Centrum voor Wiskunde en Informatica

REPORTRAPPORT

SEN

Software Engineering

Software ENgineering

Source tree composition

M. de Jonge

Report SEN-R0204 February 28, 2002

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-369X

Source Tree Composition

Merijn de Jonge
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
`http://www.cwi.nl/~mdejonge/`

ABSTRACT
Dividing software systems in components improves software reusability as well as software maintainability. Components live at several levels, we concentrate on the implementation level where components are formed by source files, divided over directory structures.

Such source code components are usually strongly coupled in the directory structure of a software system. Their compilation is usually controlled by a single global build process. This entangling of source trees and build processes often makes reuse of source code components in different software systems difficult. It also makes software systems inflexible because integration of additional source code components in source trees and build processes is difficult.

This paper's subject is to increase software reuse by decreasing coupling of source code components. It is achieved by automized assembly of software systems from reusable source code components and involves integration of source trees, build processes, and configuration processes. Application domains include *generative programming*, *product-line architectures*, and *commercial off-the-shelf (COTS)* software engineering.

1998 ACM Computing Classification System: D.2.2, D.2.3, D.2.7, D.2.9, D.2.11, D.2.12, D.12.13, D.2.m

Keywords and Phrases: source tree composition, reuse, kernel, linux, components, software product lines

Note: To appear in the proceedings of the Seventh International Conference on Software Reuse (ICSR7), 2002.
Note: Work carried out under CWI project SEN1.2, Domain-Specific Languages, sponsored by the Telematica Instituut.

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 [23]; (iii) it complicates the process of building self-contained distributions from a system and all its components. Package managers (such as RPM [1]) 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 [23, 29]. They also provide restricted control over a component's configuration. All these complicating factors hamper software reuse and negatively influence granularity of reuse [28].

We argue that *source code components* (as alternative to binary components) can improve software reuse in 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 Sect. 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 paper.

This paper is organized as follows. Section 2 motivates the need for advanced techniques to perform source tree composition. Section 3 describes terminology. Section 4 describes the process of source tree composition. Section 5 and 6 describe abstraction mechanisms over source trees and composite software systems. Section 7 describes automated source tree composition. It discusses the tool `autobundle`, online package bases, and product-line architectures. Section 8 describes experiences with source tree composition. Related work and concluding remarks are discussed in Sect. 9 and 10.

2. Motivation

In most software systems the constituent source code components are 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` [15] from the top-level Makefile for each source code component. Often, a global GNU `autoconf` [24] 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 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 [23]. 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 [28]. 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 Fig. 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) [3] system, source tree composition might be easy. This is because, ideally, a CM system administrates the build knowledge of all components, their dependencies, etc., and is able to perform the composition automatically.[1]

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 [7, 28]. 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 distributed as source. This is because: i) standardization of CM

[1] Observe that in practice, CM systems are often confused with version management systems. The latter do not administrate knowledge suitable for source tree composition.

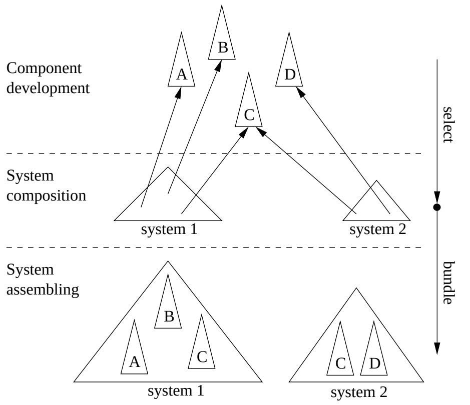


Figure 1: Component development, system composition, and system assembly with source code component reuse. Components are developed individually; compositions of components form systems, which are assembled to form *software bundles* (self-contained software systems).

systems is lacking [28, 32]; 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 paper because integration of their components is simplified, which makes them suitable for widespread use. Moreover, as we will see in Sect. 7.3, product-line architectures, which are concerned with assembling families of related applications, can also benefit from source tree composition.

3. TERMINOLOGY

System building is the process of deriving the targets of a software system (or software component) from source [9]. 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* [11]. Static 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 static 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.

```
package
identification
         name=CobolSQLTrans
      version=1.0
     location=http://www.coboltrans.org
         info=http://www.coboltrans.org/doc
  description='Transformation framework for COBOL with embedded SQL’
     keywords=cobol, sql, transformation, framework
configuration interface
  layout-preserving    ’Enable layout preserving transformations.’
requires
  cobol 0.5 with lang-ext=SQL
  asf   1.1 with traversals=on
  sglr  3.0
  gpp   2.0
```

Figure 2: An example package definition.

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 three 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 newer. Repeating the composition should therefore take as little effort as possible.

Invisible A source distribution of a system for non-developers should be offered as a unit (one source tree, one product), the internal structuring in source code components should not necessarily be visible. Integrating build and configuration processes of components is therefore a prerequisite.

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.

5. Definition of Single Source Trees

We propose *source code packages* as unit 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 [22]. We introduce *package definitions* as abstraction of source code packages. We developed a Domain Specific Language (DSL) to represent them.

```
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=aterm 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 3: Bundle definition obtained by normalizing the package definition of Fig. 2. This definition has been stripped due to space limitations.

An example is depicted in Fig. 2. It defines the software package CobolSQLTrans 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. In addition, also the URL where the package can be obtained, a short description of the package, and a list of keywords are recorded (see the identification section of Fig. 2).

Package configuration The configuration interface of a software package is defined in the configuration interface section. Partial configuration enforced by other components and composition of configuration interfaces is discussed in Sect. 6. For example, in Fig. 2, the configuration interface defines a single configuration parameter and a short usage description of this parameter. With this parameter, the CobolSQLTrans package can be configured with or without layout preserving transformations.

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 Sect. 6) to synthesize the complete set of packages that form a system. For example, the package of Fig. 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 (sglr), and pretty-printer (gpp).

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

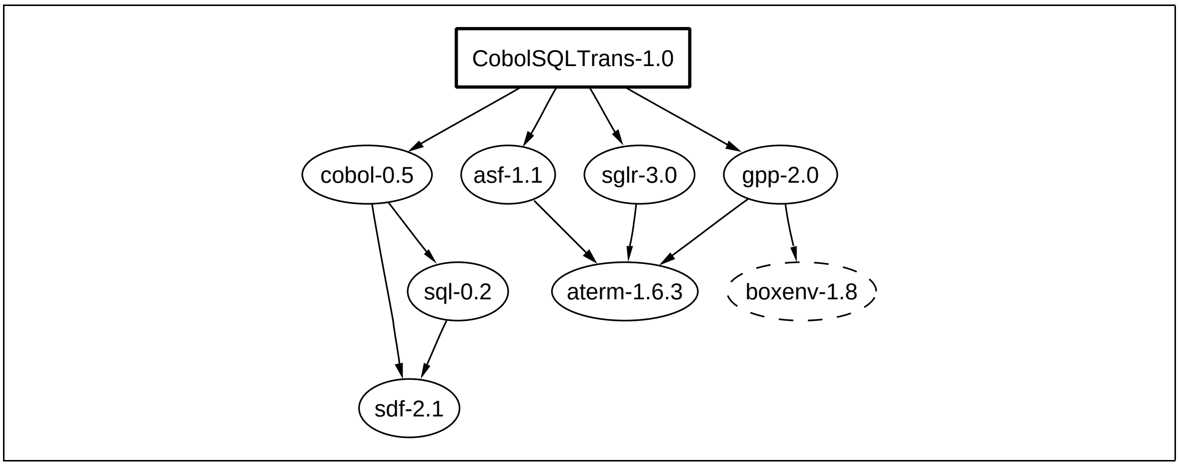


Figure 4: A package dependency graph for the Cobol transformation package of Fig. 2. The dashed circle denotes an unresolved package dependency.

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 Fig. 4). For instance, during normalization of the package definition of Fig. 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 package in Fig. 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 [31, 9, 32] for a discussion of version models). In accordance with [27], 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 determine 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 Fig. 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 up or by creating dedicated bootstrap packages).

Configuration propagation Each package definition that is collected during package normalization contains a (possible empty) set of configurable parameters, its configuration interface. Configurable parameters might get bound when the package is used by another which imposes a particular configuration. During normalization, this configuration is determined by collecting all the bindings of each package. For example, the CobolSQLTrans package of Fig. 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 Fig. 3). A conflicting configuration occurs when

Table 1: The following files are contained in a software bundle generated by `autobundle`.

`Makefile.am`	Top-level `automake` Makefile that integrates build processes of all bundled packages.
`configure.in`	An `autoconf` configuration script to perform central configuration of all packages in a software bundle.
`pkg-list`	A list of the packages of a bundle and their download locations.
`collect`	A tool that downloads, unpacks, and integrates the packages listed in `pkg-list`.
`README`	A file that briefly describes the software bundle and its packages.
`acinclude.m4`	A file containing extensions to `autoconf` functionality to make central configuration of packages possible.

a single parameter gets bound differently. Such configuration conflicts can easily be detected during package normalization.

Bundle interface construction A bundle's configuration interface 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 Fig. 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 Fig. 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.

7. Performing 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.

7.1 Autobundle

Package normalization and bundle generation are implemented by `autobundle`.[2] This tool produces an software bundle containing top-level configuration and build procedures, and a list of bundled packages with their download locations (see Table 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 Sect. 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 generated build interface of a bundle contains the actions `collect` to download and integrate the source trees of all packages, and `bundle` to also put them into a single source distribution.

[2] `autobundle` is free software and available for download at `http://www.cwi.nl/˜mdejonge/autobundle/`.

Table 2: These are the actions of the standardized build interface required by autobundle. In addition, autobundle also requires a tool configure to perform static configuration.

all	Build action to build all targets of a source code package.
install	Build action to install all targets.
clean	Build action to remove all targets and intermediate results.
dist	Build action to generate a source code distribution.
check	Build action to verify run-time behavior of the system.

The generated bundle is driven by make [15] and offers a standardized build interface (see Table 2). The build interface and corresponding build instructions are generated by autoconf [24] and automake [25]. 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 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 Fig. 5.

7.2 Online Package Bases

Dependency resolution during package normalization 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 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.

7.3 Product-Line Architectures

Online package bases allow 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 [17]) is accomplished by composing reusable components [2]. The building blocks from which applications are assembled are usually abstract requirements (consisting 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

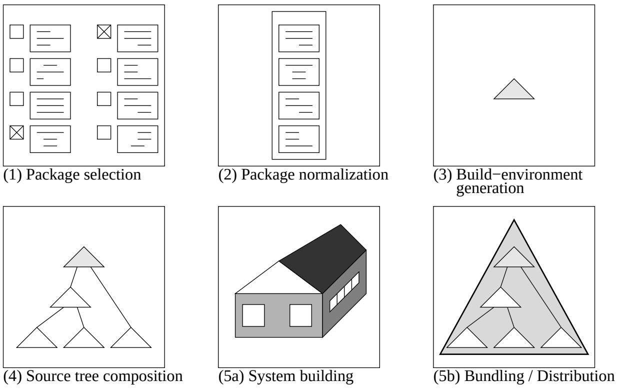

(1) Package selection
(2) Package normalization
(3) Build−environment generation
(4) Source tree composition
(5a) System building
(5b) Bundling / Distribution

Figure 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).

components) [10].

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 function as *configuration DSL* [11]. It then serves to capture configuration knowledge and to define mappings between the problem and 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.

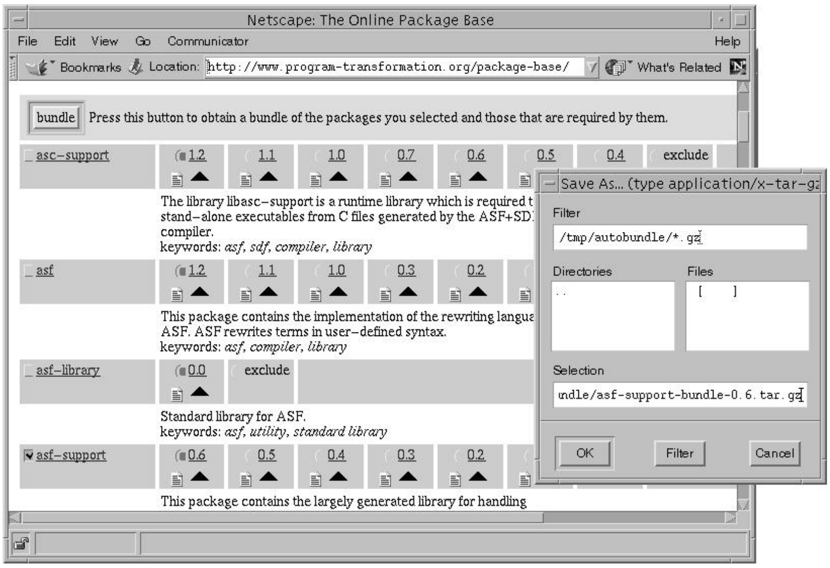
System assembling can be automated by `autobundle`. It normalizes a set of abstract components (features) and produces a source tree containing all corresponding implementation components and generates a (partial) configuration for them. Variation points of the assembled system can be configured statically 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 [21]. This makes assembling applications as easy as selecting the features of need.

8. Case Studies

System development We successfully applied source tree composition to the ASF+SDF Meta-Environment [4], 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.



Netscape: The Online Package Base
File Edit View Go Communicator Help
Bookmarks Location: http://www.program-transformation.org/package-base/ What's Related
bundle
asc-support 1.2 1.1 1.0 0.7 0.6 0.5 0.4 exclude
asf 1.2 1.1 1.0 0.3 0.2
asf-library 0.0 exclude
☑ asf-support 0.6 0.5 0.4 0.3 0.2
Save As... (type application/x-tar-gz
Filter
Directories
Files
Selection
OK
Filter
Cancel

Press this button to obtain a bundle of the packages you selected and those that are required by them.

The library libasc-support is a runtime library which is required t
stand-alone executables from C files generated by the ASF+SD
compiler.
keywords: *asf, sdf, compiler, library*

This package contains the implementation of the rewriting langua
ASF. ASF rewrites terms in user-defined syntax.
keywords: *asf, compiler, library*

Standard library for ASF.
keywords: *asf, utility, standard library*

This package contains the largely generated library for handling

Filter: /tmp/autobundle/*.gz

Directories: ..

Files: [ ]

Selection: undle/asf-support-bundle-0.6.tar.gz

Figure 6: Automated source tree composition at the Online Package Base. The Online Package Base is available at `http://www.program-transformation.org/package-base/`.

After we started using source tree composition techniques, reusability of our components greatly improved. This was demonstrated by the development of XT [20], a bundle of program transformation tools. 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.

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[3] (OPB).

With the OPB (see Fig. 6), 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[4] from AT&T, and components from a number of other research institutes.

Stratego compiler Recently, the Stratego compiler [30] has been split up in reusable packages (including the Stratego runtime system). The constituting components (developed at different institutes) are bundled with `autobundle` to form a stand-alone distribution of the compiler. With `autobundle` also more fine-grained

[3] Available at `http://www.program-transformation.org/package-base/`

[4] Available at `http://www.research.att.com/sw/tools/graphviz/`

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 are currently investigating the use of `autobundle` and online package bases in a commercial setting to transform the industrial application DocGen [14] into a product-line architecture [12]. 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) [13]. 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 visualizators). 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.

9. Related Work

Many articles, for instance [6, 5, 8] address build processes and tools to perform builds. Tools and techniques are discussed to solve limitations of traditional `make` [15], such as improving dependency resolution, build performance, and support for variant builds. Composition of source trees and build processes is not described.

Gunter [16] 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 [26] 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 [31]. For source tree composition this implies independence of individual build processes and therefore contradicts the approach of [26]. Since the approach of Miller entangles all components of the system, we believe that it will hamper software reuse.

This paper 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. The extra installation effort is problematic [29], even when partly automated by package managers (like RPM [1]). 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 paper has several similarities with the component model Koala [28, 27]. 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 [18, 19], 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 paper.

10. Concluding Remarks

This paper addresses software reuse based on source code components and software assembly using a technique called 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 provide 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.

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. The research in using `autobundle` for the construction of product-line architectures is in an early phase. Therefore, we will conduct more case studies in autobundle-based product-line architectures.

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

References

1. E. C. Bailey. *Maximum RPM*. Red Hat Software, Inc., 1997.
2. D. S. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *International Conference on Software Reuse*, volume 1844 of *LNCS*, pages 117–136. Springer-Verlag, 2000.
3. R. H. Berlack. *Software Configuration Management*. Wiley and Sons, New York, 1991.
4. M. van den Brand, A. van Deursen, J. Heering, H. de 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. In *Compiler Construction 2001 (CC 2001)*, volume 2027 of *LNCS*. Springer-Verlag, 2001.
5. P. Brereton and P. Singleton. Deductive software building. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in LNCS, pages 81–87. Springer-Verlag, Oct. 1995.
6. J. Buffenbarger and K. Gruel. A language for software subsystem composition. In *34th Annual Hawaii International Conference on System Sciences (HICSS-34)*. IEEE, 2001.
7. M. Cagan and A. Wright. Untangling configuration management. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in LNCS, pages 35–52. Springer-Verlag, 1995.
8. G. M. Clemm. The Odin system. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in LNCS, pages 241–2262. Springer-Verlag, Oct. 1995.
9. R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.
10. K. Czarnecki and U. W. Eisenecker. Components and generative programming. In O. Nierstrasz and M. Lemoine, editors, *ESEC/FSE ’99*, volume 1687 of *LNCS*, pages 2–19. Springer-Verlag / ACM Press, 1999.
11. K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools, and Applications*. Addison-Wesley, 2000.
12. A. van Deursen, M. de Jonge, and T. Kuipers. Feature-based product line instantiation using source-level packages. submitted for publication, january 2002.

13. A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 2001.

14. A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings; IEEE International Conference on Software Maintenance*, pages 40–49. IEEE Computer Society Press, 1999.

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

16. C. A. Gunter. Abstracting dependencies between software configuration items. *ACM Transactions on Software Engineering and Methodology*, 9(1):94–131, Jan. 2000.

17. J. van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In R. Kazman, P. Kruchten, C. Verhoef, and H. van Vliet, editors, *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pages 45–54. IEEE, 2001.

18. A. van der Hoek, R. S. Hall, D. Heimbigner, and A. L. Wolf. Software release management. In M. Jazayeri and H. Schauer, editors, *ESEC/FSE '97*, volume 1301 of *LNCS*, pages 159–175. Springer / ACM Press, 1997.

19. A. van der Hoek and A. L. Wolf. Software release management for component-based software, 2001. (In preparation).

20. M. de Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In M. van den Brand and D. Parigot, editors, *Proceedings of Language Descriptions, Tools and Applications (LDTA 2001)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

21. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, november 1990.

22. C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

23. Y.-J. Lin and S. P. Reiss. Configuration management in terms of modules. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in LNCS, pages 101–117. Springer-Verlag, 1995.

24. D. Mackenzie and B. Elliston. Autoconf: Generating automatic configuration scripts, 1998. `http://www.gnu.org/manual/autoconf/`.

25. D. Mackenzie and T. Tromey. Automake, 2001. `http://www.gnu.org/manual/automake/`.

26. P. Miller. Recursive make considered harmful, 1997. `http://www.pcug.org.au/~millerp/rmch/recu-make-cons-harm.html`.

27. R. van Ommering. Configuration management in component based product populations. In *Tenth International Workshop on Software Configuration Management (SCM-10)*, 2001. `http://www.ics.uci.edu/~andre/scm10/papers/ommering.pdf`.

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

29. D. B. Tucker and S. Krishnamurthi. Applying module system research to package management. In *Tenth International Workshop on Software Configuration Management (SCM-10)*, 2001. `http://www.ics.uci.edu/~andre/scm10/papers/tucker.pdf`.

30. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, May 2001.

31. D. Whitgift. *Methods and Tools for Software Configuration Management*. John Wiley & Sons, 1991.

32. A. Zeller and G. Snelting. Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398–441, Oct. 1997.