Build-Level Components

Merijn de Jonge

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 semiautomatically transforming existing software systems into build-level components. Our techniques are demonstrated in two case studies where we decouple the source tree of Graphviz into 46 build-level components and analyze the source tree of Mozilla.

Index Terms—CBSE, software component, software reuse, software construction, software engineering, source tree composition, build level.

1 Introduction

Modularity is a prerequisite for component technology [32]. Already in 1972, Parnas introduced the modularization principles of minimizing coupling between modules and maximizing cohesion within modules [30]. The former principle states that dependencies between modules should be minimized, the latter 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 to structure modules and classes in directories. Often, bad programming practice like strong coupling and weak cohesion so moves from the functional 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.

As a result, modules are entangled, the composition of directories is fixed, and build processes are fragile. This yields a situation where: 1) potentially reusable code, contained in some of the entangled modules, cannot easily be made available for reuse; 2) the fixed nature of directory hierarchies makes it hard to add or to remove functionality; 3) 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 [3] 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 using a state-of-the-art component architecture at the functional level [2], 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 [17]. Graphviz [10] and Mozilla [25] are other examples. They are too big and hide large portions of potential reusable functionality. We will discuss how their implementation can be restructured such that different 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 semiautomatic technique for decoupling existing software systems in build-level components. We demonstrate our ideas by means of two case studies. In the first, we migrate Graphviz (300,000 + LOC) to 46 build-level

- *The author is with Philips Research Laboratories, Prof. Holstlaan 4, NL-5656 AA Eindhoven, The Netherlands.*
E-mail: Merijn.de.Jonge@philips.com.

Manuscript received 1 Dec. 2004; revised 22 Feb. 2005; accepted 31 Mar. 2005; published online 26 July 2005.
Recommended for acceptance by W. Frakes and K. Kang.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0267-1204.

components. In the second, we report on an ongoing migration process of Mozilla (4,000,000 + LOC).

The paper is structured as follows: In Section 2, we introduce the concept of build-level components. In Section 3, we discuss bad programming practice and we introduce development rules for build-level components with strong cohesion and weak coupling. In Section 4, we discuss automated composition of build-level components. In Section 5, we present a semiautomatic process for decoupling software systems into build-level components. In Sections 6 and 7, we demonstrate our ideas by means of two case studies. In Section 8, we summarize our results and discuss related work.

2 Build-Level Components

According to Szyperski [32], the characteristic properties of a component are that it: 1) is a unit of independent deployment, 2) is a unit of third-party composition, and 3) 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 components), configuration interfaces serve to control component construction (i.e., to support build-time variability) and requires interfaces serve to bind dependencies on other components. Referencing other components no longer occurs via hard and fixed directory references, but only via dependency parameters of requires interfaces. These 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 [32] 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 [20] is a popular generator that produces top-level configuration scripts for software systems. The script is used to instantiate Makefiles with a concrete configuration. The input to Autoconf is a configuration description 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 Section 3).

TABLE 1
Build-Level Component Development Rules

1) Components with directory granularity.
2) Circular dependencies should be prevented.
3) Software building via standardized build interface.
4) Compile-time variability binding via standardized configuration interface.
5) Late binding of dependencies via requires interface.
6) Build process definition per component.
7) Configuration process definition per component.
8) Component deployment with build-level packages.
9) Automated component composition.

Automake [21] is a Makefile generator. Its input is a high-level build description from which standard Makefiles are generated conforming to the GNU Coding Standards [9]. 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 use 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 Section 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 (smells) that break these principles. To that end, we introduce nine build-level concepts and identify build-level smells for each of them. To bring CBSE principles to the build level, we propose build-level development rules for each concept and, if applicable, discuss their implementation with the Autotools. When applied, these development rules will turn build-level artifacts into "true" components. Table 1 lists the identified build-level development rules.

3.1 Component Granularity

Smell: Components with other than directory granularity. The granularity of a component is important for its usability [32], [17]. 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., OpenOffice [29], Mozilla [25] (see Section 7), Graphviz [11] (see Section 6), and the Linux kernel [19], [14]), where potentially 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 them 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

Smell: Circular dependencies. If two sets of files are separated in distinct directories but reference each other, then they are coupled. Although they form distinct components, they cannot be used independently due to their circular needs.

Such a decomposition into distinct directories breaks the modularization principle of maximizing cohesion. 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 indicate 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 toward 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

Smell: Nonstandardized build interfaces. There are many different build systems available, often providing different and incompatible build interfaces. They may differ both semantically and functionally. For instance, a semantical difference between the build systems Imake and Automake, is that software installation with Imake consists of the actions `install` and `install.man`, while with Automake `install` suffices. A functional difference is that Automake supports software deinstallation, while Imake does not. Software building with Imake or Automake thus requires execution of different sequences of build actions. Other build systems, such as Ant, require yet another sequence of build actions. In case of Ant, not even a standard set of build actions exists. Clearly, as a result of different build interfaces, software building is not standardized. 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.

TABLE 2
Standardized Build Interface According to GNU Coding Standards [9]

Action	Description
`all`	Build all build targets
`clean`	Remove (intermediate) build targets
`install`	Install build targets
`uninstall`	Remove installed build targets
`check`	Build and execute registered tests
`dist`	Build a software distribution
`distcheck`	Build and test a distribution

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 sequence of steps involved in the build process becomes equal for each component, and composition of build processes becomes transparent. We follow the GNU Coding Standards [9] and standardize on the build interface depicted in Table 2.

Implementation with Autotools. All build processes driven by Automake automatically implement the build interface of Table 2. By standardizing the build interface, we know that a component's build process consists of the sequence of build actions `clean`, `all`, and `install` (of which the first action is optional). Furthermore, we know that a distribution can be constructed with the `dist` action.[1]

3.4 Configuration Interface

Smell: Nonstandardized 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. Standardized configuration interfaces are needed to improve compositionality. Only then does it become transparent how to bind compile-time variability of varying compositions of build-level components. In this article, we propose the standardized configuration interface of Table 3 which follows the GNU Coding Standards [9].

Implementation with Autotools. The configuration interface of Table 3 was inspired by Autoconf. Autoconf always generates configuration processes that implement this standardized configuration interface.[2] Thus, by using

1. Observe that Automake is not strictly necessary because the build interface can also be implemented in other ways.
2. Observe that Autoconf was not really designed to support arbitrary variation points other than binary switches and dependency parameters.

TABLE 3
Standardized Syntax for Variability Parameter Binding via Configuration Tool, According to GNU Coding Standards [9]

--help	Show configuration switches
--prefix=<p>	Install software in <p>
--disable-<f>	Turn off feature <f>
--enable-<f>	Turn on feature <f>
--with-<f>=<v>	Bind feature <f> to value <v>

Autoconf, we automatically obtain 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.[3]

3.5 Requires Interface

Smell: 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 from a hypothetical component `foo`:

```
#include "../bar/bar.h"
```

This fragment clearly defines a composition and, consequently, increases coupling between `foo` and another component `bar`. This composition can also be specified in a Makefile, e.g.:

```
foo.o : foo.c cc -c foo.c -I ../bar
```

The component `bar` is a build-level dependency of `foo`. The composition expressed in the C fragment and in the Makefile 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. Due to this late binding time of dependency parameters, the concrete composition of components and the structuring in directories need not be known before configuration time. Consequently, components cannot depend on a fixed directory layout or composition and should access components only indirectly via the dispatch mechanisms of Makefile variables.

Implementation with Autotools. With Automake and Autoconf, this can be achieved by defining separate configuration switches for each required component. Observe that, conceptually, Autoconf does not distinguish between dependency and variability parameters. Both have to be declared as configuration switches. For instance, component `foo` can define a configuration switch for its dependency on `bar` as follows:

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

The set of configuration switches for binding dependency parameters forms a component's requires interface. Dependency parameters are bound at configuration time. For instance, the parameter for `bar` can be instantiated as follows:

```
configure -with-bar=/opt/components/bar
```

This will bind variable `BAR` to `/opt/components/bar`. In the Makefile of `foo` the variable `BAR` is then used to reference the `bar` component:

```
foo.o: foo.c
    cc -o foo.o foo.c -I$(BAR)/include
```

Observe that it is assumed that build results are stored in standard locations (see below).

3.6 Build Process Definition

Smell: 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.

Referencing components from Makefiles occurs via dependency parameters which are instantiated at configuration time. For instance, component `bar` can be referenced in the Makefile of `foo` via the variable `BAR`. Observe that build-level components may produce multiple build results, such as header files, libraries, and so on. These cannot be referenced individually because the variable `BAR` points to the directory structure with *all* `bar`'s build results, not to individual files. Consequently, we need agreement on where the individual build results are stored in order to be able to reference each of them. Basically, there are three approaches possible:

- All build results are stored in a single directory. This makes referencing easy because everything can be found in this single directory. However, shared data (such as documentation and include files) cannot nicely be separated from platform-specific data (such as libraries and binaries), and name clashes are more likely to occur.
- Build results are scattered across arbitrary directories. For instance, `bar` may consist of two libraries (`libbar1` and `libbar2`), which have separate

3. 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.

directories for storing build results. This approach makes component referencing difficult because it requires knowledge of the internal structuring of a component.

- Different types of build results are stored in a standard set of separate directories. This way, no internal knowledge of the structure of a component is needed because it is always clear where data is stored. Moreover, data that can be shared between platforms can be nicely separated from data that cannot be shared.

We follow the third approach and store components in a unique prefix with binaries in the sub directory `bin`, libraries in `lib`, header files in `include`, etc.[4] Given a binding `b` for a component, a library from that component can thus be accessed in the `lib` directory relative to `b`. For example, if component `bar` is stored in `/opt/components/bar`, then the library `libbar.a` can be accessed in the directory `/opt/components/bar/lib` and the header file `bar.h` in `/opt/components/bar/include`.

Implementation with Autotools. This approach fits nicely in the Autotools model, where libraries declared with `lib_LIBRARIES` are installed in `lib`, header files declared with `include_HEADERS` in `include`, and programs declared with `bin_PROGRAMS` in `bin`. To reference the component `bar`, the Makefile of `foo` will typically specify a library and include file search path as follows:

```
foo_CPPFLAGS = -I$(BAR)/include
foo_LDFLAGS = -L$(BAR)/lib
```

3.7 Configuration Process Definition

Smell: 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.

For example, consider the declaration of the `--with-stack-size` configuration switch in an Autoconf configuration process definition:

```
AC_ARG_WITH(
  [stack-size],
  AS_HELP_STRING(...),
  [STACK_SIZE=${withval}])
```

It is unclear to which directory this switch corresponds or whether it is used by multiple directories. Consequently, all directories become coupled with this configure script. Not only configuration switches, but also environment checks increase coupling. For instance, the Autoconf macro `AC_PROG_CC` is used to determine the compiler to be used on a build platform. It is expected that most, if not all, directories will depend on this environmental check and so become coupled to the configuration tool. Finally, configure scripts define the set of (Make) files to be generated after configuration:

```
AC_CONFIG_FILES(
  [Makefile foo/Makefile bar/Makefile])
```

This makes the build processes of all directories dependent on the configuration tool, because compilation cannot be done unless a proper Makefile is present. All directories thus depend on the top-level configuration tool and cannot independently be deployed. Furthermore, this top-level configuration tool defines a composition preventing third-party composition. Finally, the configuration tool needs to know the internal structuring of all directories because it needs to know which files to generate.

There are several other configuration mechanisms in use today and almost all of them lack proper modularization and abstraction mechanisms needed for proper decomposition of directory hierarchies.

Rule: Configuration process definition per component. A 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). Build-level components should therefore have independent configuration process definitions.

Implementation with Autotools. In this article, we use Autoconf configuration process definitions. Each build-level component will have a separate Autoconf configuration process definition. These definitions will have only local file references. External references are made via dependency parameters of the component's requires interface. As a result, the configuration process definition only needs to know about the component's internal structure.

3.8 Component Deployment

Smell: Using a configuration management system for component deployment. Putting a software system under control of a Software Configuration Management (SCM) system can increase coupling. The reason is that the directory structure (and thus the composition of directories) is stored in an SCM system and that it is not prepared for individual use. Obtaining subparts from a SCM system, or making different directory compositions, is therefore difficult. For instance, making a particular composition in CVS requires administrative actions because components are not first-class in CVS. This is required for each composition. Finally, composition of different SCM 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 wide-spread use of build-level components, they should be deployable independently of an SCM system. To that end, release management [12] is needed to make components available without SCM system access. Release management should include a version scheme that relates component releases to SCM revisions. In the remainder of this article, we call such a versioned release

4. According to the GNU Coding Standards [9] and the Filesystem Hierarchy Standard (FHS) [31].

(or distribution) of a build-level component a *build-level package* (or *package* for short).

Implementation with Autotools. The combination of Autoconf and Automake provides support for generating versioned software releases. The name and version of a build-level component are declared 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). The Autotools thus make decoupling from an SCM system and constructing software distributions almost trivial. This forms a strong motivation for using Autotools for build-level components.

3.9 Component Composition

Smell: 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[5]). 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.

Nevertheless, due to lacking composition technology, build-level compositions are mostly constructed by hand. Consequently, build-level artifacts are coupled and do not come available for individual reuse.

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.

4 Automated Build-Level Composition

Building and configuring components, which are developed according to the rules of Section 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 [15]. 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.

```
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
   efence   'use efence for debugging memory use'
   mylibgd  'use internal gd library instead of an installed one'
requires
   cdt      0.95
   gd       2.0
   graph    1.1  with optimization=nocycles
   pathplan 2.0
```

Fig. 1. A package definition in PDL.

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 [15] and consists of

1. resolving package dependencies,
2. retrieving and unpacking packages,
3. merging the build processes of all components, and
4. 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.

Package repositories can be put online in the form of Online Package Bases.[6] An online package base serves as a 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 implemented automated source tree composition in the toolset Autobundle.[7] In [16], we discuss how to integrate source tree composition with component development and deployment. This improves software reuse practice and provides an efficient development process for build-level CBSE.

5. For instance, Graphviz [11] contains 5,000 LOC related to build-time configuration, Mozilla [25] more than 16,000.

6. www.program-transformation.org/package-base.

7. www.program-transformation.org/Tools/AutoBundle.

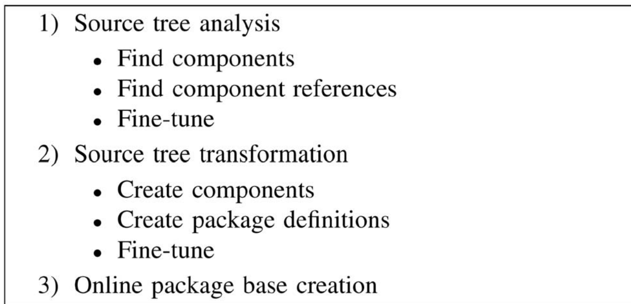

Fig. 2. The three phases of source tree decoupling.

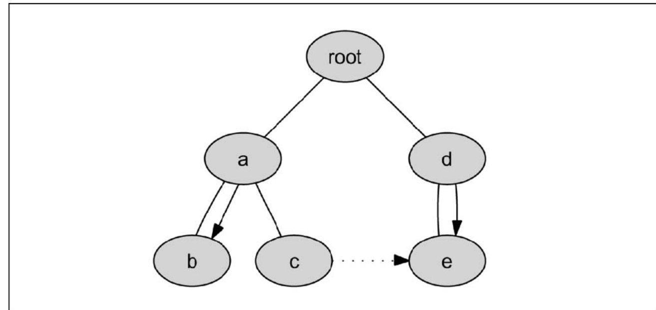

Fig. 3. A directory hierarchy with directory references represented by arrows.

5 Migration to Build-Level Components

The development rules for the component development of Section 3 and the composition technique presented in Section 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 semiautomatic technique for applying the development rules of Section 3 to existing software. Fig. 2 depicts the three-phase process for decoupling source trees into build-level components. This reengineering process analyzes the structure of a source tree to determine candidate components, and Makefiles determines 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 noncode artifacts (including build knowledge). If all sources were contained in a single directory, then some additional clustering techniques could 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: 1) each nonroot directory constitutes a separate component (i.e., a, b, c, d, and e form components); 2) 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 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 [10] to represent component dependency graphs. The adaptations are specified as graph transformations, which can be performed automatically. This made the analysis phase 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.

Creating Components. Creating a build-level component, involves: 1) isolating its implementation from the source tree, 2) creating an Autoconf configure script, and 3) 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, form separate components, they 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: 1) The name and version of the original system are replaced by the name and version of the component. 2) References to other directories are removed. Thus, all files listed in `AC_CONFIG_FILES` that do not belong to the component are removed. 3) Configuration switches are added for each component dependency. For component a and d 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 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 Section 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 these references from the Makefiles or by adding the missing files 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.

6 Decoupling Graphviz

6.1 Graphviz

Graphviz,[8] developed by AT&T, is an Open Source collection of tools for manipulating graph structures and generating graph layouts [10]. 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

8. www.graphviz.org.

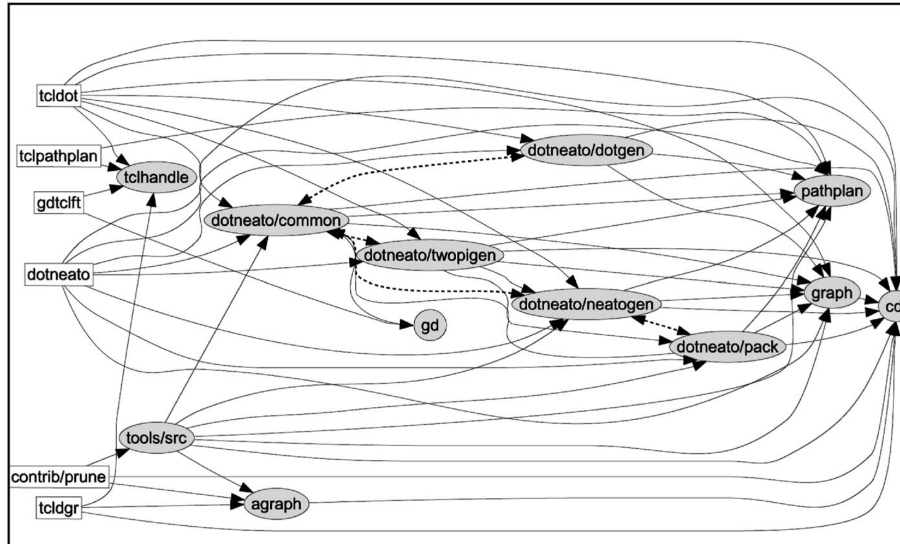

Fig. 4. Subset of Graphviz's directories and directory references.

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: 1) software distributions and installations become unnecessarily large and complex and 2) 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[9] contains 136 directories, 1,615 files, and 154 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 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: 1) 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. 2) Some arrows are pointing in two directions (i.e., the dashed arrows), indicating circular dependencies between directories. As we pointed out in Section 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 the integration of Graphviz with other software is painful.

9. Graphviz version 1.16

6.2 Restructuring Graphviz

Due to the aforementioned problems (i.e., Graphviz 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 is a perfect candidate for our semiautomatic 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 Section 3), the build-level components of Graphviz had to be repaired. This involved adapting the

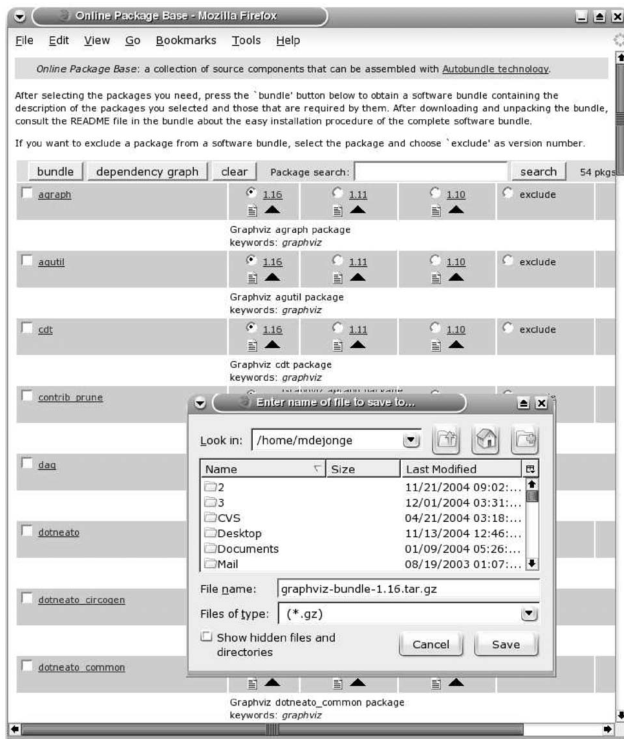

Fig. 5. The Graphviz package base with generated build-level components. The initial Graphviz distribution can be reconstructed by selecting the “graphviz” component and pressing the “bundle” button.

Makefiles of components such that all files referenced are also distributed.

6.3 Graphviz Components

The restructuring process yielded 46 build-level components. We automatically created releases for them and we generated a Graphviz online package base (see Fig. 5). Observe that coupling between these components has decreased because they no longer contain references to each others source trees. Consequently, each component can safely change its internal directory structure. Furthermore, since more fine-grained compositions can be made, cohesion of systems using graphviz’s components will increase. 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 Decoupling Mozilla

7.1 Mozilla

Mozilla is a full suite of integrated Internet applications including a Web browser, e-mail client, address book, Web page composer, Internet chat software, and calendar application [25]. The suite is developed as open source. It contains the all-in-one Internet application Mozilla and the next-generation Web browser FireFox.

The implementation of the complete suite is contained in a single source tree. At configuration time, the build system is instructed which application has to be constructed. For instance, if FireFox should be compiled, the build system is configured accordingly by instantiating the configuration system with FireFox-specific settings.

Not surprisingly, Mozilla’s source tree is huge.[10] It consists of 26,000 files in 2,500 directories. We counted 240 different file types. The build system consists of 1,350 Makefiles, that together contain 40,000 lines of build instructions. The configuration process consists of approximately 16,000 lines. Mozilla consists of more than 4,000,000 LOC.

The fact that all applications are contained in a single source tree forms an indication that the source tree contains a large amount of reusable code. Analysis of the source tree approves this. By inspecting the build system, we found 1,521 dependencies between directories. These yield an average fan-in/fan-out of 12. The fan-in of a directory d denotes the number of different directories that reference d. The fan-out of a directory d denotes the number of different directories referenced by d. Both numbers are indicators for software reuse. Fan-in denotes how often a component is reused, fan-out denotes how much the component reuses itself. The median fan-in and fan-out are 5 and 12, respectively. The maximum fan-in and fan-out are 100 and 42, respectively.

Despite reusability, functionality from the Mozilla source tree is hard to reuse outside the source tree. The reason is twofold. First, Mozilla uses a dedicated build system. It requires a special directory layout and all build and configuration knowledge is centralized. Second, the source tree has a huge amount of cyclic dependencies. This introduces special requirements on the build system (Mozilla uses a two-pass build to deal with cyclic dependencies [26]). Furthermore, it complicates reusing individual directories because one has to understand the web of dependencies a directory has. As a consequence, one cannot easily integrate Mozilla’s source directories or build process in other software systems.

The amount of reusable functionality hidden in strongly coupled directories makes the Mozilla source tree a perfect candidate for decoupling into build-level components. In the remainder of this section, we describe ongoing work about build-level reengineering of Mozilla.

7.2 Source Tree Analysis

The size of Mozilla forced us to develop robust technology for source tree analysis. To that end, we developed a Makefile grammar and a set of tools for Makefile analysis. The Makefile grammar allowed us to parse all Makefiles into a structured representation that is better suited for analysis and transformation than their textual representation.

In principle, Mozilla Makefiles are small declarative files with declarations for component names, component dependencies, and variability. Build actions are defined in a

10. We analyzed the FireFox 1.0 version of Mozilla’s source tree.

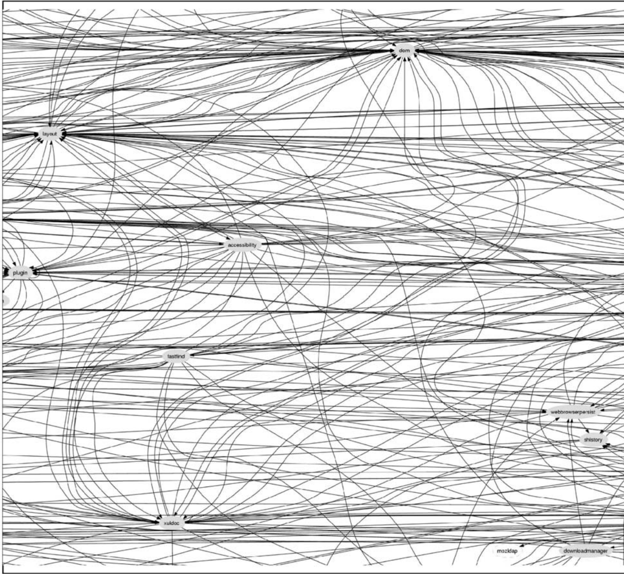

Fig. 6. Small fraction of FireFox's component dependency graph.

common file that is included by every Makefile. It turned out that Makefiles for a single component are often scattered around. Thus, the component structure does not always follow the directory structure. Cyclic dependencies are common practice and not considered harmful. Variability in the Mozilla build system is handled by conditional Makefile statements. We discovered that Makefiles are not always declarative because they often contain special build instructions. These additional build instructions have resulted in code duplication which may complicate maintenance of the build system in the future.

We transformed the parsed Makefiles to a graph in DOT format. This graph reflects the directory and component structure of Mozilla. It served as basis for further analysis and transformation. Analysis includes node and edge counting, cycle detection, and fan-in/fan-out calculation. Transformations include pruning to remove elements that are not part of the application being analyzed and coloring all edges on a cycle. Analysis of the FireFox Web browser yields a component dependency graph with 108 nodes and 1,078 edges (see Fig. 6). In this graph, we identified four clusters with cyclic dependent nodes and 37 components without cyclic dependencies.

7.3 Source Tree Transformation

Transforming Mozilla into build-level components is still work in progress. The problem with decoupling is twofold. The first problem is caused by the complex and centralized build process. Since it is not modular and not local to components, a build process has to be synthesized for each constructed component. There are two options: First, the build system can be duplicated for each component. This option fully reuses the build infrastructure that has already been developed. However, it keeps build processes complex and yields maintenance problems due to code duplication. Second, we can instead generate Automake build processes. Since build complexity is centralized and the amount of build knowledge in individual Makefiles is limited, generating Automake Makefiles should not be too difficult. First experiments indicate that both approaches are feasible. The second problem with transforming Mozilla is caused by cyclic dependencies. As a consequence of cyclic dependencies, we cannot simply isolate every component because dependencies cannot be linearized and because they require a nonstandard build process. This problem can be solved by creating composite build-level components for each cluster of cyclic-dependent components.

The migration of Mozilla can be performed in two phases. First, components are created for nodes that are not part of a cycle. This information follows directly from the component dependency graph. This phase is similar to the migration of Graphviz, except for build process creation. During the second phase, we create composite components for cyclic clusters. All components in a cyclic cluster will be deployed as a single composite build-level component. These components have multiple interfaces (one for each contained component). For instance, if subcomponents `A` and `B` are part of a composite component that installs under `prefix`, then their libraries are installed under `prefix/lib/A` and `prefix/lib/B`. We duplicate Mozilla's build process for composite components. Composite components can be refactored on demand to remove cyclic dependencies. When all cyclic dependencies have disappeared, a composite component can be split-up in separate build-level components.

In the case of FireFox, we can directly migrate 37 build-level components. We are in the process of actually applying this transformation to the Mozilla source tree. The four clusters that we identified during the analysis phase can be turned into composite components. We expect that by restructuring these clusters, a significant number of the 71 embedded components can be isolated.

8 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. Although partly based on existing technology, it is this combination of rules that is new and that offers CBSE benefits at the build level. We discussed an automated composition technique for build-level components. We defined a semiautomatic process for source tree decoupling to migrate 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 applied our techniques to Graphviz and Mozilla. Graphviz was decoupled successfully into 46 components. Migration of Mozilla is still in progress. Its build-level analysis revealed good internal reuse statistics, as well as serious deficiencies that hamper external reuse. We identified components and clusters that can be isolated for external reuse.

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 (e.g., Graphviz), 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. 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 applying our techniques to additional software systems.

Related work. Koala is a software component model that addresses source code composition [28]. Currently, it is tailored toward the C programming language. Because it focuses on a single programming language, Koala has more control over the composition process at the price of less genericity. For example, adopting nonKoala components is not trivial. In [7], we propose to combine both approaches.

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 [13]. Their main concern is understanding the different precompile, compile, and link steps that are involved in a build process, not restructuring source code, or making build and configuration processes compositional. In [33], 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. In [4], [5], build-level complexity in terms of file and directory structure is analyzed. The complexity of build and configuration processes is not analyzed. They find that system growth increases build-level complexity. Since software systems typically grow over time [1], build-level complexity is expected to increase as well. This forms a motivation for improving build-level software development practice as we proposed in this paper.

There exist several clustering techniques that help to capture the structure of existing software systems [18], [34]. In [6], 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. In [22], [23], [8], a clustering approach with module granularity is discussed. They analyze module references and they cluster software systems accordingly. Their key motivation for decomposition is improved system understanding and not reuse which is ours. Their approach has several interesting relations with our work. First, they inspect systems on a smaller (module) granularity then we (directory granularity). Second, their approach synthesizes a decomposition, while we take the clustering in a directory for granted. Third, they also use (part of Graphviz) as case study. We believe that a clustering of files in directories also gives information about a software system. The relation between their synthesized clusters and the existing directory structure is not discussed. For instance, applying our clustering approach to the `dot` program (which is part of Graphviz) yields a different number of clusters and a different structuring of files over thee clusters. It would be interesting to compare both approaches with respect to software reuse and system understanding. In addition to these, more generic approaches, such as the Software Reflexion Model [27], can also be used to reengineer software systems from build-level artifacts.

It is sometimes argued that build knowledge should not be spread across directories at all, but contained in a single `Makefile` [24]. The motivation is that only in a single `Makefile` can completeness of build dependencies 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.

REFERENCES

[1] L.A. Belady and M.M. Lehman, "A Model of Large Program Development," *IBM Systems J.*, vol. 15, no. 3, pp. 225-252, 1976.

[2] J. Bergstra and P. Klint, "The Discrete Time ToolBus—A Software Coordination Architecture," *Science of Computer Programming*, vol. 31, nos. 2-3, pp. 205-229, July 1998.

[3] M. van den Brand, A. van Deursen, J. Heering, H. Jong, M. de 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," *Proc. Conf. Compiler Construction (CC '01)*, R. Wilhelm, ed., 2001.

[4] A. Capiluppi, M. Morisio, and J.F. Ramil, "The Evolution of Source Folder Structure in Actively Evolved Open Source Systems," *Proc. 10th Int'l Symp. Software Metrics (METRICS '04)*, L. Ott, ed., pp. 2-13, 2004.

[5] A. Capiluppi, M. Morisio, and J.F. Ramil, "Structural Evolution of an Open Source System: A Case Study," *Proc. 12th IEEE Int'l Workshop Program Comprehension (IWPC '04)*, pp. 172-182, 2004.

[6] Y. Chiricota, F. Jourdan, and G. Melanoon, "Software Components Capture Using Graph Clustering," *Proc. 11th Int'l Workshop Program Comprehension (IWPC '03)*, pp. 217-226, May 2003.

[7] M. de Jonge, "Multi-Level Component Composition," *Proc. Second Groningen Workshop Software Variability Modeling (SVM '04)*, no. 2004-7-01, J. Bosch, ed., Dec. 2004.

[8] D. Doval, S. Mancoridis, and B.S. Mitchell, "Automatic Clustering of Software Systems Using a Genetic Algorithm," *Proc. Software Technology and Eng. Practice*, pp. 73-81, 1999.

[9] Free Software Foundation, GNU Coding Standards, 2004, http://www.gnu.org/prep/standards.html.

[10] E. Gansner and S. North, "An Open Graph Visualization System and Its Applications to Software Engineering," *Software—Practice and Experience,* vol. 30, no. 11, pp. 1203-1233, Sept. 2000.
[11] Graphviz, http://www.graphviz.org, 2005.
[12] A. van der Hoek and A. Wolf, "Software Release Management for Component-Based Software," *Software—Practice and Experience,* vol. 33, no. 1, pp. 77-98, Jan. 2003.
[13] R. Holt, M. Godfrey, and X. Dong, "The Build/Comprehend Pipelines," *Proc. Second ASERC Workshop Software Architecture,* Feb. 2003.
[14] M. de Jonge, "The Linux Kernel as Flexible Product-Line Architecture," Technical Report SEN-R0205, CWI, 2002.
[15] M. de Jonge, "Source Tree Composition," *Proc. Seventh Int'l Conf. Software Reuse,* C. Gacek, ed., Apr. 2002.
[16] M. de Jonge, "Package-Based Software Development," *Proc. 29th Euromicro Conf.,* pp. 76-85, Sept. 2003.
[17] M. de Jonge, "To Reuse or to Be Reused: Techniques for Component Composition and Construction," PhD thesis, Faculty of Natural Sciences, Math., and Computer Science, Univ. of Amsterdam, Jan. 2003.
[18] R. Koschke and T. Eisenbarth, "A Framework for Experimental Evaluation of Clustering Techniques," *Proc. IEEE Eighth Int'l Workshop Program Comprehension (IWPC '00),* pp. 201-210, June 2000.
[19] The Linux Kernel Archives, http://www.kernel.org, 2005.
[20] D. Mackenzie, B. Elliston, and A. Demaile, "Autoconf: Creating Automatic Configuration Scripts," Free Software Foundation, 2002, http://www.gnu.org/software/autoconf.
[21] D. Mackenzie and T. Tromey, "GNU Automake Manual," Free Software Foundation, 2003, http://www.gnu.org/software/automake.
[22] S. Mancoridis, B.S. Mitchell, Y. Chen, and G. Gansner, "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 50-59, 1999.
[23] S. Mancoridis, B.S. Mitchell, C. Rorres, C. Chen, and E.R. Gansner, "Using Automatic Clustering to Produce High-Level System Organizations of Source Code," *Proc. 1998 Int'l Workshop Program Understanding (IWPC '98),* pp. 45-52, 1998.
[24] P. Miller, "Recursive Make Considered Harmful," *Unix/Linux J. (AUUGN),* vol. 19, no. 1, pp. 14-25, 1998, http://aegis.sourceforge.net/auug97.pdf.
[25] Mozilla, http://www.mozilla.org, 2005.
[26] Mozilla's build system, http://www.mozilla.org/build/build-system.html, 2005
[27] G.C. Murphy, D. Notkin, and K.J. Sullivan, "Software Reflexion Models: Bridging the Gap Between Design and Implementation," *IEEE Trans. Software Eng.,* vol. 27, no. 4, pp. 364-380, Apr. 2001.
[28] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," *Computer,* vol. 33, no. 3, pp. 78-85, Mar. 2000.
[29] OpenOffice, http://www.openoffice.org, 2005.
[30] D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Comm. ACM,* vol. 15, no. 12, pp. 1053-1058, Dec. 1972.
[31] R. Russell, D. Quinlan, and C. Yeoh, "Filesystem Hierarchy Standard," technical report, Filesystem Hierarchy Standard Group, 2004.
[32] C. Szyperski, *Component Software: Beyond Object-Orientated Programming,* second ed., Addison-Wesley, 2002.
[33] Q. Tu, M. Godfrey, and X. Dong, "The Build-Time Architectural View," *Proc. Int'l Conf. Software Maintenance (ICSM 2001),* pp. 398-407, Nov. 2001.
[34] T.A. Wiggerts, "Using Clustering Algorithms in Legacy Systems Remodularization," *Proc. Fourth Working Conf. Reverse Eng.,* pp. 33-43, 1997.

Merijn de Jonge received the MSc degree (1997) and the PhD degree (2003) from the University of Amsterdam, the Netherlands. He worked on software component technologies, configuration management, and software deployment at the Technical University of Eindhoven and the University of Utrecht. He is currently a senior scientist at Philips Research Laboratories, Eindhoven, the Netherlands. His research interests include software reuse, component-based software engineering, program analysis, and program transformation.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.