To reuse or to be reused. Techniques for component composition and construction

de Jonge, M.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
A typical problem of component-based applications is their complicated construction and distribution because the internal structuring in components usually remains visible at construction and distribution-time. For example, it is not easy to deliver the SDL documentation generator from Chapter 5, "Cost-Effective Maintenance Tools for Proprietary Languages", as a unit to a customer, or to configure and build it as a unit. Consequently, each constituent component of a software system has to be separately retrieved, compiled, installed and so on.

This chapter tackles this problem by providing techniques for automated assembly of composite software systems from their constituent source code components. This process is called Source Tree Composition and involves integration of source trees, build processes, and configuration processes. The result is a software system that hides its internal structuring in components and, consequently, can be managed as a single unit.

Application domains of source tree composition include generative programming, product line architectures, commercial off-the-shelf (COTS) software engineering, and Language-Centered Software Engineering (LCSE). The work presented in this chapter was published earlier as [81].

6.1 Introduction

The classical approach of component composition is based on pre-installed binary components (such as pre-installed libraries). This approach however, complicates software development because: (i) system building requires extra effort to configure and install the components prior to building the system itself; (ii) it yields accessibility problems to locate components and corresponding documentation [99]; (iii) it complicates the process of building self-
contained distributions from a system and all its components. Package managers (such as RPM [6]) reduce build effort but do not help much to solve the remaining problems. Furthermore, they introduce version problems when different versions of a component are used [99, 135]. They also provide restricted control over a component's configuration. All these complicating factors hamper software reuse and negatively influence granularity of reuse [112].

We argue that source code components (as alternative to binary components) can improve software reuse for component-based software development. Source code components are source files divided in directory structures. They form the implementation of subsystems. Source code component composition yields self-contained source trees with single integrated configuration and build processes. We called this process source tree composition.

The literature contains many references to articles dealing with component composition on the design and execution level, and with build processes of individual components (see the related work in Section 6.9). However, techniques for composition of source trees of diverse components, developed in different organizations, in multiple languages, for the construction of systems which are to be reusable themselves and to be distributed in source, are underexposed and are the subject of this chapter.

The chapter is organized as follows. Section 6.2 motivates the need for advanced techniques to perform source tree composition. Section 6.3 describes terminology. Section 6.4 describes the process of source tree composition. Sections 6.5 and 6.6 describe abstraction mechanisms over source trees and composite software systems. Section 6.7 describes automated source tree composition. It discusses the tool autobundle, online package bases, and product line architectures. Section 6.8 describes experiences with source tree composition. Related work and concluding remarks are discussed in Sections 6.9 and 6.10.

### 6.2 Motivation

The source code components that form a software system are often tightly coupled: the implementation of all subsystems is contained in a single source tree, a central build process controls their build processes, and a central configuration process performs their static (compile-time) configuration. For example, a top-level Makefile often controls the global build process of a software system. A system is then built by recursively executing `make` [59] from the top-level Makefile for each source code component. Often, a global GNU autoconf [100] configuration script performs system configuration, for instance to select the compilers to use and to enable or disable debugging support.

Such tight coupling of source code components has two main advantages: (i) due to build process integration, building and configuring a system can be

---

Please note that despite the advantages that source code components provide, binary components may still be mandated, for instance, to protect intellectual property.
performed easily from one central place; (ii) distributing the system as a unit is relatively easy because all source is contained in a single tree (one source tree, one product).

Unfortunately, tight coupling of source code components also has several drawbacks:

- The composition of components is inflexible. It requires adaption of the global build instructions and (possibly) its build configuration when new components are added [99]. For example, it requires adaption of a top-level Makefile to execute `make` recursively for the new component.

- Potentially reusable code does not come available for reuse outside the system because entangled build instructions and build configuration of components are not reusable [112]. For example, as a result of using `autoconf`, a component's configuration is contained in a top-level configuration script and therefore not directly available for reuse.

- Direct references into source trees of components yield unnecessary file system dependencies between components in addition to functional dependencies. Changing the file or directory structure of one component may break another.

To address these problems, the constituent source code components of a system should be isolated and be made available for reuse (`system decomposition`). After decomposition, new systems can be developed by selecting components and assembling them together (`system composition`). This process is depicted in Figure 6.1.

For system composition not only source files are required, but also all build knowledge of all constituent source code components. Therefore, we define `source tree composition` as the composition of all files, directories, and build knowledge of all reused components. To benefit from the advantages of a tightly coupled system, source tree composition should yield a self-contained source tree with central build and configuration processes, which can be distributed as a unit.

When the reuse scope of software components is restricted to a single Configuration Management (CM) [17] system, source tree composition might be easy. This is because, ideally, a CM system administers the build knowledge of all components, their dependencies, etc., and is able to perform the composition automatically.\(^2\)

When the reuse scope is extended to multiple projects or organizations, source tree composition becomes harder because configuration management (including build knowledge) needs to be untangled [40, 112]. Source tree composition is further complicated when third party components are reused, when the resulting system has to be reusable itself, and when it has to be

---

\(^2\)Observe that in practice, CM systems are often confused with version management systems. The latter do not administrate knowledge suitable for source tree composition.
Component development, system composition, and system assembly with source code component reuse. Components are developed individually; compositions of components form systems, which are assembled to form software bundles (self-contained software systems).

distributed as source. This is because: i) standardization of CM systems is lacking [112, 151]; ii) control over build processes of third party components is restricted; iii) expertise on building the system and its constituent components might be unavailable.

Summarizing, to increase reuse of source code components, source tree composition should be made more generally applicable. This requires techniques to hide the decomposition of systems at distribution time, to fully integrate build processes of (third party) components, and to minimize configuration and build effort of the system. Once generally applicable, source tree composition simplifies assembling component-based software systems from implementing source code components.

Suppliers of Commercial Off-The-Shelf (COTS) source code components and of Open Source Software (OSS) components can benefit from the techniques presented in this chapter because integration of their components is
simplified, which makes them suitable for widespread use. Moreover, as we will see in Section 6.7.3 and in Chapter 7, "Feature-Based Product Line Instantiation using Source-Level Packages", product line architectures, which are concerned with assembling families of related applications, can also benefit from source tree composition.

6.3 Terminology

System building is the process of deriving the targets of a software system (or software component) from source [47]. We call the set of targets (such as executables, libraries, and documentation) a software product, and define a software package as a distribution unit of a versioned software system in either binary or source form.

A system's build process is divided in several steps, which we call build actions. They constitute a system's build interface. A build action is defined in terms of build instructions which state how to fulfill the action. For example, a build process driven by make typically contains the build actions all, install, and check. The all action, which builds the complete software product, might be implemented as a sequence of build instructions in which an executable is derived from C program text by calling a compiler and a linker.

System building and system behavior can be controlled by static configuration [51]. Stractially configurable parameters define at compile-time which parts of a system to build and how to build them. Examples of such parameters are debug support (by turning debug information on or off), and the set of drivers to include in an executable. We call the set of statically configurable parameters of a system a configuration interface.

We define a source tree as a directory hierarchy containing all source files of a software (sub) system. A source tree includes the sources of the system itself, files containing build instructions (such as Makefiles), and configuration files, such as autoconf configuration scripts.

6.4 Source tree composition

Source tree composition is the process of assembling software systems by putting source trees of reusable components together. It involves merging source trees, build processes, and configuration processes. Source tree composition yields a single source tree with centralized build and configuration processes.

The aim of source tree composition is to improve reusability of source code components. To be successful, source tree composition should meet the following two requirements:

Repeatable To benefit from any evolution of the individual components, it is essential that an old version of a component can easily be replaced by a
Due to lacking standardization of build and configuration processes, these requirements are hard to satisfy. Especially when drawing on a diverse collection of software components, developed and maintained in different institutes, by different people, and implemented in different programming languages. Composition of source trees therefore often requires fine-tuning a system's build and configuration process, or even adapting the components themselves.

To improve this situation, we propose to formalize the parameters of source code packages and to hide component-specific build and configuration processes behind interfaces. A standardized build interface defines the build actions of a component. A configuration interface defines a component's configurable items. An integrated build process is formed by composing the build actions of each component sequentially. The configuration interface of a composed system is formed by merging the configuration interfaces of its constituent components.
6.5 Definition of single source trees

We propose source code packages as units of reuse for source tree composition. They help to: i) easily distinguish different versions of a component and to allow them to coexist; ii) make source tree composition institute and project-independent because versioned distributions are independent of any CM system; iii) allow simultaneous development and use of source code components.

To be effectively reusable, software packages require abstractions [92]. We introduce package definitions as abstraction of source code packages. We developed a domain-specific language to represent them, of which an example is depicted in Figure 6.2. It defines the software package CoboSQLTrans which is intended to develop transformations for COBOL with embedded SQL.

Package definitions define the parameters of packages, which include package identification, package dependencies, and package configuration.

Package identification The minimal information that is needed to identify a software package are its name and version number. These, as well as the URL where the package can be obtained, a short description of the package, and a list of keywords are recorded in a package's identification section (see Figure 6.2).

Package configuration The configuration interface of a software package is defined in the configuration interface section. In Figure 6.2, the configuration interface defines a single configuration parameter and a short usage description of this parameter. With this parameter, support for layout preserving transformations in the CoboSQLTrans package can be turned on or off. Partial configuration enforced by other components and composition of configuration interfaces is discussed in Section 6.6.

Package dependencies To support true development with reuse, a package definition can list the packages that it reuses in the requires section. Package definitions also allow to define a (partial) static configuration for required packages. Package dependencies are used during package normalization (see Section 6.6) to synthesize the complete set of packages that form a system. For example, the package of Figure 6.2 requires at least version 0.5 of the cobol package and configures it with embedded SQL. Further package requirements are the Algebraic Specification Formalism (ASF) as programming language with support for automatic term traversal, a parser (sg1r), and a pretty-printer (GPP).
bundle
name=CobolSQLTrans-bundle version=1.0
configuration interface
layout-preserving
'Enable layout preserving transformations.'
boxenv
'Location of external boxenv package.'
bundles
package
name=sdf version=2.1
configuration
package
name=sql version=0.2
configuration
package
name= Cobol version=0.5
configuration
lang-ext=SQL
package
name=atterm version=1.6.3
configuration
package
name=asf version=1.1
configuration
traversals=on
package
name=sglr version=3.0
configuration
package
name=gpp version=2.0
configuration
package
name=CobolSQLTrans version=1.0
configuration

Figure 6.3 Bundle definition obtained by normalizing the package definition of Figure 6.2. This definition has been stripped due to space limitations.

6.6 Definition of composite source trees

A software bundle is the source tree that results from a particular source tree composition. A bundle definition (see Figure 6.3) defines the ingredients of a bundle, its configuration interface, and its identification. The ingredients of a bundle are defined as composition of package definitions.

A bundle definition is obtained through a process called package normaliza-
Section 6.6  Definition of composite source trees

**Figure 6.4** A package dependency graph for the COBOL transformation package of Figure 6.2. The dashed node denotes an unresolved package dependency.

which includes package dependency and version resolution, build order arrangement, configuration distribution, and bundle interface construction.

**Dependency resolution**  Unless otherwise specified, package normalization calculates the transitive closure of all required packages and collects all corresponding package definitions. The list of required packages follows directly from the bundle’s package dependency graph (see Figure 6.4). For instance, during normalization of the package definition of Figure 6.2, dependency upon the aterm package is signaled and its definition is included in the bundle definition. When a package definition is missing (see the dashed node in Figure 6.4), a configuration parameter is added to the bundle’s configuration interface (see below).

**Version resolution**  One software bundle cannot contain multiple versions of a single package. When dependency resolution signals that different versions of a package are required, the package normalization process should decide which version to bundle.

Essential for package normalization is compatibility between different versions of a package (see [149, 47, 151] for a discussion of version models). In accordance with [110], we require *backwards compatibility* to make sure that a particular version of a package can always be replaced by one of its successors. When backwards compatibility of a package cannot be satisfied, a new package (with a different name) should be created. Our tooling can be instantiated with different version schemes allowing experimenting with other (weakened) version requirements.

**Build order arrangement**  Package dependencies serve to define the build order of composite software systems: building a package should be delayed until
all of its required packages have been built. During package normalization, the collected package definitions are correctly ordered linearly according to a bottom-up traversal of the dependency graph. Therefore, the cobol package occurs after the sql package in the bundle definition of Figure 6.3. Circular dependencies between packages are not allowed. Such circularities correspond to bootstrapping problems and should be solved by package developers (for instance by splitting packages or by creating dedicated bootstrap packages).

**Configuration propagation** Each package definition that is collected during package normalization contains a (possibly empty) set of configurable parameters, its configuration interface. Configurable parameters might get bound when the package is used by another package imposing a particular configuration. During normalization, this configuration is determined by collecting all the bindings of each package. For example, the CobolSQLTrans package of Figure 6.2 binds the configurable parameter lang-ext of the cobol package to SQL, the parameter traversals of the asf package is bound to on (see Figure 6.3). A conflicting configuration occurs when a single parameter gets bound differently. As an example, consider a software bundle that bundles packages A (which has a debug configuration switch), B, and C. A configuration conflict occurs when package B uses the debug switch of package A to turn debug support on, while package B uses it to turn debugging off. Such configuration conflicts can easily be detected during package normalization.

**Bundle interface construction** The configuration interface of a bundle is formed by collecting all unbound configurable parameters of bundled packages. In addition, it is extended with parameters for unresolved package requirements and for packages that have been explicitly excluded from the package normalization process. These parameters serve to specify the installation locations of missing packages at compile-time. The configuration interface of the CobolSQLTrans package (see Figure 6.3) is formed by the layout-preserving parameter originating from the CobolSQLTrans package, and the boxenv parameter which is due to the unresolved dependency of the gpp package (see Figure 6.4).

After normalization, a bundle definition defines a software system as collection of software packages. It includes package definitions of all required packages and configuration parameters for those that are missing. Furthermore, it defines a partial configuration for packages and their build order. This information is sufficient to perform a composition of source trees. In the next section we discuss how this can be automated.
### Automated source tree composition

We automated source tree composition in the tool *autobundle*. In addition, we implemented tools to make package definitions available via *online package bases*. Online package bases form central meeting points for package developers and package users, and provide online package selection, bundling, and contribution via Internet. These techniques can be used to automate system assembling in product line architectures.

#### 6.7.1 Autobundle

Package normalization and bundle generation are implemented by *autobundle*. This tool produces a software bundle containing top-level configuration and build procedures, and a list of bundled packages with their download locations (see Table 6.1).

The generated bundle does not contain the source trees of individual packages yet, but rather the tool *collect* that can collect the packages and integrate them in the generated bundle automatically. The reason to generate an empty bundle is twofold: i) since *autobundle* typically runs on a server (see Section 6.7.2), collecting, integrating, and building distributions would reduce server performance too much. By letting the user perform these tasks, the server gets relieved significantly. ii) It protects an *autobundle* server from legal issues when copyright restrictions prohibit redistribution or bundling of packages because no software is redistributed or bundled at all.

To obtain the software packages and to build self-contained distributions, the build interface of a generated bundle contains the build actions *collect*, to download and integrate the source trees of all packages, and *bundle* to also

---

**Table 6.1** Files that are contained in a generated software bundle.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makefile.am</td>
<td>Top-level automake Makefile that integrates build processes of all bundled packages.</td>
</tr>
<tr>
<td>configure.in</td>
<td>An autoconf configuration script to perform central configuration of all packages in a software bundle.</td>
</tr>
<tr>
<td>pkg-list</td>
<td>A list of the packages of a bundle, their versions, and download locations.</td>
</tr>
<tr>
<td>collect</td>
<td>A tool that downloads, unpacks, and integrates the packages listed in pkg-list.</td>
</tr>
<tr>
<td>README</td>
<td>A file that briefly describes the software bundle and its packages.</td>
</tr>
<tr>
<td>acinclude.m4</td>
<td>A file containing extensions to autoconf functionality to make central configuration of packages possible.</td>
</tr>
</tbody>
</table>
Table 6.2 Build actions of the standardized build interface required by autobundle. In addition, a tool configure for static configuration is also required.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>Build action to build all targets of a source code package.</td>
</tr>
<tr>
<td>install</td>
<td>Build action to install all targets.</td>
</tr>
<tr>
<td>clean</td>
<td>Build action to remove all targets and intermediate results.</td>
</tr>
<tr>
<td>dist</td>
<td>Build action to generate a source code distribution.</td>
</tr>
<tr>
<td>check</td>
<td>Build action to verify run-time behavior of the system.</td>
</tr>
</tbody>
</table>

put them into a single source distribution.

The generated bundle is driven by make [59] and offers a standardized build interface (see Table 6.2). The build interface and corresponding build instructions are generated by autoconf [100] and automake [101]. The tool autoconf generates software configuration scripts and standardizes static software configuration. The tool automake provides a standardized set of build actions by generating Makefiles from abstract build process descriptions. Currently we require that these tools are also used by bundled packages. We used the tools because they are freely available and in widespread use. However, they are not essential for the concept of source tree composition. Essential is the availability of a standardized build interface (such as the one in Table 6.2); any build system that implements this interface would suffice. Moreover, when a build system does not implement this interface, it would not be difficult to hide the package-specific configuration and build instructions behind the standardized build interface.

After the packages are automatically collected and integrated, the top-level build and configuration processes take care of building and configuring the individual components in the correct order. The build process also provides support for generating a self-contained source distribution from the complete bundle. This hides the structuring of the system in components and allows a developer to distribute his software product as a single unit. The complete process is depicted in Figure 6.5.

### 6.7.2 Online package bases

Resolution of package dependencies is performed by searching for package definitions in package repositories. We developed tools to make such repositories browsable and searchable via Inter/Intranet, and we implemented HTML form generation for interactive package selection. The form constitutes an online package base and lists packages and available versions together with descriptions and keywords. The form can be filled out by selecting the packages of need. By pressing the “bundle” button, the autobundle server is requested to generate the desired bundle. Anyone can contribute by filling out an online package contribution form. After submitting this form, a package definition is
Figure 6.5 Construction and distribution of software systems with source tree composition. (1) Packages of need are selected. (2) The selected set of packages is normalized to form a bundle definition. (3) From this definition an empty software bundle is generated. (4) Required software packages are collected and integrated in the bundle, after which the system can be built (5a), or be distributed as a self-contained unit (5b).

generated and the online package base is updated. This is the only required step to make an autoconf/automake-based package available for reuse with autobundle.

Online package bases can be deployed to enable and control software reuse within a particular reuse scope (for instance, group, department, or company wide). They make software reuse and software dependencies explicit because a distribution policy of software components is required when source code packages form the unit of reuse.

6.7.3 Product line architectures

Online package bases allow the software engineer to easily assemble systems by selecting components of need. An assembled system is partly configured depending on the combination of components. Remaining variation points can be configured at compile-time. This approach of system assembly is related to the domain of product line architectures.

A Product Line Architecture (PLA) is a design for families of related applications; application construction (also called product instantiation [66]) is accomplished by composing reusable components [10]. The building blocks from which applications are assembled are usually abstract requirements (con-
sisting of application-oriented concepts and features). For the construction of the application, corresponding implementation components are required. To automate component assembly, configuration knowledge is required which maps between the problem space (consisting of abstract requirements) and the solution space (consisting of implementation components) [50].

We believe that package definitions, bundle generation, and online package bases serve implementing a PLA by automating the integration of source trees and static configuration. Integration of functionality of components still needs to be implemented in the components themselves, for instance as part of a component's build process.

Our package definition language can serve as a configuration DSL (Domain-Specific Language) [51]. It then serves to capture configuration knowledge and to define mappings from the problem space to the solution space. Abstract components from the problem space are distinguished from implementation components by having an empty location field in their package definition. A mapping is defined by specifying an implementation component in the requires section of an abstract package definition.

System assembling can be automated by autobundle. It normalizes a set of abstract components (features) and produces a source tree containing all corresponding implementation components and generates a (partial) configuration for them. Variation points of the assembled system can be configured statically via the generated configuration interface. An assembled system forms a unit which can easily be distributed and reused in other products.

Definitions of abstract packages can be made available via online package bases. Package bases then serve to represent application-oriented concepts and features similar to feature diagrams [86]. This makes assembling applications as easy as selecting the features of need.

Using source tree composition for product lines is further explored in Chapter 7, "Feature-Based Product Line Instantiation using Source-Level Packages".

6.8 Case studies

System development We successfully applied source tree composition to the ASF+SDF Meta-Environment [27], an integrated environment for the development of programming languages and tools, which has been developed at our research group. Source tree composition solved the following problems that we encountered in the past:

- We had difficulties in distributing the system as a unit. We were using ad-hoc methods to bundle all required components and to integrate their build processes.

- We were encountering the well-known problem of simultaneously developing and using tools. Because we did not have a distribution policy for
individual components, development and use of components were often conflicting activities.

- Most of the constituent components were generic in nature. Due to their entangling in the system's source tree however, reuse of individual components across project boundaries proved to be extremely problematic.

After we started using source tree composition techniques, reusability of our components greatly improved. This was demonstrated by the development of XT, a bundle of program transformation tools (see Chapter 3). It bundles components from the ASF+SDF Meta-Environment together with a diverse collection of components related to program transformation. Currently, XT is assembled from 25 reusable source code components developed at three different institutes.4

For both projects, package definitions, package normalization, and bundle generation proved to be extremely helpful for building self-contained source distributions. With these techniques, building distributions of the ASF+SDF Meta-Environment and of XT became a completely automated process. Defining the top-level component of a system (i.e., the root node in the system's package dependency graph) suffices to generate a distribution of the system.

**Online Package Base** To improve flexibility of component composition, we defined package definitions for all of our software packages, included them in a single package repository and made that available via Internet as the Online Package Base (see Figure 6.6).

With the Online Package Base (OPB), building source distributions of XT and of the ASF+SDF Meta-Environment becomes a dynamic process and reduces to selecting one of these packages and submitting a bundle request to the autobundle server. The exact contents of both distributions can be controlled for specific needs by in/excluding components, or by enforcing additional version requirements of individual components. Similarly, any composition of our components can be obtained via the OPB.

Although it was initiated to simplify and increase reuse of our own software packages, anyone can now contribute by filling out a package contribution form. Hence, compositions with third-party components can also be made. For example, the OPB contains several package definitions for GNU software, the graph drawing package graphviz from AT&T, and components from a number of other research institutes.

**Stratego compiler** Recently, the Stratego compiler [140] has been split up in reusable packages (including the Stratego run-time system). The constituting components (developed at different institutes) are bundled with autobundle to form a stand-alone distribution of the compiler. With autobundle also

4See Appendix A for information about the availability of the Online Package Base.
more fine-grained reuse of these packages is possible. An example is the distribution of a compiled Stratego program with only the Stratego run-time system. The Stratego compiler also illustrates the usefulness of nested bundles. Though a composite bundle, the Stratego compiler is treated as a single component by the xt bundle in which it is included.

**Product line architectures**  We have investigated the use of *autobundle* and online package bases in a commercial setting to transform the industrial application DOCGEN [57] into a product line architecture [52]. DOCGEN is a documentation generator which generates interactive, hyperlinked documentation about legacy systems. Documentation generation consists of generic and specific artifact extraction and visualization in a customer-specific layout. It is important that customer-specific code is not delivered to other customers (i.e., that certain packages are not bundled).

The variation points of DOCGEN have been examined and captured in a Feature Description Language (FDL) [56]. We are analyzing how feature selection (for instance the artifacts to document and which layout to use) can be performed via an online package base. Package definitions serve to map selected features to corresponding implementing components (such as specific extractors and visualizers). Such a feature set is normalized by *autobundle* to a bundle of software packages, which are then integrated into a single source tree that forms the intended customer-specific product. In Chapter 7, “Feature-Based Product Line Instantiation using Source-Level Packages”, we will further discuss the DOCGEN product line.

### 6.9 Related work

Many articles, for instance [39, 35, 46], address build processes and tools to perform builds. Tools and techniques are discussed to solve limitations of traditional *make* [59], such as improving dependency resolution, build performance, and support for variant builds. Composition of source trees and build processes is not addressed.

Gunter [65] discusses an abstract model of dependencies between software configuration items based on a theory of concurrent computations over a class of Petri nets. It can be used to combine build processes of various software environments.

Miller [104] motivates global definition of a system's build process to allow maximal dependency tracking and to improve build performance. However, to enable composition of components, independence of components (weak coupling) is important [149]. For source tree composition this implies independence of individual build processes and therefore contradicts the approach of [104]. Since the approach of Miller entangles all components of the system, we believe that it will hamper software reuse.
This is the **Online Package Base**: a collection of reusable Free Software packages developed at different institutes.

After selecting the packages you need, press the 'bundle' button below to obtain a software bundle containing the description of the packages you selected and those that are required by them. After downloading and unpacking the bundle, consult the README file in the bundle about the easy installation procedure of the complete software bundle.

If you want to exclude a package from a software bundle, select the package and choose 'exclude' as version number.

You can contribute your own software packages, by donating a package definition file. To do so, please fill out a [package contribution form](#).

To get an impression about software reuse via the Package Base, visit the Package Dependency Graph.

**Figure 6.6** Automated source tree composition at the Online Package Base. See Appendix A for information about the availability of the Online Package Base.
This chapter addresses techniques to assemble software systems by integrating source trees of reusable components. In practice, such components are often distributed separately and their installation is required prior to building the system itself. This extra installation effort is problematic [135], even when partly automated by package managers (like RPM [6]). Although source tree composition simplifies software building, it does not make package management superfluous. The use of package managers is therefore still advocated to assist system administrators in installing (binary) distributions of assembled systems.

The work presented in this chapter has several similarities with the component model Koala [112, 110]. The Koala model has a component description language like our package definition language, and implementations and component descriptions are stored in central repositories accessible via Internet. They also emphasize the need for backward compatibility and the need to untangle build knowledge from an SCM system to make components reusable. Unlike our approach, the system is restricted to the C programming language, and merging the underlying implementations of selected components is not addressed.

In [71], a software release management process is discussed that documents released source code components, records and exploits dependencies amongst components, and supports location and retrieval of groups of compatible components. Their primarily focus is component release and installation, not development of composite systems and component integration as is the case in this chapter.

6.10 Concluding remarks

This chapter addresses software reuse based on source code components and on software assembly using the technique source tree composition. Source tree composition integrates source trees and build processes of individual source code components to form self-contained source trees with single integrated configuration and build processes.

Contributions We provided an abstraction mechanism for source code packages and software bundles in the form of package and bundle definitions. By normalizing a collection of package definitions (package normalization) a composition of packages is synthesized. The tool autobundle implements package normalization and bundle generation. It fully automates source tree composition. Online package bases, which are automatically generated from package repositories, make package selection easy. They enable source code reuse within a particular reuse scope. Source tree composition can be deployed to automate dynamic system assembly in product line architectures.
Component and reuse. The implementation of autobundle and the tools for generating online package bases follows the Language-Centered Software...
Table 6.3  Reuse table for the autobundle package. The table shows that for this package, a total of 1,699 new lines of code had to be written.

<table>
<thead>
<tr>
<th>Component</th>
<th>Non-transitive reuse LOC</th>
<th>RSI</th>
<th>Reuse%</th>
<th>Transitive reuse LOC</th>
<th>Reuse%</th>
</tr>
</thead>
<tbody>
<tr>
<td>autobundle</td>
<td>175</td>
<td>0</td>
<td>0%</td>
<td>4,062</td>
<td>95%</td>
</tr>
<tr>
<td>bundle2configure</td>
<td>950</td>
<td>780</td>
<td>82%</td>
<td>950</td>
<td>82%</td>
</tr>
<tr>
<td>bundle2pkglist</td>
<td>576</td>
<td>541</td>
<td>93%</td>
<td>576</td>
<td>93%</td>
</tr>
<tr>
<td>bundlegen</td>
<td>2,361</td>
<td>1,505</td>
<td>63%</td>
<td>2,361</td>
<td>63%</td>
</tr>
<tr>
<td>pkgs2form</td>
<td>1,451</td>
<td>1,082</td>
<td>74%</td>
<td>1,451</td>
<td>74%</td>
</tr>
<tr>
<td>pkg-search</td>
<td>873</td>
<td>779</td>
<td>89%</td>
<td>873</td>
<td>89%</td>
</tr>
<tr>
<td><strong>Totals:</strong></td>
<td><strong>6,386</strong></td>
<td><strong>4,687</strong></td>
<td><strong>73%</strong></td>
<td><strong>10,273</strong></td>
<td><strong>83%</strong></td>
</tr>
</tbody>
</table>

Engineering (LCSE) model as discussed in Chapter 2, “Grammars as Contracts”. For the implementation a significant amount of code is reused from the XT bundle, including generators for obtaining language-specific libraries and full-fledged components for parsing and pretty-printing.

Figure 6.7 displays the autobundle package, its constituent components, and the components it reuses (see Section 3.6 on page 42 for information about component diagrams). The autobundle package implements 6 components and reuses 10 components from 6 different packages.

Table 6.3 depicts component sizes and reuse levels of the autobundle package. The table shows that the implementation consists of approximately 6,300 lines of code, of which more than 4,600 lines are reused. This yields a reuse level between 73% and 83%. Section 3.6 justifies these numbers and describes how they are obtained by analyzing component implementations.

**Future work**  We depend on backwards compatibility of software packages. This requirement is hard to enforce and weakening it is an interesting topic for further research. The other requirement that we depend on now, is the use of autoconf and automake, which implement a standard configuration and build interface. We have ideas for a generic approach to hide component-specific build and configuration procedures behind standardized interfaces, but this still requires additional research.

**Acknowledgments**  We thank Arie van Deursen, Paul Klint, Leon Moonen, and Joost Visser for valuable discussions and feedback on earlier versions of this chapter.