Decoupling Source Trees into Build-Level Components*

Merijn de Jonge

Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
mdejonge@cs.uu.nl

Abstract. Reuse between software systems is often not optimal. An important reason is that while at the functional level well-known modularization principles are applied for structuring functionality in modules, this is not the case at the build level for structuring files in directories. This leads to a situation where files are entangled in directory hierarchies and build processes, making it hard to extract functionality and to make functionality suitable for reuse. Consequently, software may not come available for reuse at all, or only in rather large chunks of functionality, which may lead to extra software dependencies.
In this paper we propose to improve this situation by applying component-based software engineering (CBSE) principles to the build level. We discuss how existing software systems break CBSE principles, we introduce the notion of build-level components, and we define rules for developing such components. To make our techniques feasible, we define a reengineering process for semi-automatically transforming existing software systems into build-level components. Our techniques are demonstrated in a case study where we decouple the source tree of Graphviz into 47 build-level components.

1 Introduction

Modularity is a prerequisite for component technology [17]. Already in 1972, Parnas introduced the modularization principles of minimizing coupling between modules and maximizing cohesion within modules [16]. The former principle states that dependencies between modules should be minimized, the latter principle states that strongly related things belong to the same module. These principles are well understood at the functional level for structuring functionality in functions or methods and in modules or classes.

Unfortunately, these principles are usually not applied at the build level for structuring modules and classes in directories. Often, bad programming practice like strong coupling and weak cohesion therefore move from the functional level to the build level.

In practice, many software systems therefore consist of large collections of files that are structured rather ad-hoc into directory hierarchies. Between these directories a lot of references exist (= strong coupling) and directories often contain too many files (= weak cohesion). Build knowledge gets unnecessarily complicated due to improper structuring in monolithic configuration files and build scripts.

* This research was sponsored by the Dutch National Research Organization (NWO), Jacquard project TraCE.

As a result, modules are entangled, the composition of directories is fixed, and build processes are fragile. This yields a situation where: i) potentially reusable code, contained in some of the entangled modules, cannot easily be made available for reuse; ii) the fixed nature of directory hierarchies makes it hard to add or to remove functionality; iii) the build system will easily break when the directory structure changes, or when files are removed or renamed.

To improve this situation, we can learn from component-based software engineering (CBSE) principles. In CBSE, functionality is only accessed via well-defined interfaces, and one cannot depend on the internal structure of components. Unfortunately, CBSE principles are not yet applied at the build level. Reusability of components is therefore hampered, even when CBSE principles are applied at the functional level.

For example, the ASF+SDF Meta-Environment [1] is a generic framework for language tool development. It contains generic components for parsing, pretty-printing, rewriting, debugging, and so on. Despite their generic nature and their component-based implementation, they were not reusable in other applications due to their build-level entangling in the ASF+SDF Meta-Environment. After applying the CBSE principles discussed in this paper, they became distinct components, which are now reused in several different applications [10]. Graphviz [4] is another example. It is too large, yielding too many external dependencies. In this paper we demonstrate how its implementation can be restructured such that its individual parts can be separately reused.

In this paper we discuss how to apply CBSE principles to the build level, such that access to files only occurs via interfaces, and dependencies on internal directory structures can be dropped. We also describe a composition technique for assembling software systems from build-level components. CBSE principles at the build level help to improve reuse practice because build-level components can be reused individually and be assembled into different software systems. To make our techniques feasible, we propose a semi-automatic technique for decoupling existing software systems in build-level components. We demonstrate our ideas by means of a case study, where Graphviz (300,000+ LOC) is migrated to 47 build-level components.

The paper is structured as follows. In Sect. 2 we introduce the concept of build-level components. In Sect. 3 we discuss bad programming practice and we introduce development rules for build-level components with strong cohesion and weak coupling. In Sect. 4 we discuss automated composition of build-level components. In Sect. 5 we present a semi-automatic process for decoupling software systems into build-level components. In Sect. 6 we demonstrate our ideas by means of a case study. In Sect. 7 we summarize our results and discuss related work.

2 Build-level Components

According to Szyperski [17], the characteristic properties of a component are that it: i) is a unit of independent deployment; ii) is a unit of third-party composition; iii) has no (externally) observable state. He gives the following definition of a component:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

Component-based software engineering (CBSE) is mostly concerned with execution-level components (such as COM, CCM, or EJB components). We propose to apply CBSE principles also to the build level (i.e., to directory hierarchies containing ingredients of an application's build process, such as source files, build and configuration files, libraries, and so on). Components are then formed by directories and serve as unit of composition.

Access to build-level components occurs via build, configuration, and requires interfaces. Build interfaces serve to execute actions of a component's build process (e.g., to build or install a component), configuration interfaces serve to control how a component should be build (i.e., to support build-time variability). Requires interfaces serve to bind dependencies on other components. Referencing other components no longer occurs via hard and fixed directory references, but only via the dependency parameters of a requires interface. Dependency parameters allow *late binding* by third-parties. Since all component access occurs via interfaces, build-level components can be independently deployed and their internal structure can safely be changed. Directories with these properties satisfy the component definition of [17] and can be used for build-level CBSE.

This paper is concerned with developing build-level components and with extracting such components from existing applications. Our work is based on the GNU Autotools, which serve build-time configuration (Autoconf) and software building (Automake). Strictly spoken, the GNU Autotools are not essential, but they make life much easier.

Autoconf [12] is a popular configuration script generator that produces a top-level configuration script for a software system. The script is used to instantiate Makefiles with a concrete configuration. The input to Autoconf is a configuration script in which, amongst others, configuration switches and checks can be defined. We use Autoconf because it provides a consistent way for build-time configuration (i.e., all software systems driven by Autoconf can be configured similarly). This simplifies composition of build-level components (see Sect. 3).

Automake [13] is a Makefile generator. Its input is a high-level build description from which standard Makefiles are generated conforming to the GNU Makefile Standards [3]. The benefits of Automake are that it simplifies the development of build processes, and that it standardizes the process of software building. The latter is of great importance for CBSE and the main reason that we depend on Automake. Build processes generated by Automake always provide the same set of build actions. Automake thus generates standardized build interfaces. Having standardized build interfaces enables composition of build-level components (see Sect. 3).

3 Build-level Development Rules

Many software systems break CBSE principles at the build level. This results in stronger coupling and weaker cohesion. In this section we analyze typical build-level practices that break these principles, and we provide component development rules that enable CBSE at the build level.

3.1 Component Granularity

Pitfall: Components with Other than Directory Granularity. The granularity of a component is important for its usability [17, 10]. If components are too large, then cohesion is weak. Consequently, by reusing them, too much functionality is obtained that is not needed at all. On the other hand, if components are too small then coupling will be strong and it may take too much effort to assemble a system from them.

In practice, build-level component granularity is too large. There are many examples of software systems (e.g., Graphviz, Mozilla, and the Linux kernel [7]) where potential reusable functionality is not structured in separately reusable directory hierarchies. Consequently, the complete directory hierarchy containing the implementation of a full software system has to be used if only a small portion of functionality is actually needed. Often this is not an option and reuse will not take place.

Rule: Components with Directory Granularity. Build-level components should have directory granularity, cohesion in directories should be strong, and coupling between directories should be minimal. With strong cohesion the contents of a directory forms a unit and chances are low that there is a need to only reuse a subset of the directory. Minimal coupling between directories makes directories independently deployable; an important CBSE principle. If a particular directory is not intended for individual reuse, then it can be part of a larger directory structure. This slightly increases component granularity, but prevents the existence of many small-sized components that are not actually reused individually.

3.2 Circular Dependencies

Pitfall: Circular Dependencies. If two collections of files are separated in distinct directories but reference each other, then they are strongly coupled. Although they form distinct components, they cannot be used independently because of their circular needs.

Such a decomposition into distinct directories breaks the modularization principle of minimizing coupling. Basically, circular dependencies prove that cohesion between directories is strong and that they belong together (or, at least, that they should be decomposed in another way). Circular dependencies between directories therefore, almost always indicates that something is wrong with the structure of the implementation of a software system in files and directories.

Rule: Circular Dependencies Should Be Prevented. Striving towards weak coupling forms an important motivation for minimizing circular dependencies. One solution is to simply merge circular dependent directories. If this significantly reduces cohesion then a third directory may be constructed capturing the strongly related subparts of both directories. Both directories then become dependent on the newly created one, but not the other way around.

3.3 Build Interface

Pitfall: Non-standardized Build Interfaces. There are many different build systems available, often providing incompatible build interfaces. For instance, software building with Imake, Ant, or Automake requires execution of different sequences of build

actions. These different build interfaces hamper compositionality of build-level components, because build process definitions cannot be composed transparently. The reason is that internal knowledge of a build-level component is required in order to determine which actions constitute a component's build process. This breaks the abstraction principle of CBSE which prescribes that component access only goes through well-defined interfaces.

Rule: Software Building via Standardized Build Interface. In order to make build-level components compositional, build process definitions should all implement the same build interface. This way the steps involved in the build process become equal for each component.

Implementation with Autotools. We standardize on the build interface offered by Automake. This interface includes the build actions `all` for building, `clean` for removing generated files, `install` for installing files, `test` for running tests, `dist` for building distributions, and `distcheck` for building and validating distributions.

3.4 Configuration Interface

Pitfall: Non-standardized Configuration Interfaces. What holds for build processes also holds for configuration processes. If standardization is lacking and different configuration mechanisms are in play, then configuration is not transparent, hampering compositionality. For configuration, knowledge of the component is then needed to determine the configuration mechanism used. Again, this breaks the abstraction principle of CBSE because access to the component outside its interfaces is inevitable.

Rule: Compile-time Variability Binding via Standardized Configuration Interface. Standardization of variability interfaces is needed to improve compositionality. Only then it becomes transparent how to bind compile-time variability of varying compositions of build-level components.

Implementation with Autotools. In this paper we standardize on the configuration interface offered by Autoconf. This provides a standard way for binding configuration and dependency parameters. For example, to turn a feature `f` on and to bind a parameter `p` to `some_value`, an Autoconf configuration script can be executed as `./configure --enable-f --with-p=some_value`. By using Autoconf a component can be configured individually, as well as in different compositions, and it is always clear how its variability parameters can be bound.[1]

3.5 Requires Interface

Pitfall: Early Binding of Build-level Dependencies. A composition of directories is often specified in source modules or in build processes. For instance, consider the C fragment `#include "../bar/bar.h"` from a hypothetical component `foo`. This fragment clearly defines a composition and, consequently, increases coupling between

[1] Observe that, Autoconf is not strictly necessary because a similar configuration interface (with the same commands and syntax) can be obtained in other ways as well.

`foo` and another component `bar`. The component `bar` is a build-level dependency of `foo`. The composition expressed in the C fragment is therefore a form of *early-binding*. Early binding of dependencies increases coupling, prevents independent deployment, and third-party binding.

Rule: Late Binding of Build-level Dependencies via Requires Interface. References to directories and files should be bound via *dependency parameters* of a component's *requires interface*. This is a form of late binding that allows third parties to make a composition, and that caters for different directory layouts.

Implementation with Autotools. With Automake and Autoconf this can be achieved by defining separate configuration switches for each required component. For instance, component `foo` can define a configuration switch for its dependency on `bar` as follows:

```
AC_ARG_WITH(--with-bar, [...], BAR=${withval})
AC_SUBST(BAR)
```

In the Makefile of `foo` the variable `BAR` is then used to reference the `bar` component. Dependency parameters are bound at configuration time.

3.6 Build Process Definition

Pitfall: Single Build Process Definition. If build knowledge of a composition is centralized (e.g., in a top-level Makefile), then coupling between components is increased and the components cannot easily be deployed individually. This is because build knowledge for a specific component needs to be extracted, which is difficult and error prone. Unfortunately, single build process definitions are common practice.

Rule: Build Process Definition per Component. Build-level components need individual build process definitions. This way a component can be built independently of other components and, consequently, be part of different compositions. There are many ways to define an individual build process. For instance, it can be defined as a batch file, a traditional Makefile, or as an Automake Makefile. In this paper we use Automake Makefiles.

3.7 Configuration Process Definition

Pitfall: Single Configuration Process Definition. It is common practice to centralize build-time configuration knowledge of software systems. Inside the files that capture this configuration knowledge, it is usually not clear which configuration parameters belong to which directory, and which parameters are shared. This form of coupling hampers reuse because components cannot be deployed individually without this knowledge and because extracting component-specific configuration knowledge is difficult.

Rule: Configuration Process Definition per Component. A software build process often contains numerous build-time variation points. To allow independent deployment, the configuration process, in which such variation points are bound, needs to be independent of other build-level components (only when generated from individual ones, a single configuration process definition is acceptable). To that end, each build-level component should have an independent configuration process definition. In this paper we use Autoconf configuration scripts.

3.8 Component Deployment

Pitfall: Using a Configuration Management System for Component Deployment. Putting a software system under control of a Configuration Management (CM) system can increase coupling. The reason is that the directory structure (and thus the composition of directories) is stored in a CM system and that it is not prepared for individual use. It is therefore often not easy to obtain subparts from a CM system. Furthermore, it is not easy to make different compositions of directories controlled by a CM system. For instance, CVS requires administrative actions to make a particular composition. This is required for each composition. Finally, composition of different CM systems (for instance CVS with SubVersion) is, to the best of our knowledge, not possible with current technology.

Rule: Component Deployment with Build-level Packages. To allow more wide-spread use of build-level components, they should be deployable independently of a CM system. To that end, release management [5] is needed to make components available without CM system access. Release management should include a version scheme that relates component releases to CM revisions. In the remainder of this paper we call such a versioned release of a build-level component a *build-level package* (or *package* for short).

Implementation with Autotools. The combination of Autoconf and Automake already provides support for software versioning and for generating versioned software releases. Each build-level component is given a name and a version number in the Autoconf configure script. Automake provides the build action `dist` to make a versioned release. The `distcheck` action serves to validate a distribution (e.g., to check that no files are missing in the distribution).

3.9 Component Composition

Pitfall: Making a Composition by Hand. Most software systems are manual compositions of directories, files, build processes, and configuration processes. Unfortunately, it is difficult to define a configuration process for a composite system (the complexity of configuration processes of several existing software systems demonstrate that it is no sinecure[2]). It is also difficult to correctly determine all software dependencies, and to define a composite build process. Finally, build and configuration processes are often hard to understand. These difficulties make the composition process time consuming and error prone. In addition, the composition process is hard to reproduce, and changing a composition, by adding new directories or removing existing ones, is costly. This situation gets worse when the number of components increases.

Rule: Automated Component Composition. Since it is expected that the composition process needs to be repeated (because compositions are subject to change), a need exists to keep the composition effort to a minimum. Automated component composition is therefore a prerequisite to achieve effective CBSE practice at the build level. Automated composition makes it easy to reuse components over and over again in different compositions and to manage the evolution of existing compositions over time.

[2] For instance, Graphviz contains 5,000 LOC related to build-time configuration, Mozilla more than 8,000.

```
package
identification
          name=dotneato
       version=1.0
      location=file:///home/mdejonge/graphviz/dist
           info=http://www.graphviz.org
  description='Graphviz dotneato package'
    keywords=graph, visualization, transformation
configuration interface
  dmalloc    'use dmalloc for debugging memory use'
  efence     'use efence for debugging memory use'
requires
  cdt 0.95
  gd 2.0
  graph 1.1 with optimization=nocycles
  pathplan 2.0
```

Fig. 1. A package definition in PDL.

4 Automated Build-level Composition

Building and configuring components, which are developed according to the rules of Sect. 3, can be performed solely via build and configuration interfaces. This property allows for automated build-level composition.

To enable automated build-level composition, we developed the package definition language (PDL) to formalize component-specific information [8]. A package definition serves to capture component identification information, to define variability parameters in a configuration interface, and to define dependency parameters. An example package definition is depicted in Fig. 1.

Build-level composition is based on component releases (packages). Hence package dependencies are expressed as name/version tuples and package locations (defining where packages can be retrieved from) are expressed as URLs. Package dependencies may contain parameter bindings. For instance, the package definition in Fig. 1, binds the parameter optimization to nocycles. Package definitions are stored in package repositories.

The information stored in package definitions is sufficient to automate the composition process. This process is called Source Tree Composition [8] and consists of i) resolving package dependencies; ii) retrieving and unpacking packages; iii) merging the build processes of all components; iv) merging the configuration processes of all components. The result of source tree composition is a directory hierarchy containing the build-level components (according to a transitive closure of package dependencies), and a top-level build and configuration process. Typical deployment tasks, such as building, installing, and distributing can be performed for the composition as a whole, rather than for each constituent component separately. Hence, the composite software system can be managed as a single unit.

1. Source tree analysis
   - Find components
   - Find component references
   - Fine-tune
2. Source tree transformation
   - Create components
   - Create package definitions
   - Fine-tune
3. Online package base creation

Fig. 2. The three phases of source tree decoupling.

Package repositories can be put online in the form of Online Package Bases.[3] An online package base serves as component repository from where people can select build-level components of interest. Then, by simply pressing a button, a composite software system is automatically produced from the selected components.

We have implemented automated source tree composition in the tool set Autobundle.[4] In [9] we discuss how source tree composition can be used to integrate component development and deployment. This improves software reuse practice and provides an efficient development process for build-level CBSE.

5 Migration to Build-level Components

The development rules for component development of Sect. 3 and the composition technique presented in Sect. 4, bring CBSE principles to the build level. Together they allow development of software in separate reusable components and their composition in multiple software systems. Although build-level CBSE seems promising, adapting existing software forms a barrier that stands in the way of adopting the techniques presented thus far. The question that comes into mind is: can't we reengineer existing software systems into build-level components automatically?

To that end, we present a semi-automatic technique for applying the development rules of Sect. 3 to existing software. Fig. 2 depicts the three-phase process for decoupling source trees into build-level components. This reengineering process analyses the structure of a source tree to determine candidate components, and Makefiles to determine component references. This information is used to split the source tree into pieces, and to generate component-specific Makefiles and configure scripts. Below we discuss the process in more detail.

5.1 Source Tree Analysis

We assume that source code is structured in subdirectories. A root directory only contains non-code artifacts (including build knowledge). If all sources were contained in a

[3] `http://program-transformation.org/package-base`
[4] `http://www.cs.uu.nl/~mdejonge/software`

single directory, then some additional clustering techniques can be used to group related files in directories.

Finding Components. The structure of a source tree in directories determines the set of build-level components. Consider Fig. 3, where nodes denote directories, edges directory structure, and arrows directory references. Basically, there are two approaches: i) each non-root directory constitutes a separate component (i.e., *a*, *b*, *c*, *d*, and *e* form components); ii) each directory hierarchy below *root*, without external references to its subdirectories constitutes a component (i.e., *abc*, *d*, and *e*). In the first approach, each directory is a candidate for potential reuse. This leads to fine-grained reuse but also to a large number of components. In the second approach, actual reuse information serves to determine what the candidates for reuse are. Since nodes *b* and *c* in Fig. 3 are not referenced outside the tree rooted at *a*, both are considered *not* reusable. This reduces the number of components, but also results in more coarse-grained reuse. In this paper we follow the first approach.

Finding References. Directory references serve to determine component dependencies. That is, if a directory reference from *a* to *b* exists and *a* and *b* become separate components, then *b* becomes a dependency of *a*.

Directory references are found by inspecting the Automake Makefiles in the source tree for directory patterns. For each directory reference found it is checked that it points to a directory inside the source tree and that the target directory contains an Automake Makefile. Thus, references outside the source tree and references to directories that are not part of the build process are discarded.

Fine Tuning. From the information that is gathered thus far, we can construct a component dependency graph that models components and their relations. This model serves as input for the transformation phase discussed below. Fine tuning consists of modifying the graph to specific needs and to repair some problems:

- Additional edges and arrows can be added to the graph, in case the analysis failed to find them all automatically.
- The component dependency graph needs to be adapted in case of cyclic dependencies. These are not automatically repaired because changing a cycle into a tree and selecting a root node cannot be done unambiguously.
- The graph can be adapted to combine certain nodes to represent single, rather than separate components.

We use DOT [4] to represent component dependency graphs. The adaptations are specified as graph transformations, which can be performed automatically. The complete analysis phase then becomes an automated process that can be repeated when needed.

5.2 Source Tree Transformation

The source tree transformation phase consists of splitting-up a source tree into build-level components, and creating package definitions for each of them. This process is driven by the information contained in the component dependency graph constructed during the first phase. In the discussion below, we assume that it contains three components, capturing the directories *abc*, *d*, and *e* of Fig. 3.

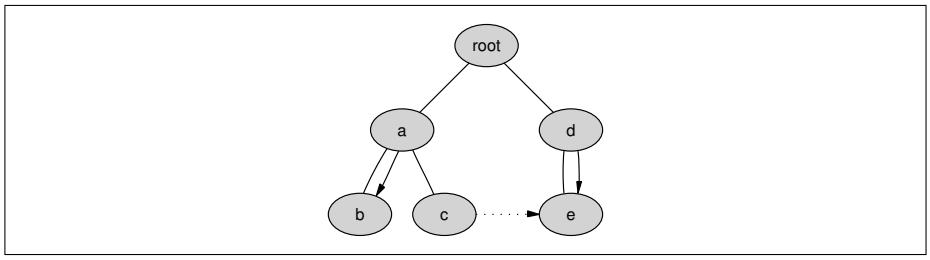

Fig. 3. A directory hierarchy with directory references represented by arrows.

Creating Components. Creating a build-level component, involves: i) isolating its implementation from the source tree; ii) creating an Autoconf configure script; iii) creating an Automake Makefile.

To isolate the implementation of a build-level component *c* from a source tree *s*, the subtree containing its implementation is moved outside *s*. If the subtree of *c* has subdirectories, which, according to the component dependency graph, belong to other components, then these subdirectories are recursively moved outside *c*. For instance, in case of Fig. 3, the subtrees rooted at nodes *a* and *d* are placed outside the source tree. Component *d* has a subdirectory *e* that forms a separate component and is therefore moved outside *d*. The subdirectories of *a* are not moved because they do not form separate components.

Component-specific Autoconf configure scripts are created from the top-level configure script. The following adaptations are made: i) The name and version of the original system are replaced by the name and version of the component; ii) References to other directories are removed. Thus, all files listed in `AC_CONFIG_FILES` that do not belong to the component are removed; iii) Configuration switches are added for each component dependency. For component *a* of Fig. 3 this means that a configuration switch for component *e* is created. The binding of this switch is accessible in Makefiles as `${E}`. The resulting Autoconf configure script is tailored for a single component: it instantiates only Makefiles of the component and it does not contain hard references to other components.

The most complex task is creating Automake Makefiles. First, this involves removing directory names for those directories that have become separate components. In particular, this means that these names are removed from the `SUBDIRS` variable. If this variable becomes empty, the variable itself is removed. Second, self-references are changed. In the original tree, the component *e* from Fig. 3 might reference itself in different ways, e.g., as `${top_srcdir}/d/e`, or `../e`. These have to be changed according to the new directory structure, e.g., in `${top_srcdir}/e`. Third, reusable files should be made public accessible in standard locations. The original source tree may contain direct file references but these are no longer allowed. For instance, in the original source tree (see Fig. 3) one can depend on the exact directory structure and access a C header file `f.h` in directory *b* as `${top_srcdir}/a/b/f.h` However, in the new situation this is not allowed because a component can only be accessed via its interfaces and one cannot depend on its internal structuring. This implies that for a file to be

accessible, it needs to be placed in a standard location. We accomplish this by replacing `noinst_HEADERS` and `include_HEADERS` variables by `pkginclude_HEADERS`. This guarantees that the header file `f.h` always gets installed in `include/a/` relative to some compile-time configurable directory. Other files, such as libraries, are made accessible in a similar fashion. Fourth, directory references are changed into component references. This implies that all file referencing goes via interfaces. For example, the file `f.h` that belongs to component *a*, can then be accessed as `${A}/include/a/f.h`. The variable `A` is bound at composition time. The result is that external references into a component's source tree no longer exist. The component can therefore safely change its internal structure when needed.

Creating Package Definitions. The component dependencies of a component are captured in an automatically-generated package definition. This package definition also contains a standard identification section, containing the name and version of the package, and the location from where it can be retrieved. In addition, a configuration interface section is constructed by collecting all configuration switches from the Autoconf configure script.

Fine Tuning. A build-level component that is the result of the procedure above, has a component-specific configuration and build process. Component dependency parameters can be bound with configuration switches. Now is the time to fine-tune the component to repair problems resulting from its isolated structure:

- Because circular dependencies are no longer allowed, the implementation of components having circular dependencies needs to be fixed. This involves restructuring files or creating new components as discussed in Sect. 3.
- The automatic source tree transformation might fail in discovering and changing all directory and file references. These can now be repaired manually.
- Software systems driven by Automake and Autoconf do not always produce complete distributions. This means that a distribution does not include all files that are referenced by its Makefiles. Build-level components inherit these errors. To make them suitable for composition, these errors must be repaired, either by removing them from the Makefiles, or by adding them to the `EXTRA_DIST` variable.

The modifications can be defined as patches, such that they can be processed automatically. This yields a fully automated transformation process. After making sure that these patches yield a component for which the `distcheck` build action succeeds, the component is ready and can be imported in a CM system for further development.

5.3 Package Base Creation

The last phase in source tree decoupling, is to make the components available for use. This implies that component distributions are created and released, and that an online package base is generated from the package definitions. Component releases are stored at the location specified in the generated package definitions. The online package base is driven by Autobundle. It offers a WEB form from which component selections can easily be assembled by pressing a single button. This makes the functionality that was first entangled in a single source tree, separately reusable.

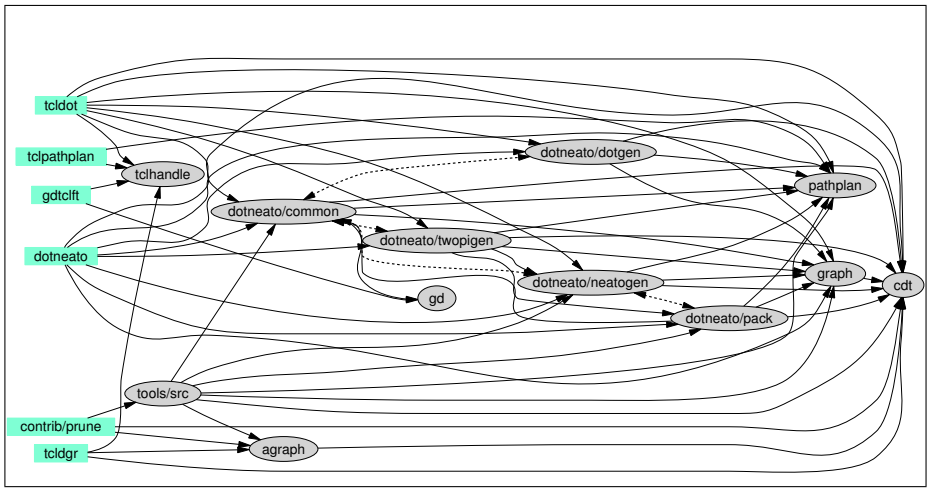

Fig. 4. Subset of Graphviz's directories and directory references.

6 Graphviz: a Case Study

In this section we discuss the application of source tree decoupling to a real-world application.

6.1 Graphviz

Graphviz[5], developed by AT&T, is an Open Source collection of tools for manipulating graph structures and generating graph layouts [4]. It consists of many small utilities that operate on the DOT graph format. Its structuring in many small utilities makes it of general use for all kinds of graph visualization and manipulation problems. This is affirmed by its adoption in a large number of software systems.

Thus, the functionality offered by Graphviz turns out to be effectively reusable. However, for many uses of Graphviz only a small subset of the tool set is actually needed (e.g., only the program `dot` might be needed for visualizing graphs). Graphviz does not support reuse of this granularity. This has two important drawbacks for software systems that are using Graphviz: i) software distributions and installations become unnecessarily large and complex; ii) it introduces several dependencies on external software, which are in fact not used.

At the build level we can make some other observations. The Graphviz distribution[6] contains 264 directories, 1,856 files, and 174 directory references. Graphviz is implemented in multiple programming languages, including C, C++, Tcl/Tk, AWK, and shell scripts. The Graphviz implementation consists of more than 300,000 lines of code. In Fig. 4 we depict a small portion of the build-level structure of Graphviz, containing

[5] `http://www.graphviz.org`
[6] Graphviz version 1.10

directories (as nodes) and directory references (as arrows). Boxes correspond to root nodes (i.e., directories to which no references exist). From this picture we can make two observations: i) the many directory references reveal that there is much reuse at the build level (each directory reference corresponds to a reuse relation from one directory to another). Despite their reusability, they are not available for reuse outside Graphviz's source tree; ii) some arrows are pointing in two directions (i.e., the dashed arrows), indicating circular dependencies between directories. As we pointed out in Sect. 3, this forms an indication for problems in the structure of Graphviz. In addition, the configuration process of Graphviz is quite complicated (i.e., more than 5,000 lines of code related to build-time configuration of Graphviz). It is therefore hard to extract from the Graphviz source tree just what is needed, and integration of Graphviz with other software is painful.

6.2 Restructuring Graphviz

Due to the aforementioned problems (i.e., Graphviz is is too large, it has too many external dependencies, its configuration process is too complex, it has cyclic dependencies, and it contains reusable functionality that is not available for external reuse), Graphviz forms a perfect candidate for applying our semi-automatic restructuring technique. Below follows a discussion of the different steps that we performed to restructure Graphviz.

Fixing Circular Dependencies. Because of circular dependencies between directories, we first had to remove the corresponding cycles from the component dependency graph. We defined this adaption as a simple graph transformation. At the end of the source tree transformation phase, we removed circular references from the generated build-level components as well. This had little impact, because they were either unnecessary and could simply be removed, or they could be solved by moving some files.

Restructuring. The component structure produced at the first migration phase was not completely satisfactory. Some components were too fine-grained and needed to be combined with others. Therefore we removed some of the nodes and edges from the component dependency graph by means of an automatic graph transformation. In some cases we had to move files between components, because they were accessed from one component but contained in another.

Repairing Makefiles. Graphviz is not prepared for rebuilding distributions. The problem is that the Makefiles contain references to files that are contained in Graphviz's CM system but not in Graphviz distributions. Consequently, building a distribution from a distribution fails because of missing files. Since build-level composition is based on packages, which are independent of a CM system by definition (see Sect. 3), the build-level components of Graphviz had to be repaired. This involved adapting the Makefiles of components such that all files referenced are also distributed.

6.3 Graphviz Components

The restructuring process yielded 47 build-level components. We automatically created releases for them and we generated a Graphviz online package base. Finally, we generated a new (abstract) package definition called `graphviz` that is depended an all

top-level components (i.e., corresponding to the boxes of Fig. 4). The corresponding composition of components is similar to the initial Graphviz source tree. This demonstrates that we can reconstruct the initial Graphviz distribution with build-level composition. In addition, we combined the Graphviz package base with additional package bases to make build-level compositions of Graphviz components and arbitrary other build-level components. This demonstrated build-level CBSE in practice.

7 Concluding Remarks

In this paper we argued that software reuse is hampered because the modularization principles of strong cohesion and weak coupling are not applied at the build level for structuring files in directories. Consequently, files with potential reusable functionality are often entangled in source trees and their build instructions hidden in monolithic configuration and build process definitions. The effort of isolating modules for reuse in other software systems usually does not outweigh the benefits of reusing the module. Consequently, reuse is not optimal or too coarse grained.

Contributions. In this paper we proposed to apply component-based software engineering (CBSE) principles to the build level, such that build-level components are accessed only via well-defined interfaces. We analyzed bad programming style, practiced in many software systems, that breaks CBSE principles. We defined rules for developing "good" build-level components. We discussed an automated composition technique for build-level components. In order to make our techniques feasible, we defined a semi-automatic process for source tree decoupling. It aims at easily migrating existing software to sets of build-level components. This process consists of a source tree analysis and a source tree transformation phase, where build-level components are identified and isolated to form individual reusable components. We demonstrated our techniques by decoupling Graphviz into 47 components.

Discussion. The most prominent shortcoming of our approach is the dependency on Autoconf and Automake. However, since these tools are so often used in practice, and because many systems are migrating to adopt them (Graphviz is a good example of this), we believe that this dependence is acceptable.

Currently, we are not able to precisely track what the configuration switches and environment checks of a component are. Consequently, the per-component generated configure scripts need some manual adaption to remove stuff that does not belong to the component. Observe however, that only information is removed; the generated components will therefore work with or without this extra information.

Graphviz was not a toy application to test our techniques with. Since it has migrated from a build system without Automake, its build and configuration processes contain several inconsistencies, as well as constructs that break Automake principles. The successful migration of Graphviz therefore strengthens our confidence in the feasibility of our techniques. Nevertheless, we look forward to apply our techniques to additional software systems.

Related work. Koala is a software component model that addresses source code composition [15]. Unlike our approach, Koala is concerned only with C source code. Component composition therefore involves composing individual C source modules and defining a sequence of compiler calls. Because it is tailored towards a single programming language, Koala has more control over the composition process at the price of less genericity. For example, adopting non-Koala components is therefore difficult.

Reengineering build and configuration processes, and decoupling source trees into components is a research topic that is not so well addressed. Holt *et al*. emphasize that the comprehension process for a larger software system should mimic the system's build process [6]. Their main concern is understanding the different pre-compile, compile, and link steps that are involved in a build process, not restructuring source code, or making build and configuration processes compositional. In [18], the notion of build-time architectural views is explored. They model build-time relations between subparts of complex software systems. They do not consider the structuring of files in directories and splitting up complex software systems in individual reusable parts.

There exist several clustering techniques that help to capture the structure of existing software systems [11]. In [2], a method is described for finding good clusterings of software systems. Such clusters correspond to use-relations (such as calling a method, or including a C header file). A cluster therefore not always corresponds to a component. In our approach we use the directory structure for clustering source code into components.

It is sometimes argued that build knowledge should not be spread across directories at all, but contained in a single `Makefile` [14]. The motivation is that only in a single `Makefile`, completeness of build dependencies can be achieved. This is merely due to limitations of traditional `make` implementations. Unless such global Makefiles are generated, they completely ignore modularization principles necessary for decomposing directory structures.

Acknowledgments. John Ellson for discussions about Graphviz and Eelco Dolstra for feedback on drafts of this paper.

References

1. M. van den Brand, A. Deursen, J. Heering, H. Jong, M. Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, 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.
2. Y. Chiricota, F. Jourdan, and G. Melançon. Software components capture using graph clustering. In *Proceedings: IEEE 11th International Workshop on Program Comprehension (IWPC'03)*, pages 217–226. IEEE Computer Society Press, May 2003.
3. Free Software Foundation. *GNU Coding Standards*, 2004. `http://www.gnu.org/prep/standards.html`.
4. E. Gansner and S. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, Sept. 2000.
5. A. van der Hoek and A. Wolf. Software release management for component-based software. *Software – Practice and Experience*, 33(1):77–98, Jan. 2003.

6. R. Holt, M. Godfrey, and X. Dong. The build / comprehend pipelines. In *Proceedings: Second ASERC Workshop on Software Architecture*, Feb. 2003.
7. M. de Jonge. The Linux kernel as flexible product-line architecture. Technical Report SEN-R0205, CWI, 2002.
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. Package-based software development. In *Proceedings: 29th Euromicro Conference*, pages 76–85. IEEE Computer Society Press, Sept. 2003.
10. 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.
11. R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings: IEEE 8th International Workshop on Program Comprehension (IWPC'00)*, pages 201–210. IEEE Computer Society Press, June 2000.
12. D. Mackenzie, B. Elliston, and A. Demaile. *Autoconf: creating automatic configuration scripts*. Free Software Foundation, 2002. Available at `http://www.gnu.org/software/autoconf`.
13. D. Mackenzie and T. Tromey. *GNU Automake Manual*. Free Software Foundation, 2003. Available at `http://www.gnu.org/software/automake`.
14. P. Miller. Recursive make considered harmful. *AUUGN*, 19(1):14–25, 1998. Available at `http://aegis.sourceforge.net/auug97.pdf`.
15. 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.
16. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972.
17. C. Szyperski. *Component Software: Beyond Object-Orientated Programming*. Addison-Wesley, second edition, 2002.
18. Q. Tu, M. Godfrey, and X. Dong. The build-time architectural view. In *Proceedings: IEEE International Conference on Software Maintenance (ICSM 2001)*, pages 398–407. IEEE Computer Society Press, Nov. 2001.