Subversion provides the command *svn switch* for this. Subversion will take care adding, removing or patching where necessary.

Note that the sharing achieved in the repository also has an effect on how local checkouts are updated. For instance, recall that the third release of *toolbus* in the example involved the removal of *bin/atdiff*. If we assume that the user has installed the second release, and decides to upgrade, the only action that takes place at the user site is the removal of *bin/atdiff*, since the third release of both *toolbus* and *toolbuslib* contain the same change sets as second release of both these components.

4.3 Techniques for Relocatability

Installed application releases are ready to use with the exception of one technicality that was mentioned before, which is: relocation. Since the released files may contain references to locations on the build server at the side of development, these references become stale as soon as the users installed them. We therefore require that applications distributed this way should be binary relocatable. There are a number of ways to ensure that distributions are relocatable. Some of these are briefly discussed below.

There are ways to discover dynamically what the locations are of libraries and/or executables that are required at runtime. For instance, AutoPackage [2] provides a (Linux-only) library that can be queried at runtime to obtain 'your' location at runtime. Since the files contributed by each component are composed into a single directory hierarchy, dependencies can be found relative to the obtained location.

Another approach is to use wrapper scripts. As part of the deployment of an application a script could be generated that invokes the deployed application. This script would then set appropriate environment variables (e.g. PATH or LD_LIBRARY_PATH on Unix) or pass the location of the composed prefix on the commandline.

Finally, we could use string rewriting to effectively relocate unrelocatable files just after deployment. This amounts to replacing build time paths with their runtime counter-parts in every file. Special care must be taken in the case of binary files, since it is very easy to destroy their integrity. This trick, however, has been applied successfully.

5 Evaluation

5.1 Experimental Validation

A prototype implementation has been developed as part of the Sisyphus integration framework [20]. It has been used to deliver updates for a semi-large component-based system, consisting of around 30 components: the ASF+SDF Meta-Environment [4]. All built artifacts were put under Subversion, as described in the previous section. As expected, the repository did not grow exponentially, although all 40 component compositions were stored multiple times.

The ASF+SDF Meta-Environment is released and delivered using source tree composition [6]. This entails that every component has an abstract build interface based on AutConf. The prefixes passed using *--prefix* during build are known at the time of deployment so could be substituted quite safely. In order to keep binary files consistent, the prefixes passed to the build interface were supplanted with superfluous '/' characters to ensure enough space for the subtituted (user) path. This trick has not posed any problem as of yet, probably because package-based development requires that every dependency is always passed explicitly to the AutoConf generated *./configure* script.

A small Ruby script served as update tool. It queries the repository, listing all available releases. If you select one, the tree is checked out to a certain directory. After relocation the Meta-Environment is ready to use. Before any upgrade or downgrade however, the tool undoes the relocation to prevent Subversion from seeing them as "local modifications".

5.2 Release Management Requirements

The subject of lightweight application upgrade belongs to the field of software release management. In [18], the authors list a number of requirements for effective release management in the context of component-based software. I discuss each of them briefly here and show that our approach satisfies them appropriately.

Dependencies should be explicit and easily recorded Incremental continuous integration of components presumes that dependencies are declared as meta data within the source tree of the component. Thus, this requirement is satisfied.

Releases should be kept consistent This requirement entails that releases are immutable. The incremental continuous integration approach discussed in this paper guarantees this.

The scope of the release should be controllable Scope determines who is allowed to obtain a software release. The release repository presented in this paper enables the use of any access control mechanism that is provided by Subversion.

A history of retrievals should be kept Although I do not address this requirement directly, if the Subversion release repository is served over HTTP using Apache, it is easily implemented by consulting Apache's access logs.

With respect to release management the implementation of change set composition using Subversion has one apparent weakness. Since Subversion does not allow cross-repository branching it would be hard to compose application releases using third-party components. However, this can be circumvented by using the Subversion dump utility that exports sections of a repository on file. Such a file can then be transferred to a different repository.

5.3 Update Management Requirements

In Section 1 I listed the requirements for application upgrade from the user perspective. Let's discuss each of them in turn to evaluate whether application upgrade using Subversion satsifies them.

Lightweightness No invasive software deployment tool has to be installed to receive updates. Many language bindings exist for subversion, so self-updating functionality can be easily bundled with the application itself.

Genericity Change set composition works with files of any kind; there is no programming language dependency. Moreover, Subversion is portable across many platforms, thereby imposing no constraints on the development or user environment.

Safety The Subversion *switch* command is used for both upgrade and downgrade. A failed upgrade can thus be quickly rolled back. Another contribution to safety is the fact that Subversion repository modifications are atomic, meaning that the application user is shielded from inconsistent intermediate states, and that releases put out in parallel do not interfere.

Efficiency Efficiency is achieved on two accounts. First the use of Subversion as delivery protocol ensures that an upgrade involves the transfer of just the differences between the old version and the new version. Secondly, while the unit of delivery is a full application, only the files per component are effectively stored, and even these are stored differentially.

Although all requirements are fulfilled satisfactory, the primary weakness of binary change set composition remains the fact that distributed files have to be relocatable. Solving this problem is left as future work.

6 Conclusion and Future Work

In this paper I have discussed the requirements that have to be fulfilled so that application upgrade is a burden neither for the development side, nor for the user side. Related work in the area of software release management did not live up to these requirements. The binary change set composition technique does live up to these requirements, and can be used to deliver new application releases accurately, frequently and quickly. The implementation on top of Subversion shows that the approach is feasible and may serve as a low impact adoption path.

However, ample opportunities for future work remain. Currently, which configurations of components should be released and updated is implicit. To promote variation this should be explicitly specified. Therefore, one direction of future work is to investigate how binary change set composition could be applied in the context of component-based product populations [21]. This ways, different product variants could be automatically released and updated.

Another direction of future work concerns the integration of deployment functionality with the released application itself. Nowadays, many applications contain functionality to check for new updates. If they are available they are installed and the application is restarted. It would be interesting if using the approach of this paper one could design such "update buttons" in a reusable and generic way. Similarly, it should be investigated how such self-updating applications could be enhanced with functionality for reporting bugs or other kinds of feedback.

Finally, the notion of application state has been completely disregarded in this paper. Application state has many faces, from configuration parameters set by the user, to complete databases. The deployment method must ensure that this data is preserved across upgrades. Moreover, if an upgrade involves a change in the data format of this state, for instance, the database schema, XML schema, or grammar, then the data has to be migrated. Further research is required to see if results from the areas of, for instance, schema evolution [13] or data synchronization [8] can applied in this context. In this scenario the data should not only be preserved but converted to the new format. It is as of yet unclear how to do this in a sufficiently generic way.

References

1. AutoConf. Online: `http://www.gnu.org/software/autoconf`.
2. AutoPackage. Online: `http://www.autopackage.org`.
3. E. C. Bailey. *Maximum RPM. Taking the Red Hat Package Manager to the Limit.* Red Hat, Inc., 2000. Online: `http://www.rpm.org/max-rpm`.

4. M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.
5. Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. *Version Control with Subversion*. O'Reilly Media, 2004. Online: `http://svnbook.red-bean.com/`.
6. M. de Jonge. Source tree composition. In Cristina Gacek, editor, *Proceedings: Seventh International Conf. on Software Reuse*, volume 2319 of *LNCS*, pages 17–32. Springer-Verlag, April 2002.
7. E. Dolstra, M. de Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In Lee Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, pages 79–92, Atlanta, Georgia, USA, November 2004. USENIX.
8. J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. In *Database Programming Languages (DBPL)*, August 2005.
9. M. Fowler and M. Foemmel. Continuous integration. Online: `http://www.martinfowler.com/articles/continuousIntegration.html`.
10. E. Grossman. An update on software updates. *ACM Queue*, March 2005.
11. Richard S. Hall, Dennis Heimbigner, and Alexander L. Wolf. A cooperative approach to support software deployment using the software dock. In *Proceedings of the 1999 International Conf. on Software Engineering (ICSE'99)*, pages 174–183, New York, May 1999. Association for Computing Machinery.
12. S. Jansen, G. Ballintijn, and S. Brinkkemper. A process framework and typology for software product updaters. In *9th European Conference on Software Maintenance and Reengineering (CSMR)*, 2005.
13. Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005.
14. C. Lüer and A. van der Hoek. JPloy: User-centric deployment support in a component platform. In *Second International Working Conference on Component Deployment*, pages 190–204, May 2004.
15. FreeBSD Ports. Online: `http://www.freebsd.org/ports`.
16. G. Noronha Silva. *APT HOWTO*. Debian, 2004. Online: `http://www.debian.org/doc/manuals/apt-howto/index.en.html`.
17. Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2nd edition, 2002.
18. André van der Hoek and Alexander L. Wolf. Software release management for component-based software. *Software—Practice and Experience*, 33(1):77–98, 2003.
19. Tijs van der Storm. Continuous release and upgrade of component-based software. In Jim Whitehead and Annita Persson Dahlqvist, editors, *Proceedings of the 12th International Workshop on Software Configuration Management (SCM-12)*, 2005.
20. Tijs van der Storm. The Sisyphus continuous integration system. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE Computer Society Press, 2007. To Appear.
21. Rob van Ommering. Configuration management in component based product populations. In *Proc. of the 10th Intl. Workshop on Software Configuration Management*, May 2001.