Package-Based Software Development

Merijn de Jonge
Department of Mathematics and Computer Science,
Eindhoven University of Technology, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
M.de.Jonge@tue.nl

Abstract

The main goal of component-based software engineering is to decrease development time and development costs of software systems, by reusing prefabricated building blocks.

In this paper we focus on software reuse within the implementation of such component-based applications, and on the corresponding software development process. As it turns out, achieving effective reuse practice between the components of a single application and between the components of multiple applications has serious effects on the complexity of the software development process.

While software reuse demands for separation (of reusable blocks), the software development process demands for integration. Consequently, achieving optimal software reuse practice and an efficient development process are goals that seem hard to unite.

In this paper we discuss how these goals can be combined. We propose techniques that promote fine-grained software reuse across system, group, and institute boundaries, as well as integrated development of component compositions. The paper describes these techniques and demonstrates how they can be used in practice.

1 Introduction

Component-based software engineering (CBSE) aims at accelerating software development and decreasing development costs, by building software systems from prefabricated building blocks (components). According to the definition of Szyperski [16], components are binary, independently deployable building blocks, that can be composed by third parties. At the heart of CBSE lies software reuse, i.e., the use of existing artifacts for the construction of software.

In this paper we focus on the implementation and development process of such components. The central question is: how to establish effective reuse practice between the components of component-based software systems?

The goal of our research is to develop techniques that can improve the practice of software reuse within the components that form component-based systems. We are not only interested in software reuse within a single application, but (especially) in software reuse across distinct applications. Furthermore, we are interested in extending the reuse scope from single product/group reuse to multi product/group reuse, crossing organization boundaries.

Improving reuse practice demands fine-grained reuse [9]. That is, if a component contains potential reusable functionality, then that functionality should not be combined with other functionality, but constitute a separate reusable component. Composition with other functionality should be delayed in the development process. Thus, from a reuse perspective: ‘composition comes last’.

However, as we will see in Section 2, the complexity of the development process of a software system increases with the number of components involved. Typical development tasks, such as integration testing, combined development of different parts of a system (i.e., *crosscutting development*), and software building, then become complicated activities. Obviously, the software development process benefits from a small number of components, each offering much functionality. Thus, from a development perspective: ‘composition comes first’.

Both contradicting demands make achieving an optimal reuse practice hard, because, with current development techniques, they cannot be combined. This paper offers techniques that combine fine-grained software reuse and integrated component development. As a result, software reuse is promoted due to many small-sized components, without affecting the software development process, which still permits components to be developed together, rather than in isolation.

The key ingredients of our approach are: i) *source tree composition*, a technique to assemble the source modules of components, and ii) *integrated package development*, which allows multiple components to be developed simultaneously in the context of different component-based software

systems. These techniques together form an instantiation of the CBSE paradigm, which we called called 'package-based software development'.

The paper is structured as follows. In Section 2 we motivate the need for package-based software development. In Section 3 we describe the ideas and concepts of source tree composition. In Section 4 we address automated source tree composition. In Section 5 we discuss the package development cycle, and we explain how development of multiple packages can be combined. In Section 7 we demonstrate how package-based software development works in practice. Finally, in Section 8 we summarize our approach and discuss related work.

2 Motivation

The implementation of a software system is most often contained in a set of source files, which are structured in a directory hierarchy. It is common practice to use a Software Configuration Management (SCM) system to keep track of changes made to these files, and to manage different variations of the software system. This structuring of source files in directories, and the use of SCM systems during development, influences the reusability of software, as well as the complexity of the software development process. Below we will discuss these effects.

Software development is simplified when the development processes of the subparts of a software system are integrated, rather than isolated. This is because activities like testing, debugging, software building, and configuration management do not need to be performed separately for each subsystem, but can be performed on a complete system. Structuring the implementation of all subsystems in a single directory hierarchy (using a single SCM system) is therefore usually the preferred approach when developing a software system.

However, a source file that is contained in one directory hierarchy is difficult to reuse in another, without having to reuse *all* files contained in that directory hierarchy. For software reuse it is therefore preferable to organize all reusable parts separately. This reduces coupling between parts resulting from implicit dependencies (such as directory references) and from tangling of subsystems (for instance due to combined build actions and compile-time configuration).

If, for the sake of reuse, the implementation of a system is structured in separate directory hierarchies (using multiple SCM systems), the development process gets complicated. For instance, how can we easily run integration tests after making a change in some of these hierarchies? How to apply and keep track of changes across these hierarchies (so called *crosscutting changes*)? How to keep track of which source files constitute a system? How to build/compile the system from its source files?

Things get more complicated if we consider developing multiple software systems. How to prevent that a system breaks when some shared source files are being developed for the sake of another system? How to assemble, build, and test an arbitrary number of systems? How to define the exact ingredients of each software system? How to manage different versions of source files effectively?

Reuse across group and institute boundaries further complicate the software development process. How to develop source files simultaneously by different groups? How to reuse source files without access to a SCM system? How to assemble systems when their ingredients are spread across multiple organizations? How to assemble systems from source files, developed at different institutes using different development tools?

Thus, reuse across system, group, and institute boundaries, complicates the development process of component-based systems significantly. On the one hand, this is because it demands that the implementation is organized in separate directory hierarchies, on the other hand because it requires organizational flexibility in structuring SCM systems and in providing access to source files.

Package-based software development strives to improve this situation by: i) introducing the notion of *source code components* (i.e., building blocks at the source code level in the form of directory hierarchies); ii) using source tree composition as accompanying composition technique, which merges multiple source code components into single directory hierarchies; iii) distinguishing deployment and development mode of components; iv) proposing an explicit release management for deployed components; v) allowing combined development of components in development mode vi) allowing multiple SCM systems to be jointly used.

The main effects of these techniques are that i) fine-grained software reuse is promoted; ii) development of components can take place in the context of real software systems; iii) multiple SCM systems can be jointly used, wherever they are located; and iv) access to a SCM system is only required when developing a component.

3 Source tree composition

Source tree composition [8] is a key ingredient of package-based software development. In this section we discuss techniques and concepts of source tree composition.

3.1 Description

We define a *source tree* as a directory hierarchy containing *all* source files of a software (sub) system. It contains implementation files, files containing build instructions (such as Makefiles), configuration files (such as `Autoconf` configuration scripts), documentation etc.

We define a *source code component* as a source tree with a standardized *build interface* and a standardized *configuration interface*. A build interface defines the steps needed to build/compile the component. A configuration interface defines compile-time variability of the component.

Source tree composition is a systematic technique to assemble source trees from reusable source components. Source tree composition yields a self-contained source tree, with integrated build and configuration processes (we call this a software *bundle*). Self-containteness of a source tree means that all required source code is included. That is, the reused source components are included, as well as the source components that are required by them. The integrated build and configuration processes serve to combine the individual build/configuration processes of all included components. Hence, it requires only a single build action to build all the individual components together. Likewise, configuration of the complete system is driven by a single configuration process.

The primary goal of source tree composition is to promote source code reuse. This is achieved by:

- Combining the advantages of separate components (reuse) and of monolithic components (ease of use). Source tree composition provides a systematic way of assembling (small) reusable components to form large software systems. On the one hand, the reused components remain reusable outside the system, on the other hand, the system can be managed as a unit (i.e., system configuration, building, and deployment can be performed for the complete system, rather then for components individually).

- Promoting fine-grained software reuse. By making it easy to manage large amounts of components, source tree composition promotes development of small reusable components, which, in turn, help to prevent code duplication [9].

- Promoting collaborative software development. By making it easy to integrate third-party software, developing software collaboratively becomes more attractive. Standardized build and configuration interfaces, and a systematic composition technique help to accomplish this.

- Explicitly defined software dependencies. Source tree composition requires that dependencies between source code components are defined explicitly (see below). This prevents entangling of source components and therefore increases a component's reusability.

- Being portable across SCM systems. In practice, software is often entangled in a specific SCM system [14, 1]). Standardized build and configuration interfaces, and packages as unit of distribution (see below) help to increase a component's reusability by removing such dependencies.

3.2 Concepts

Source tree composition is a technique to integrate source files, build processes, and configuration processes. Below we will discuss the concepts that lie at the heart of source tree composition.

Build interface Source tree composition requires that components implement a standardized build interface, consisting of *build actions*. Build actions include actions for compilation (`all`), installation (`install`), and packaging (`dist`). Not surprisingly, these actions easily map to make [6] targets. The purpose of a build interface is to standardize the process of software building to a fixed sequence of build actions. Source tree composition constructs a build process that integrates the build processes of all components in a bundle. This makes building the system as easy as building a single component. A composite build process is formed by composing the build actions for the constituent components sequentially.

Configuration interface A source component has a standardized configuration interface. The purpose of this interface is to reduce the configuration effort of multiple components by making configuration a uniform activity. A configuration interface consists of configuration parameters, which may either be dependency parameters, or variability parameters. Dependency parameters serve to explicitly define component dependencies (see below). Variability parameters serve to bind compile-time variability of a component. Configuration parameters can either be bound during source tree composition, or at compile-time.

Component dependencies Source code components may depend on other source components. Source tree composition requires that such dependencies are defined explicitly via configuration interfaces. For instance, if a component A requires (reuses) another component B, then A's configuration interface should contain a dependency parameter for component B. This parameter can be bound during source tree composition, in case B is integrated in the composite source tree, or at compile-time otherwise. Part of source tree composition is calculating the transitive closure of all reused source components. This results in a dependency graph, depicting all needed source code components (represented as nodes), and all reuse relations (represented as edges). If a dependency cannot be resolved, a dependency parameter is added to the configuration interface of the bundle, which can be bound at compile-time

```
package
identification
          name=autobundle
       version=0.5
      location=http://www.cwi.nl/~mdejonge/downloads/
          info=http://www.cwi.nl/~mdejonge/autobundle/
  description='automated source tree composition'
    keywords=reuse, component-based, composition
configuration interface
  debug    'toggle debug support'
requires
  stratego 0.7.1 with optimization=speed
  gpp 2.3
  sglr 3.7
  graphviz 1.8.6
```

Figure 1. An example package definition

Build order Due to component dependencies, the build order of a composite source tree is significant. That is, a component cannot be built unless all components it depends on have been built. Consequently, software building cannot be performed in arbitrary order. A dependency graph contains all information about component dependencies. It therefore serves to determine the proper build order, by arranging the nodes in the graph linearly during a bottom-up traversal. The composite build process that is constructed as part of source tree composition respects this order.

Component configuration If a source code component reuses another, it may enforce a particular configuration of the reused component. This way a component can be adapted for specific needs. This is accomplished by binding (some of) the parameters of a component's configuration interface. Since a single component can be reused by multiple components, multiple parameter bindings may exist for a single component. As part of source tree composition, all bindings for each component are collected and combined to form a (partial) configuration of a component. Parameters that remain unbound become part of the configuration interface of the composite source tree. These can then be bound at compile-time.

Source code packages A source code package is a versioned release of a source code component. To promote reuse across institute boundaries we propose source code packages as unit of distribution because: i) they are not dependent on an SCM system and therefore easily reusable in different organizations; ii) thanks to versioning, different versions of a component can be distinguished and can coexist. Therefore, source tree composition uses packages as unit of reuse. That is, source trees are assembled from the source components that are packed in source packages.

Package definitions Characteristics of source packages are captured in package definitions (see Figure 1). A package definition contains sections that describe a package.

The 'identification' section defines the name and version of the package, specifies the location where the package can be downloaded, and briefly describes the package by way of a short description and a list of keywords. This section also contains a URL to documentation about the package.

The 'configuration interface' section defines variability parameters. For instance, the package definition in Figure 1, defines the variability parameter 'debug' that can be bound to 'on' or 'off' to toggle generation of debug information.

The 'requires' section defines package dependencies, which correspond to reuse relations. The dependencies defined in this section constitute the dependency parameters of a component's configuration interface. Dependencies are expressed as a triple containing: i) the name of the required package, ii) its version number, and iii) a possible empty list of configuration parameter bindings. For instance, the autobundle package of Figure 1 reuses amongst others, version 0.7.1 of the stratego package. The autobundle package enforces a special configuration of this package by binding the configuration parameter 'optimization' to 'speed'.

4 Automated source tree composition

Source tree composition can be automated, such that, after selecting packages of need, a composite source tree is generated from these selected packages and those that are required by them.

4.1 Autobundle

We implemented source tree composition in the tool `autobundle`. This tool produces a software bundle containing a top-level configuration and build procedure, and a list of bundled packages with download locations.

The generated bundle does not contain the source trees of individual packages yet, but rather the tool `collect` that can collect the packages and integrate them in the generated bundle automatically. To obtain and integrate the software packages, the generated build interface of a bundle contains the build action `collect` to download and integrate the source trees of all packages.

The generated bundle is driven by `make` [6] and offers a standardized build interface. The build interface and corresponding build instructions are generated by `autoconf` [10] and `automake` [11]. The tool `autoconf` generates software configuration scripts and standardizes compile-time software configuration. The tool `automake` provides a standardized set of build actions by generating Makefiles from abstract build process descriptions. Currently we require that these tools are also used by bundled

packages, but they are not essential for the concept of source tree composition.

After the packages are automatically collected and integrated, the top-level build and configuration processes take care of building and configuring the individual components in the correct order.

4.2 Package repositories

Given a set of packages, `autobundle` synthesizes the transitive closure of all needed packages, yielding a dependency graph (see Section 3). The dependency graph is constructed according to the information contained in package definitions. In order for `autobundle` to be able to access package definitions, they are stored in central places, called *package repositories*.

A package repository is a collection of package definitions. Whenever `autobundle` encounters a package dependency (i.e., for each entry in the 'requires' section of a package definition), it accesses the package repository to read the package definition for that package.

There is no restriction on the number of package repositories to be used. If multiple repositories are used, they can be ordered using a package search path. This path defines the order in which `autobundle` searches package repositories. It enables `autobundle` to unambiguously select a package definition, in case duplicates exist.

Observe that a package repository defines an explicit *reuse scope* (i.e., every package for which the package definition can be accessed by `autobundle` is a candidate for reuse). The use of multiple package repositories allows a precise definition of the reuse scope. For instance, it allows to define reuse scopes for world-wide, institute-wide, or group-wide software reuse.

4.3 Online package bases

The package definitions in a package repository also serve to generate web-forms for online package selection and package querying from. These generated web-forms are called *online package bases*.[1]

From an online package base, a composite source tree can be obtained by first selecting the packages of need, and then pressing the 'bundle' button. Pressing this button instructs `autobundle` to produce the intended software bundle as a gzipped tar file.

Online package bases also serve to easily select particular package versions, to view package definitions, and to inspect dependency graphs. The latter are generated on request for arbitrary component compositions.

[1] For an example, see the Online Package Base, which is located at `http://www.cwi.nl/~mdejonge/package-base`.

5 Integrated package development

Automated source tree composition simplifies assembling software systems from individual source code components. It yields composite source trees containing the complete implementation of a system, as well as integrated build and configuration processes. These composite source trees are now ready for deployment: they can be packaged, transfered to a target machine, installed, and used.

Thus, thanks to source tree composition, a composite software system can be handled as a unit at deployment time. In this section we describe how a composition of components can also be handled as a unit at *development* time. As we already pointed out in Section 2, this would improve development practice of component-based software systems. This is because the development—integration—test cycle is shortened, crosscutting development is permitted, system-wide development (within development environment) is allowed, and, in case of simultaneous development of a component in multiple systems, inconsistencies can be signaled soon.

5.1 Package development cycle

Recall from Section 3, that a source code package is a unit of reuse. It is a versioned release of a piece of source code, together with a build and configuration process. Packages can be developed, maintained, and released individually. They are independent of a particular SCM system, and can be developed by different groups at different institutes.

The development cycle of an individually developed source code package can be characterized as follows:

1. Make the desired changes to a source code package;
2. Test the changed implementation (this can can be seen as a form of *unit testing*);
3. On success, increase the package's version number;
4. Release the changed package.

From this development cycle, we see that it does not include *integration testing*. That is, packages are tested in isolation, not in the context of a software system. If we do consider integration testing in a systematic way, source tree composition can be used as part of an extended development cycle:

5. Select the packages that form a target software system;
6. Bundle the selected packages and those that are required by them with `autobundle`;
7. Unpack the generated software bundle;
8. Collect the ingredients of the bundle using `collect`;

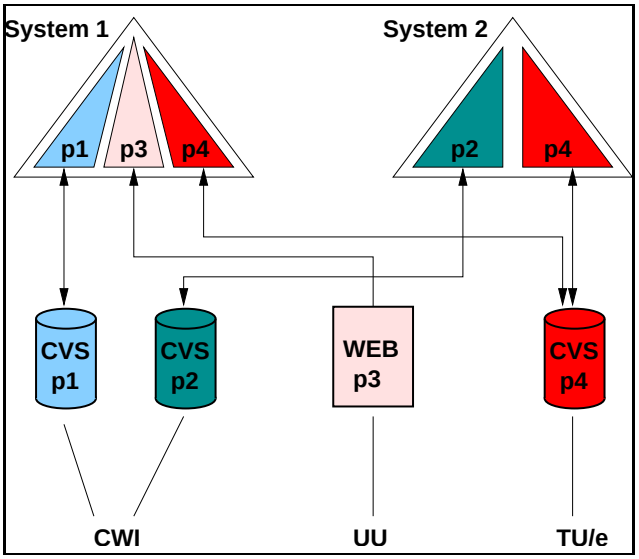

Figure 2. An example of package development in two distinct software systems.

9. Configure it using the generated configuration process;
10. Build the bundle using the generated build process;
11. Test the application to see whether the changed package functions as intended.

If multiple packages are developed simultaneously, steps 1–4 have to be performed for each of them. Steps 5–11 have to be executed for integration testing whenever one or more packages have changed.

Observe that, assuming an SCM system is in use, changes cannot be made to the composite system. This is because the composite system is assembled from source code packages only, which (by definition) are independent of an SCM system. Consequently, if integration fails, individual packages have to be modified in isolation again, by reiterating the development cycle at step 1.

The above clearly demonstrates that having truly reusable source code packages seriously complicates the development process of component-based applications.

5.2 Integrated package development cycle

The previous section demonstrated that the package development cycle, as a consequence of having truly reusable source code components, is too complex. Even with the help of automated source tree composition, which automates software assembly, the development cycle is not very practical. What's more, one could argue that integration testing should *not* be performed *after* package release, as we suggested in Section 5.1, but *before*. This suggests that the presented development cycle is not only complex, but also has serious deficiencies.

To improve this situation, we propose to collapse the development cycle of individual packages, and to integrate them in steps 5–11. That is, we propose integrated development of individual source code packages in the context of composite software systems. In the perspective of software developers, this implies that package boundaries become less visible, enabling transparent development across package boundaries.

Integration of package development cycles should fulfill the following requirements:

- Integration should preserve component structure. That is, although less visible for a developer, the structure in source components remains existent;
- Integrating multiple SCM systems should be possible;
- Easy switching between development and deployment of packages should be supported (see below);
- Collaborative software development (by multiple groups or institutes) should be supported;
- Ability to simultaneously develop a source code component in multiple software systems.

Figure 2 shows an example of integrated package development. This figure shows two systems composed of 4 different source code components (p1–p4). The components p1, p2, and p4 are obtained from an SCM system (CVS in this example). They are being developed in the context of system 1 and system 2. Component p4 is developed simultaneously in the context of both systems. System 1 also contains component p3 in the form of a released package that is obtained from a web-site. The components are developed at three different institutes (CWI, UU, and TU/e). The CWI components are developed by two different groups, each having a separate SCM system.

The key ideas of our approach can be summarized as:

- Components can be developed/maintained in the context of a (real) software system. For instance, p1, p3, and p4 are under development. They can be developed and tested in the context of system 1 and system 2;
- Component development and deployment can be combined. For instance, p1 and p4 in system 1 are under development, while p3 is being deployed;
- One can easily switch between component development and component deployment. Once made our changes to p1, verified that it operates correctly, we can release it, and use (i.e., deploy) the freshly released version. On the other hand, if p3 also needs modification, we switch to development mode, and obtain a version from its corresponding SCM system;

- Crosscutting development is easy. Because p1, p3, and p4 are under development, and merged in a single source tree, a developer can easily make changes anywhere in these components. Once he is done, changes are automatically committed to the right SCM system;
- Components can be developed simultaneously in the context of different systems. For instance, p4 is being developed in the context of both systems. If changes are made in the context of system 1, they also become available in the context of system 2. If a change to the shared component p4 in the context of system 1 breaks system 2, this can soon be discovered and solved. Such inconsistencies might be discovered much later if components were developed in isolation;
- Collaborative software development is promoted. Software systems can be assembled from packages originating from multiple SCM systems, located at different institutes. Third party software, for which no access to the corresponding SCM system exists, can also be integrated in the form of released packages.

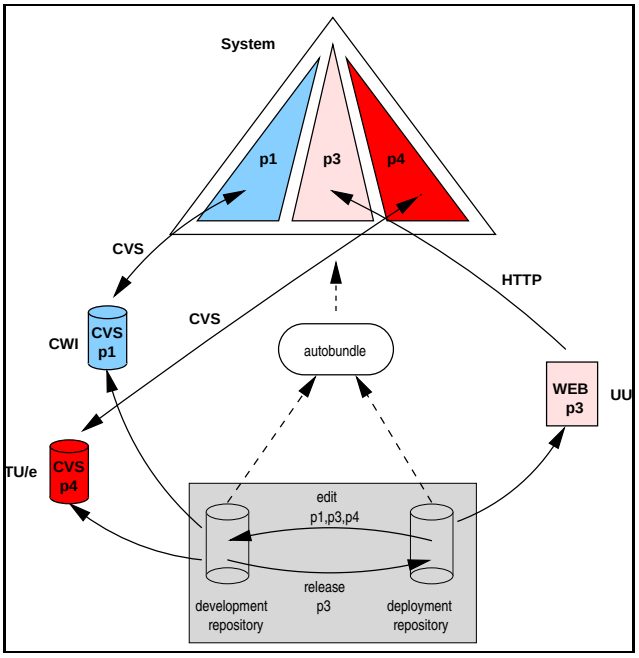

Figure 3. Architecture for integrated package development with autobundle.

6 Implementation

In order to integrate the package development cycles of multiple packages, we need to be able to assemble composite source trees from individual source packages, which may be obtained from a web site or from an SCM system. The packages may be located at different institutes and in different SCM systems. Clearly, this is a form of source tree composition which can be performed by `autobundle`. To that end, we made use of several special features of source tree composition/`autobundle`, and we made a small extension to the package definition format. In addition, we developed two extra tools for switching between development and deployment mode. These features, the modified package definition format, and the two additional tools are discussed below. Figure 3 shows the architecture for package-based software development, where `autobundle` is used to integrate package development cycles.

Supporting multiple download protocols In order to combine component deployment (by downloading packages from Internet), and component development (by obtaining the sources of a package from an SCM system, we need to support different download protocols. The standard download protocol used by `autobundle`, is the HTTP protocol. It is intended for component deployment, and can be specified in the location field of a package definition:

```
location=http://www.cwi.nl/package-base
```

In addition to the HTTP protocol `autobundle` also supports the CVS [2] protocol. This is used for component development by obtaining sources files from a CVS repository:

```
location=cvs://cvs.cwi.nl/cvs-repository
```

The `collect` tool (see Section 3), that is used to obtain and integrate the sources of a software bundle, interprets these entries, and, depending on the protocol, downloads a package from the web, or performs a 'checkout' of a head revision of a CVS repository.

It is not difficult to add support for additional protocols. Only a change to the tool `collect` is required. For instance, we recently added support for SubVersion [5] (the successor of CVS).

Multiple download locations To support component development as well as component deployment, `autobundle` needs to know the location of a package's SCM system (for development) and its download location (for deployment). In order to be able to switch between component development and component deployment, both download locations have to be captured in a package definition. To that end, we slightly extended the syntax of package definitions and introduced the notion of 'primary' and 'secondary' package locations:

```
location
  primary=http://www.cwi.nl/package-base
  secondary=cvs://cvs.cwi.nl/CVS
```

If a component is used (deployed), the primary location field points to the HTTP location, while the secondary location field points to the SCM system. If a component is

being developed, both fields are swapped: the primary field now points to the SCM system, the secondary points to the HTTP location.

Package search path As we already pointed out in Section 3, `autobundle` can use multiple package repositories. The list of repositories that `autobundle` searches for package definitions is called the 'package search path'.

In order to switch between component deployment and component development, we will use (at least) two package repositories (see Figure 3).[2] The first package repository contains ordinary package definitions for component deployment. We call it the 'deployment (package) repository'. The second repository has highest priority and is used to store package definitions for packages under development. It is called the 'development (package) repository'.

Since the development repository has highest priority, `autobundle` will search that repository first when looking for package definitions. Before switching to development mode, a package definition is copied from the deployment repository to the development repository, and both location fields in the package definition are swapped. When switching back to deployment mode, the package definition from the development repository is removed. However, if the package is released, the package definition is moved back to the deployment repository, after updating version information and swapping the location fields.

Observe that the deployment repository is only modified due to package releases. This repository may therefore be globally accessible and be used by multiple people/projects. The development repository however, is modified as a result of package development. This repository thus contains state information (which packages are under development), which makes it developer-specific. It should therefore be stored in a private location.

Tool integration We have shown how source tree composition can be used to integrate the development cycle of multiple packages. The tool `autobundle` is used to merge source trees contained in source packages. Two package repositories are used to store package definitions for deployed packages as well as for packages under development. The `collect` tool is used to obtain a component from Internet or from an SCM system. Finally, CVS is used by `collect` to obtain a package from a CVS repository.

With these tools, a developer can switch between development and deployment mode of components. To make switching more easy, we integrated these tools in two additional tools. The first tool (`edit`) turns a number of packages into development mode. The second tool (`release`) turns packages into deployment mode.

[2]More repositories can be used if desired (see Section 4.2), but in this paper we only consider two package repositories.

The `edit` tool puts a given set of packages into development mode by obtaining a head revision of the packages from the corresponding CVS repositories. To that end, `edit` performs the following tasks: i) it copies the package definitions to the development package repository; ii) it swaps the values of the two location fields in each package definition; iii) it removes the packages from the composite source tree; iv) it uses autobundle to obtain a new bundle based on the development packages; v) it uses `collect` to obtain the sources of the packages that are now under development from the corresponding CVS repositories.

Uncommitted changes from a previous development cycle are saved in a copy of a previously checked-out version (see below). If such a copy exists, `collect` will restore it and merges any changes that have been made ever since.

The `release` tool switches a set of packages into deployment mode. It performs the following actions. i) if not all changes to a package were committed to the SCM system, it makes a copy of the checked-out version, and checks a fresh version out (this ensures that only committed changes go into a new release, but also that uncommitted changes will not be lost); ii) it increases the version number of the package; iii) it makes a new distribution of the package; iv) it releases the package by uploading it to the HTTP location; v) it moves the package definition from the development to the deployment repository; vi) it uses autobundle to obtain a new bundle based on the deployment packages; v) it uses `collect` to obtain the sources of the packages from their HTTP locations.

Observe that it is possible to change a package to development mode at any time. If changes were made to a package, these are saved in a copy of the checked-out version before unpacking a released package.

7 Package development: an example

Figure 3 depicts integrated package development with `autobundle`, the deployment and development repositories, and the tools `edit` and `release`. Below we describe the development cycles that are depicted in the figure.

We start by configuring a package search path using the environment variable PACKAGE_SEARCH_PATH. Under Unix that can be done with

```
> PACKAGE_SEARCH_PATH=\
    ${HOME}/cvs-packages:\
    /opt/package-base
> export PACKAGE_SEARCH_PATH
```

Here we use `${HOME}/cvs-packages` as development repository and `/opt/package-base` as deployment repository. The `autobundle` tool is then used to assemble the initial system (System 1 in Figure 2).

```
> autobundle -p p1
```

This command creates a software bundle from the package p1, and the packages p3 and p4, which are required by p1. Initially, the development repository `${HOME}/cvs-packages` is empty. Hence, the bundle only contains deployed packages using `collect`.

The next step is to obtain and integrate the sources of these packages.

```
> ./collect
Obtaining a distribution of "p1" via HTTP
Obtaining a distribution of "p3" via HTTP
Obtaining a distribution of "p4" via HTTP
```

This yields a composite source tree and an integrated build environment for the three packages. Configuring, building, and testing of the composite system can, thanks to the generated build environment, be performed for the system as a whole, rather than for the packages separately.

Now suppose that we want to develop this system and that we need to make a crosscutting change in all three packages. To that end, we use the `edit` tool to turn these packages into development mode.

```
> edit p1 p3 p4
Obtaining p1 from CVS
Obtaining p3 from CVS
Obtaining p4 from CVS
```

As a result of this command, the packages are now obtained from CVS. For instance, if the package definition for p1 contains 'cvs.cwi.nl:/CVS' as location field, then the source code of p1 is obtained by running the command

```
cvs -d cvs.cwi.nl:/CVS checkout p1
```

Now that we have obtained the source code for the three packages, we can start developing by editing the source code anywhere in he composite source tree. All ordinary CVS commands can be used in the root of the source tree, or in any subdirectory in order to commit changes, incorporate changes made by others, undo changes etc. For instance, suppose that file 'f.c' in package p3 is modified by another developer. We can merge his changes into the composite source tree by issuing the command:

```
> cvs update
cvs update: Updating p1
cvs update: Updating p3
U p3/f.c
cvs update: Updating p4
```

This example shows that a developer can manage the source tree as a unit, while in fact he is operating on three components hosted in (possibly) three distinct CVS repositories. All CVS commands work by accessing the specific CVS repository that hosts a particular package. Hence, the structure in packages remains in tact, although developers can make changes easily anywhere across package boundaries. Furthermore, since we have all source at our disposal, development tools can be used for system-wide development. For instance, an integrated development environment can be used for doing the actual development; integration tests and code analysis can be performed on the system as a whole.

After the changes to a package have been made and have been committed to the CVS repository, the package can be released with the `release` command. For instance, to release the package p3, we can issue the command

```
> release p3
Obtaining a distribution of "p3" via HTTP
```

This command will check whether all changes made to p3 have been committed. If not, a copy of the checked-out version is saved and a fresh version is obtained from the CVS repository. This is because a release should only contain committed changes, but uncommitted modifications must not be thrown away. As pointed out earlier, the `release` command then makes a new release of the package and replaces the checked-out version by the new created release. This yields the situation as depicted in Figure 3, where p1 and p4 are being developed, while p3 is being deployed. As a consequence, changes made afterwards to p3 will no longer be committed to the CVS repository, because p3 is in deployment mode now.

If, after release, the development of p3 continues, the `edit` command is used again to switch to development mode. If a copy exists of a checked-out version of p3, due to uncommitted changes at a previous development cycle, it will be restored by `edit`. This copy is brought up-to-date by merging in the changes made to p3 after the copy was made. Thus, the uncommitted changes to p3 are not lost, and development of p3 can continue after releasing p3.

8 Concluding remarks

This paper addresses the software development process of component-based software systems as well as the practice of software reuse within the implementation of such systems. We argued that, software reuse practice affects the complexity of the software development process because of contradicting demands: while software reuse demands for separation of reusable artifacts (i.e., composition comes last), the development process demands for their integration (i.e., composition comes first).

Contributions In order to combine effective software reuse practice and efficient software development, we proposed package-based software development. Package-based software development introduces the notion of implementation-level components, called *source code components*, which are distributed in the form of SCM system-independent *packages*. Source code components help to improve reuse practice, because they offer a disciplined way to structure reusable functionality in fine-grained reusable

components. We proposed source tree composition as accompanying composition technique. It serves to assemble single directory hierarchies containing the full implementation of component-based systems. Source tree composition therefore offers the ability to benefit from integrated development of component compositions. To allow reuse across system, group, and institute boundaries, we extended source tree composition to support integration of multiple SCM systems, and combined development and deployment of source components. This allows for an integrated package development cycle of packages hosted at distinct SCM systems in the context of different software systems.

Discussion Package-based software development and the tools that support it, are still subject of research. Consequently, they suffer from several limitation, most notably: i) multiple types of SCM systems (for instance CVS and SUBVERSION) cannot be jointly used; ii) there is a need to support more SCM systems in addition to CVS and SUBVERSION; iii) the component structuring is not completely hidden for a developer, because he/she needs to be aware of which components are under development. Currently, it is the developer's responsibility to switch between deployment and development mode. Additional tool support should improve this situation in the future.

Related work SCM support for component-based software development is addressed by [12, 17]. They analyze requirements of configuration management in a component-based software development process. They focus on single SCM systems, and composition of SCM systems is therefore not addressed. Also, managing reuse across project, group, and institute boundaries is not discussed.

A promising SCM system supporting crosscutting development of components and collaborative software development across project and institute boundaries is STELLATION [3, 4]. We have plans to incorporate and use it for further research on packaged-based software development.

Software Release Management (SRM) is concerned with making software available to users [7]. SRM is concerned with software deployment only, not with combined development and deployment of components as we are. A combination of both is discussed in [15]

Koala is a software component model in which (like we) packages are used to structure source code components [13]. Koala packages are independently developed by development teams using distinct SCM systems. The model does not support composition of SCM systems, nor combined development of package compositions.

Availability `autobundle` is free software and can be downloaded from `http://www.cwi.nl/~mdejonge/autobundle/`

References

[1] M. Cagan and A. Wright. Untangling configuration management. In J. Estublier, editor, *Software Configuration Management*, volume 1005 of *Lecture Notes in Computer Science*, pages 35–52. Springer-Verlag, 1995.

[2] P. Cederqvist et al. *Version Managemetn with CVS*, 2002. Available at `http://www.cvshome.org`.

[3] M. C. Chu-Carroll. Supporting distributed collaboration through multidimensional software configuration management. In *Tenth International Workshop on Software Configuration Management (SCM-10)*, 2001.

[4] M. C. Chu-Carroll, J. Wright, and D. Shields. Supporting aggregation in fine grained software configuration management. In *Proceedings: Tenth Symposium on Foundations of Software Engineering*, pages 99–108. ACM Press, 2002.

[5] B. Collins-Sussman, B. Fitzpatrick, and M. Pilato. Subversion: The definitive guide, 2003. Available at `http://subversion.tigris.org`.

[6] S. I. Feldman. Make – A program for maintaining computer programs. *Software – Practice and Experience*, 9(3):255–265, Mar. 1979.

[7] A. Hoek van der and A. Wolf. Software release management for component-based software. *Software – Practice and Experience*, 33(1):77–98, Jan. 2003.

[8] M. de Jonge. Source tree composition. In C. Gacek, editor, *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, Apr. 2002.

[9] M. de Jonge. *To Reuse or To Be Reused: Techniques for Component Composition and Construction*. PhD thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, Jan. 2003.

[10] D. Mackenzie and B. Elliston. Autoconf: Generating automatic configuration scripts, 1998.

[11] D. Mackenzie and T. Tromey. Automake, 2001. Available at `http://www.gnu.org/manual/automake/`.

[12] H. Mei, L. Zhang, and F. Yang. A software configuration management model for supporting component-based software development. *ACM SIGSOFT Software Engineering Notes*, 26(2):53–58, 2001.

[13] R. van Ommering. Configuration management in component based product populations. In *Tenth International Workshop on Software Configuration Management (SCM-10)*. University of California, Irvine, 2001.

[14] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer*, 33(3):78–85, Mar. 2000.

[15] S. Sowrirajan and A. Hoek van der. Managing the evolution of distributed and interrelated components. In B. Westfechtel and A. Hoek van der, editors, *Software Configuration Management*, volume 2649 of *Lecture Notes in Computer Science*, pages 217–230. Springer-Verlag, 2003.

[16] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[17] L. Zhang, H. Mei, and H. Zhu. A configuration management system supporting component-based software development. In *Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, pages 25–30. IEEE Computer Society Press, 2001.