Build-level Component-Based Software Engineering

Merijn de Jonge

Technical Report UU-CS-2004-046
Institute of Information and Computing Sciences
Utrecht University

October 27, 2004

ISSN 0924-3275

Address:
Merijn de Jonge
mdejonge@cs.uu.nl
http://www.cs.uu.nl/~mdejonge

Institute of Information and Computing Sciences
Utrecht University
P.O.Box 80089
3508 TB Utrecht

Build-level Component-Based Software Engineering

Merijn de Jonge[*,†]

Utrecht University, P.O. Box 80089, 3508 TB Utrecht, The Netherlands

SUMMARY

A lot of potentially reusable functionality exists in current software systems that does not become available for reuse. An important reason is that while at the functional level well-known modularization principles are applied for structuring functionality into modules, this is not the case at the build level for structuring files in directories. This leads to a situation where files are entangled in directory hierarchies and build processes, making it hard to extract functionality and to make functionality suitable for reuse. Consequently, software may not become available for reuse at all, or only in rather large chunks of functionality, which may lead to extra software dependencies.

We propose to improve this situation by applying component-based software engineering (CBSE) principles to the build level. Build-level components serve to make artefacts reusable, while build-level composition techniques serve to assemble complete applications. This combines advantages of having small components (e.g., reuse) and large systems (e.g., ease of deployment). In this article we explore build-level CBSE in full detail. We discuss theory and practice of build-level development, deployment and composition techniques. Case studies demonstrate the feasibility of our techniques.

KEY WORDS: CBSE, software component, software reuse, software construction, software engineering, source tree composition

1. Introduction

Software reuse between software systems is sub-optimal: functionality that is potentially reusable, or functionality that is reused within a single software system, often does not become available for reuse in other systems. Consequently, either a complete system has to be reused, if only part of its functionality is needed, or this functionality has to be extracted from the system with rather adhoc technology. Reuse of a complete software system may introduce extra software dependencies, or may

[*]Correspondence to: Utrecht University, P.O. Box 80089, 3508 TB Utrecht, The Netherlands
[†]E-mail: mdejonge@cs.uu.nl
Contract/grant sponsor: This research was sponsored in part by the Dutch National Research Organization (NWO), Jacquard project TraCE.

complicate the software construction and the software deployment process. Extracting functionality may be too complicated and may not outweigh the benefits of reusing the functionality. It is therefore likely that reuse will not take place at all.

Software systems are usually not developed as collections of individually deployable and reusable building blocks because software construction processes (e.g., MAKEFILES) are hard to develop. Moreover, tool support is lacking to assist in developing such processes, and in composing them for the construction of complete software systems. Finally, building a large software system via a single construction process takes less effort then building it via many separate ones. It is therefore not feasible or desirable to develop separate construction processes for each reusable portion of functionality. As a result, they are developed by hand for a software system as a whole. Extracting reusable functionality from such software systems requires collecting all required source files, and extracting the appropriate parts from the construction process. This is extremely difficult due to the entangling of source files and construction processes, and because construction processes are hard to understand.

We claim that the aforementioned problems are caused by a lack of abstraction mechanisms at the level of files, directories, build, and configuration processes (i.e., the *build level*). This forms a significant factor that hampers software reuse. To improve this situation, we propose to adopt principles from Component-based Software Engineering (CBSE) at the build level.

A prerequisite for component technology is modularity [52]. Already in 1972, Parnas introduced the modularization principles of minimizing coupling between modules and maximizing cohesion within modules [47]. The former principle states that dependencies between modules should be minimized, the latter principle states that strongly related artefacts belong to the same module. These principles are well understood at the functional level for structuring functionality in functions or methods and in modules or classes. Unfortunately, they are usually not applied at the build level for structuring modules and classes in directories. Often, bad programming practice like strong coupling and weak cohesion therefore moves from the functional level to the build level. As a result, many software systems consist of large collections of files that are structured rather ad-hoc into directory hierarchies. Between these directories a lot of references exist (= strong coupling) and directories often contain too many files (= weak cohesion). Build knowledge gets unnecessarily complicated due to improper structuring into monolithic configuration files and build scripts.

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

In CBSE, functionality is only accessed via well-defined interfaces and one cannot depend on the internal structure of components. We propose to apply CBSE principles to the build-level, such that all component access occurs via build-level interfaces and dependencies on internal directory structures can be dropped. This will bring modularity and proper abstraction mechanisms to the build-level. Consequently, compositionality of build-level artefacts improves, giving rise to build-level components. Build-level components can be developed and deployed individually and promote software reuse across software systems. With accompanying composition techniques we will be able to combine the benefit of small components (reuse) and large systems (ease of deployment).

In this article we will explore build-level CBSE in detail. We will develop theory and infrastructure for the development, deployment, and composition of build-level components. The goal is to remove

build-level barriers that stand in the way of effective software reuse crossing application, group, and institute boundaries. We demonstrate the effectiveness of our approach in several case studies. The work presented in this article builds forth on our earlier work [32, 33, 35].

The article is structured as follows. Section 2 describes the build-level and introduces the concept of build-level components. Section 3 addresses build-level component deployment. Composition of build-level components is discussed in Section 4 and 5. Section 6 discusses CBSE with build-level components. Section 7 describes case studies. Section 8 discusses some generalizations of our ideas. Related work is discussed in Section 9 and in Section 10 we summarize our results.

2. The build level

The build level is concerned with i) the files that constitute a software system (such as source and documentation files); ii) the structuring of these files in directories; iii) the relations (i.e., dependencies) between these files and directories; and iv) the software construction process (i.e., the process to build a software system from its source files). The build level has file-granularity. That is, files are atomic units and directories are containers to group and structure files.

The full collection of (source) files of a software system, structured in directories, together with construction rules to build the software system is usually called a *source tree*. Consequently, source trees play a prominent role at the build level. Although in its purest form a source tree only contains source files, the build level is not restricted to source files only. Since a compiled artefact can be seen as a cached result of the construction process [17], partly or fully constructed software systems are of equal importance at the build level as source files. Hence, the build-level is concerned with software systems being in source or in binary form or in any form between.

2.1. Software construction

Software construction is the process of building a software system from its source files. Software construction consists of a (compile-time) *configuration process* and a *build process*. A build process deals with building the software system from its source files (for instance by compilation). The configuration process controls the build process by binding build-time variation points [29].

The software build process A build process typically consists of a sequence of instructions to compile source to object files and to link object files and libraries to executable programs. This sequence of instructions can be executed manually or automatically by a *build manager*. Build instructions cannot be executed in arbitrary order. For instance, before linking object files into an application the object files have to be created by the compiler. Consequently, a build process has a specific *build order*. This order is obtained from a *dependency graph*, which is formed from *build dependencies*. Build dependencies define what the inputs of build instructions are. A dependency graph also serves *build optimization* to minimize the number of required build instructions when some artefacts have changed.

The complexity of build processes make build managers prerequisite tools. Build managers automate build processes defined in *build definitions*. A build definition is a formal description of a build process. This can be just a sequence of tool invocations, or a more advanced description in terms of *targets* (what

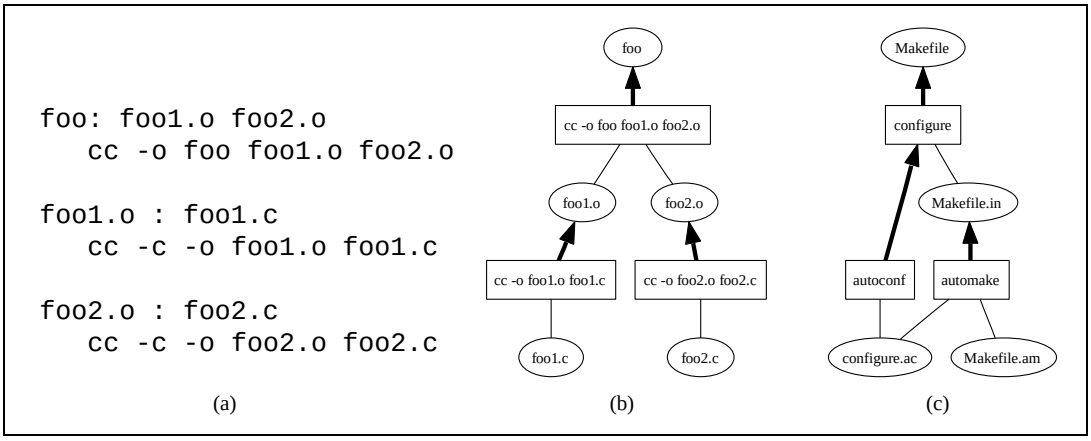

Figure 1. (a) An example MAKEFILE. (b) The corresponding dependency graph. (c) A simplified overview of build and configuration process generation with Autotools.

needs to be build), *dependencies* (what is needed to build a target), and *instructions* (how the generate targets from their dependencies).

MAKE [19], or one of its derivates, is the most commonly used build manager. Build definitions for MAKE are specified in a file called `Makefile`. In its simplest form it contains rules consisting of a target pattern, a dependency pattern, and a sequence of instructions. For instance, the MAKEFILE in Figure 1(a) defines three rules. The first defines how `foo` can be built. It defines a dependency on `foo1.o` and `foo2.o`, and it defines that `foo` can be constructed from these files by running the C compiler `cc`. The last two rules define how `foo1.o` and `foo2.o` can be constructed from `foo1.c` and `foo2.c`, respectively. The build process is initiated by running `make foo` (or `make` for short).

Figure 1(b) depicts the dependency graph for the MAKEFILE of Figure 1(a). Edges in this graph denote dependencies, whereas arrows denote build instructions. From this graph we can directly determine a build order by linearizing the file nodes (depicted as ovals) in a depth first order. For instance, `foo1.c`, `foo2.c`, `foo1.o`, `foo2.o`, `foo`. The dependency graph also shows which targets need to be built when a file has changed (i.e., all nodes on the path from the changed file to the root of the dependency graph). For instance, `foo1.o` does not need to be reconstructed when `foo2.c` has changed.

The software configuration process This controls the build process by binding build-time variability parameters. These parameters can be categorized as *environment configuration* or *functional configuration*. The first type deals with adapting a software system and its build process to a build environment. Examples are: selecting a compiler and setting compiler switches, specifying locations of required libraries, header files, and additional tools that are used in the build process or in the application. The second type of configuration deals with controlling the functionality of a software

system. Typical examples are: debugging that can be toggled on or off, the set of drivers that should be compiled into the system, or the customer-specific set of features that a system should support.

There are various ways to bind variability parameters. The most frequently used mechanisms are to specify variability bindings in a configuration file or in a build definition (e.g., in a MAKEFILE). Variability bindings can be specified manually or automatically. Manual binding is the traditional way and often implies editing numerous configuration files and MAKEFILES, sometimes scattered around in a source tree. Automated variability binding is performed by configuration tools which instantiate or generate MAKEFILES or configuration files according to a set of variability bindings.

MAKEFILES support variability by means of variables. For instance, by using a variable to hold the name of the compiler, the build process can easily be configured to use another compiler:

```
CC    = gcc
foo2.o : foo2.c
   $(CC) -c -o foo2.o foo2.c
```

Functional behavior of applications can be controlled similarly with MAKEFILE variables. For instance:

```
CC    = cc
DEBUG = -DDEBUG
foo2.o : foo2.c
   $(CC) -c -o foo2.o foo2.c $(DEBUG)
```

This MAKEFILE passes the value of the variable `DEBUG` to the C-compiler. Conditional compilation techniques (e.g., a preprocessor) can now be used to include or exclude functionality depending on the value of the variable `DEBUG`.

2.2. GNU Autotools

MAKE suffers from numerous limitations. Of concern in this article are the following limitations:

- It is difficult to achieve completeness of dependencies in a MAKEFILE. If a dependency graph is incomplete, then the build order and the build process may be incorrect. The first leads to build failures, the latter to build results that are not consistent with the corresponding source files.
- It is difficult to correctly define all build instructions. Defining how to compile a C program might not be so complex, but defining how a dynamic library should be compiled certainly is. Since abstraction mechanisms offered by MAKE are minimal, one has to write these instructions over and over again, which is a lot of work, easily leads to mistakes, and is hard to maintain.
- It is extremely difficult to write portable MAKEFILES. Again, due to lacking abstraction mechanisms, one has to deal with low-level compiler switches, which differ between compilers and between platforms. Using only variables in MAKEFILES to support this form of variability is certainly not sufficient.

There exist several approaches to deal with these issues. In this article we will make use of GNU Autotools because this collection of build-level tools is both widely used and freely available. Below we briefly describe the two tools from this collection that we use.

AUTOMAKE [39] is a MAKEFILE generator. Its input is a high-level declarative build process definition which specifies what the sources and targets are, not what the instructions are to produce the targets. The output of AUTOMAKE is a MAKEFILE with all necessary build instructions defined in a portable way. The generated MAKEFILES conform to the GNU MAKEFILE Standard [21]. For example, with AUTOMAKE, the build process definition of Figure 1(a) can be reduced to:

```
bin_PROGRAMS = foo
foo_SOURCES  = foo1.c foo2.c
```

The resulting MAKEFILE not only knows how to build `foo` from its source files, but also how and where to install (or de-install) `foo` in the file system, how to bundle its source files into a source distribution, and more. The tool AUTOMAKE can also track the source code dependencies for a number of programming languages. Unfortunately, dependency tracking is language-specific and therefore not supported for all languages, and completeness is not guaranteed.

AUTOCONF [38] is a configuration script generator. The generated script understands configuration switches and is able to perform automatic environment checks. The variability bindings resulting from values passed to configuration switches and environment checks are used to instantiate MAKEFILES. Many of these bindings are used in the rules in the MAKEFILES generated by AUTOMAKE. An AUTOCONF definition for program `foo` of Figure 1(a) might look like:

```
AC_INIT([foo],[0.1],[maintainer@some.site.org])
AM_INIT_AUTOMAKE
AC_PROG_CC
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

This small configuration script defines the name, version, and maintainer of `foo`, it defines that a C compiler is required, and that the file `Makefile` should be instantiated with the variability bindings.

Figure 1(c) gives an overview of how configuration scripts and MAKEFILES are generated by Autotools. A configuration description is defined in a file called `configure.ac`. The AUTOMAKE build definition is defined in `Makefile.am`. AUTOCONF reads the configure script definition and produces the configuration script `configure`. The tool AUTOMAKE reads `Makefile.am` and `configure.ac` and produces `Makefile.in`. The latter file will be instantiated with configuration parameter bindings when executing CONFIGURE. The tools AUTOCONF and AUTOMAKE serve to generate a build environment and are therefore executed by developers of a software system. The tools CONFIGURE and MAKE are part of the build environment and executed by people wishing to build the system. For reasons of clarity Figure 1(c) only gives a simplified picture of the tool chain and file generation process of Autotools.

2.3. Build-level abstractions with Autotools

Below we introduce abstraction mechanisms for build-level CBSE. We show that these mechanisms fit seamlessly in the Autotools model, which formed a strong motivation for using Autotools.

Build interface In addition to the concepts of MAKE discussed thus far (i.e., targets, dependencies, instructions, and variables), we will also use the concept of *phony targets*. A phony target is a make target that does not correspond to a file in the file system. Basically, a phony target forms one of the few abstraction mechanisms that can be used in MAKEFILES. A phony target is used to group other targets and to hide concrete target names. For instance, a MAKEFILE typically contains the phony target `all`, which, in the form of dependencies, defines what files should be produced by the build process. By running `make all`, a user can execute the build process without any internal knowldege of the build process. Similarly, with the phony target `clean`, a user can remove generated files, without having to know the names of these files.

Software build processes are usually activated via phony targets. Phony targets thus form the interface to build initiators. From now on, we call a phony target a *build action* and the set of build actions provided by a MAKEFILE a *build interface*. Unfortunately, build interfaces often differ among software systems, which hampers uniform software construction. We will therefore use a standardized build interface which is depicted in Table I(a). This interface is always implemented by MAKEFILES generated by AUTOMAKE.[2] Observe that there are many alternatives to the build interface of Table I(a) possible. Also observe that the use of AUTOMAKE is not essential for obtaining build processes that implement the build interface of Table I(a). Therefore, we are neither claiming that the interface of Table I(a) is the best nor that the use of AUTOMAKE is essential (see Section 8).

Variability parameter binding By running a CONFIGURE configuration tool, variability parameters will get bound. These parameter bindings are passed to the build process by substituting them in MAKEFILES. Each variability parameter thus corresponds to a MAKEFILE variable. Variability parameters can be declared in AUTOCONF configuration process definitions using the AUTOCONF `AC_SUBST` construct. With `AC_SUBST(DEBUG)` AUTOCONF is instructed to substitute the value of the variable `DEBUG` in generated MAKEFILES. Furthermore, it instructs AUTOMAKE to create a MAKEFILE variable `DEBUG` and to generate the template variable declaration `DEBUG = @DEBUG@` in MAKEFILE.IN files. If `DEBUG` will get bound to `true` during the configuration process, then the generated MAKEFILE will contain the variable declaration `DEBUG = true`. The binding of this variability parameter can be obtained in the MAKEFILE as `$(DEBUG)`.

Configuration interface We explained how variability parameters of a build process can be bound by a CONFIGURE tool generated by AUTOCONF. These parameters have first to be bound during a configuration process. To that end, AUTOCONF supports two binding mechanisms. The first mechanism is via automatic environment checks. These inspect the environment of the computer on which the CONFIGURE tool is executed to find locations of files, check for the existence of libraries, and so on. In this article, we will not make use of automatic variability parameter binding via environment checks.

The second mechanism to bind variability parameters is via configuration switches. AUTOCONF supports two forms of configuration switches: boolean switches and argument switches. The first form is used to bind a parameter to either *yes* or *no*. For example, a boolean configuration switch can be defined for the `DEBUG` parameter as follows:

```
AC_ARG_ENABLE([debug],
  AS_HELP_STRING([--enable-debug],[Toggle debug support.]),
  [DEBUG=${enableval}],
  [DEBUG=no])
AC_SUBST([DEBUG])
```

The resulting CONFIGURE tool accepts the switches `--enable-debug` and `--disable-debug` and binds the variable `DEBUG` accordingly. The first assignment to the variable `DEBUG` is performed when one of these two switches is used, the second assignment otherwise. The variable `enableval` holds the value `yes` or `no`, depending on the switch that was used.

Argument switches are used to bind configuration parameters to arbitrary values. For instance, suppose that `STACK_SIZE` is a variability parameter, then an argument switch is created as follows:

[2] Actually, AUTOMAKE-generated MAKEFILES support more build actions, but these are not addressed in this article.

Table I. (a) Standardized build interface. (b) Syntax for variability parameter binding via configuration tool.

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

(a)

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

(b)

```
AC_ARG_WITH([stack-size],
  AS_HELP_STRING([--with-stack-size],[Set size of stack.]),
  [STACK_SIZE=${withval}],
  [STACK_SIZE=100])
AC_SUBST([STACK_SIZE])
```

This fragment adds the configuration switch `--with-stack-size` to a CONFIGURE tool. By passing e.g., `--with-stack-size=50` to CONFIGURE, the stack size parameter will be bound to 50. If the switch is not used, the parameter is bound to a default value of 100.

We will call the mechanism used to bind build-time variability parameters a *configuration interface*. Unfortunately, different software systems use different configuration mechanisms, which hampers uniform build-level configuration. Fortunately, the interface of configuration processes driven by AUTOCONF is standardized. This is because configuration is always performed with the CONFIGURE tool, which provides a consistent way and syntax to bind variability parameters (see Table I(b)). This forms a strong motivations for using AUTOCONF in this article.

2.4. Build-level Components

According to Szyperski [52], the characteristic properties of a component are that it: i) is a unit of independent deployment; ii) is a unit of third-party composition; iii) has no (externally) observable state. He gives the following definition of a component:

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

Component-based software engineering (CBSE) is mostly concerned with execution-level components (such as COM, CCM, or EJB components). We propose to apply CBSE principles also at the build level. Components are then formed by directories and serve as unit of composition.

Modularity is a prerequisite for component technology [52]. We observed that the modularization principles of minimal coupling and maximal cohesion not only exist at the functional-level (i.e., within

and between files and classes), but also at the build-level (i.e., within and between directories). We observe the following similarity:

	functional-level	*build-level*
atom	function / method	file
container	module / class	directory

Consequently, to apply CBSE principles to the build-level, build-level modularity is a prerequisite. Despite the similarity with the functional level, it turns out that the modularization principles are only rarely applied to the build-level. Strong coupling is caused by having too many directory references, whereas weak cohesion is caused when too many loosely related files are contained in a single directory. In practice, files and directories are therefore often strongly coupled and have weak cohesion. As a result, existing implementations of software systems are not structured in build-level components, and neither independent deployment nor third party composition is feasible in practice.

To improve this situation, we propose to apply the CBSE principle of accessing components only trough interfaces to the build level. To that end, we adopt the abstraction mechanisms that we introduced in the previous section as key concepts at the build level. All component access then occurs via build, configuration, and requires interfaces. Build interfaces serve to execute actions of a component's build process (e.g., to build or install a component), configuration interfaces serve to control how a component should be build (i.e., to support build-time variability). Requires interfaces serve to bind dependencies on other components. Dependency parameters allow *late binding* by third-parties. Since all component access occurs via interfaces, build-level components can be independently deployed and their internal structure can safely be changed. Directories with these properties satisfy the component definition of [52] and can be used for build-level CBSE.

Many existing software systems break CBSE principles at the build level. This results in stronger coupling and weaker cohesion. In [35] we analyzed typical build-level practices (smells) that break these principles. We introduced 9 build-level concepts and identified build-level smells for each of them. To bring CBSE principles to the build level, we proposed build-level development rules for each concept, and, if applicable, discussed their implementation with Autotools. When applied, these development rules turn build-level artefacts into 'true' components. Table II enumerates the build-level development rules that we defined and which we will use in the remainder of this article.

3. Build-level deployment

Build-level components are compositional by a third-party and independently deployable. This section focuses on deployment of build-level components. We discuss build-level composition in Section 4.

3.1. Package as unit of deployment

Build-level artefacts are often entangled in Software Configuration Managements (SCM) systems [35]. To allow wide-spread use, build-level components should be deployable independently of an SCM system. This is accomplished with software release management [28], which makes components available without SCM system access. Release management should include a version scheme that relates component releases to SCM revisions. A *release* of a build-level component is an independent

Table II. Build-level component development rules.

1. A component has directory granularity.
2. Circular dependencies between components should be prevented.
3. Software building is performed via standardized build interface.
4. Binding of compile-time variability via standardized configuration interface.
5. Binding of build-level dependencies via requires interface.
6. A build process definition for each component.
7. A configuration process definition for each component.
8. Deployment of components in the form of build-level packages.
9. Composition of build-level components is automated.

collection of all the files that belong to the component. This includes source files, a build process, a configuration process, documentation, and so on. In the remainder of this article we call such a versioned release (or distribution) of a build-level component a *build-level package* (or *package* for short). A package serves two purposes: i) it decouples a build-level component from an SCM system; ii) it makes different versions of a component distinguishable. Basically, a package forms a unit of deployment, while a build-level component forms a unit of composition. Build-level packages have the following properties:

Unique version Each release of a package is given a unique version. This enables the coexistence of different versions of a package and ensures that they can be distinguished. If an SCM system controls the files of a build-level component, then the package version corresponds to a particular revision of the files in the SCM system.

Immutable Once a package has been released it should never change. If changing a released package would be allowed, then package releases can no longer uniquely be identified because different releases of a package may exist with the same version number. Furthermore, making a change to an already released package may break software that uses that specific release. Finally, the correspondence with an SCM system revision disappears. This would complicate the reproduction of packages.

Backwards-compatible In accordance with [45], we require that packages are *backwards compatible*. This ensures 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, then a new package (with a different name) should be created.

Controlled quality Packages give control over what software comes available for reuse. Typically, a package is only released when certain quality standards are met. For instance when all tests for a component succeed. This way, reuse of a malfunctioning component under development is prevented. Package-based reuse so increases the overall quality of reuse-based software systems.

Infinite life-time A package should (in principle) never disappear once is has been released. The reason is that, unless reuse of the package can be traced, there is no way to determine whether

```
package
identification
        name=CobolSQLTrans
     version=1.0
    location=http://www.cobol-trans.org
        info=http://www.cobol-trans.org/doc
  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 of a package definition in the Package Definition Language (PDL).

the package is still in use or not. If it is in use, then a need may exist to reconstruct the using system and, consequently, the reconstruction would fail if the package is no longer available. In practice, a package should retire only when it is no longer used within its reuse scope, or when economical or practical reasons demand a shorter life time.

These properties make software reuse more robust because they help to guarantee functional stability as well as liveness of packages. Observe that we did not prescribe how a package should be released, or when it should be released. This is part of *software release management* [28] and not further addressed in this article. We simply assume that packages are made available via a file system or via some network service such as HTTP or FTP.

3.2. Package definitions

A build-level component has a name, a standardized build interface, and a standardized configuration interface that defines variability and dependency parameters. A build-level package has meta-information such as a version number. For automatic processing of build-level components, such as automated composition (see Section 4), these characterizing properties of packages need to be formalized. This is accomplished with *package definitions*, which are formal abstractions for build-level packages. We defined the *Package Definition Language* (PDL) for declaring packages. See Figure 2 for an example package definition in PDL.

A package definition only captures package-specific information. For instance, build results are stored in standard locations (see Section 4.4) and need not be specified. Build interfaces are also not explictly declared because they are equal for all components (see Section 2.4).

A package definition consists of sections, which define different properties of a package. The following sections are supported:

Identification This section defines the name and version number of a package. Furthermore, it contains a description of the package and a list of keywords that characterizes the package.

```
module Package
exports
  context-free syntax
    "package" Identification Interface Requires                    → PackageDef

    "identification" Name Version Location Info Description Keywords → Identification
    "name" "=" Literal                                             → Name
    "version" "=" Version                                          → Version
    "location" "=" URL?                                            → Location
    "info" "=" URL?                                                → Info
    "description" "=" Text?                                        → Description
    "keywords" "=" {Literal+ ","}*                                 → Keywords

    "configuration" "interface" Option*                            → Interface
    Literal Text                                                   → Option

    "requires" ReqPackage*                                         → Requires
    Literal Version ( "with" Switch+ )?                            → ReqPackage
    Literal                                                        → Switch
    Literal "=" Literal                                            → Switch
```

Figure 3. Context-free syntax of the Package Definition Language (PDL).

Finally, this section contains pointers (as URLs) to the location where the package can be obtained and to documentation about the package (e.g., to the package's home page).

Configuration interface This section defines the variability parameters of a build-level component. The section consists of a list of pairs, defining the name of a variability parameter and a description of the parameter.

Requires This section defines dependency parameters and variability parameter bindings for required components. The section consists of a list of component names, version requirements, and optional variability parameter bindings. Since we assume backwards-compatibility of build-level components, a version requirement of *x* is not strict but corresponds to version *x* and above.

The concrete-syntax definition of PDL is depicted in Figure 3. This syntax definition is defined in the syntax definition formalism SDF [26, 54]. Note that the orientation of productions is flipped with respect to BNF notation. Below follows a short discussion of PDL.

Concrete packages The location field of a package is optional. We call a package definition *concrete* if a location is defined, which means that an implementation for the package exists. The location of a package is defined as an URL, which means that the package definition and the package itself can be stored in different locations. This is an important property of package definitions as will become clear in Section 3.3.

Abstract packages We call a package definition *abstract* if a location URL in a package definition is absent. It implies that there exists no implementation for that package. Abstract packages are

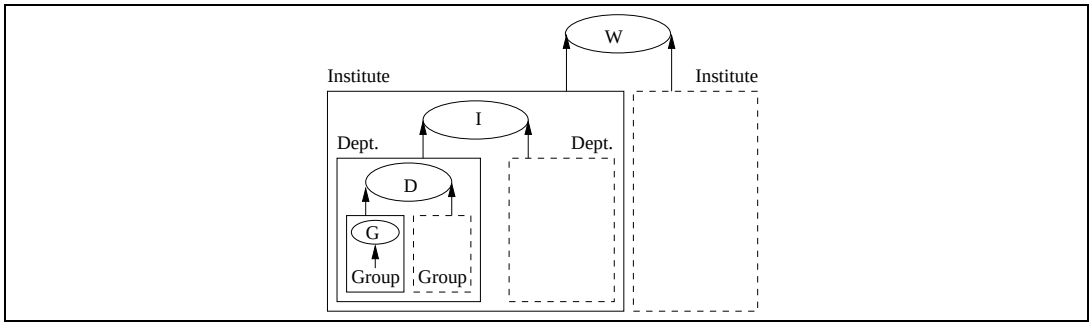

Figure 4. Package repositories (depicted by ovals) and reuse scopes. Scopes are indicated by letters: W=world, I=institute, D=department, and G=group. Reuse scope is indicated by arrows. For instance, packages in a department repository have department-wide reuse scope (D) and are accessible by all groups in that department.

used to group packages. The packages to be grouped are defined in the 'requires' section of the abstract package definition. An abstract package definition thus forms an abstraction for groups of packages. This is a useful mechanism to gradually increase component granularity [16].

Multiple download protocols To increase the flexibility of package deployment, different protocols can be specified in the location URL. Typical protocols are: "http", "file", and "ftp". In Section 6 we will discuss two additional protocols ("cvs" and "svn").

The package definition depicted in Figure 2 defines version 1.0 of a fictional package for Cobol transformation. The package can be obtained from the WEB-site `www.cobol-trans.org`. The documentation of the package can also be obtained from that WEB-site. The package definition declares one variability parameter, `layout-preserving`, to toggle a domain feature for preserving layout during transformations. The package requires four other packages: `cobol` with version $\geq$ 0.5, `asf` $\geq$ 1.1, `sglr` $\geq$ 3.0, and `gpp` $\geq$ 2.0. The `cobol` package is configured to support Cobol with embedded SQL. The `asf` package is configured with the `traversals` feature turned on.

3.3. Package repositories

Package definitions are stored in *package repositories*. A package repository is a directory containing collections of package definition files. The ability to separate package definitions from package implementations allows that package definitions can be grouped together in a repository, while each package definition contains a pointer to its implementation. This way, packages, originating from a diverse collection of locations, can be managed as a single collection. Such collections can be accessed by composition tools, giving rise to reuse scopes, and be transformed into user-friendly package bases.

Reuse Scope All package access occurs through package repositories. Consequently, a build-level component is only a candidate for reuse if its package definition is contained in a repository that is accessible by a composition tool. Package repositories thus define a *reuse scope*. We define the

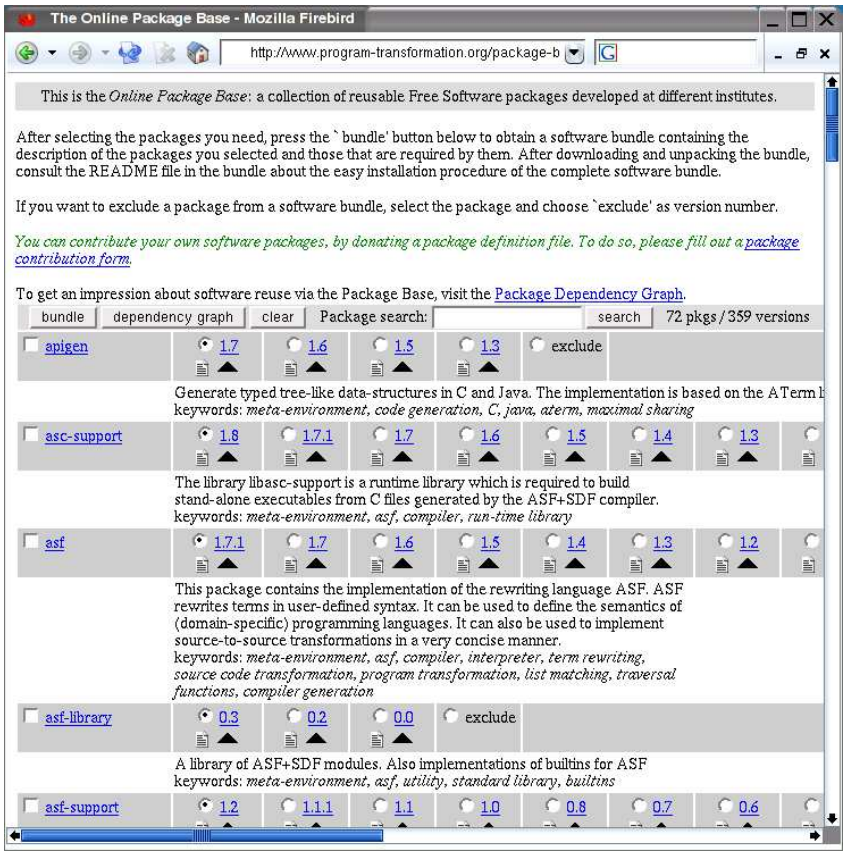

Figure 5. The Online Package Base (OPB), a user-friendly interface to package repositories [46].

reuse scope of a package as the space in which its package definition is visible to (or accessible by) composition tools. Different scopes can be defined, such as *group-wide*, *department-wide*, *institute-wide*, or *world-wide*.

Multiple repositories can be combined to extend the search space of composition tools and to increase the number of packages that can be reused. Access to packages can be carefully controlled by using different scopes. This is depicted in Figure 4, which shows four different types of reuse scopes. All package repositories that are transitively reachable from a group can be combined. Consequently, members of a group can use any package from the package repositories that are visible by the group. There is no limit on the number of package repositories that can exist or that can be combined. For example, a department may create multiple package repositories to separate package definitions from different projects. This is not shown in Figure 4. The ability to compose different package repositories gives fine control over the set of packages that can be reused.

module Bundle
imports Package[3.2]

exports
context-free syntax
"bundle" Identification Interface Requires Bundles → Bundle
"bundles" PackageUse***** → Bundles
"package" Identification Configuration → PackageUse
"configuration" Switch***** → Configuration

Figure 6. Context-free syntax of the Bundle Definition Language (BDL).

Online Package Bases The formal representation of packages presented in Section 3.2, allows for the development of tools that can increase the accessibility of package repositories. For instance, we developed search capabilities to find packages in large collections of package definitions. Furthermore, package definitions can be presented in a user-friendly way via HTML-forms, from which packages can be queried, inspected, and selected. We call a package repository accessible via the WEB an *Online Package Base* (OPB). Figure 5 depicts the OPB located at `www.program-transformation.org/package-base`. Its packages have a world-wide reuse scope.

Online package bases offer a user-friendly interface to package repositories. They can be queried by typing keywords in the search field, and be extended by uploading new package definitions. Furthermore, packages can be retrieved by clicking on a version number, package documentation can be viewed, dependencies can be visualized, and package definitions can be inspected. Finally, compositions of packages can be generated. The latter will be discussed in Section 4. In Section 5 we discuss, among others, the architecture of online packages bases and the tools needed to produce them.

4. Build-level component composition

In the previous sections we discussed build-level components and their deployment. This section is concerned with composing build-level components to form real applications. Build-level component composition involves merging all files and directories of the constituent components, as well as their build and configuration processes. The result is a directory hierarchy with a single build and configuration process. Typical deployment tasks, such as building and installing, can be performed on the composition as a whole, rather than on each component individually. Likewise, the composition can be released and distributed as a single unit.

4.1. Software bundles

We call a composition of build-level components a *software bundle* (or bundle for short). A software bundle consists of the files and directories of the constituent components, a top-level build process, and a top-level configuration process. The top-level build process implements the standard build

```
bundle
  identification
    name=CobolSQLTrans-bundle version=1.0 ...
  configuration interface
    layout-preserving 'Enable layout preserving transformations.'
  requires
  bundles
    package
      identification
        name=aterm version=1.6.3 ...
      configuration
        dbg=on gcc=on
    package
      identification
        name=pt-support version=0.6 ...
      configuration
        toolbus=${prefix} aterm=${prefix} toolbuslib=${prefix}
    package
      identification
        name=sglr version=3.2 ...
      configuration
        pt-support=${prefix} aterm=${prefix}
    package
      identification
        name=cobol version=0.5 ...
      configuration
        lang-ext=SQL sglr=${prefix}
    package
      identification
        name=asf version=1.1 ...
      configuration
        traversals=on pt-support=${prefix} aterm=${prefix}
    package
      identification
        name=CobolSQLTrans version=1.0 ...
      configuration
        cobol=${prefix} asf=${prefix} sglr=${prefix} gpp=${prefix}
```

Figure 7. Example bundle definition in BDL. It has been stripped to save space.

interface of Table I(a) and unites the build processes of the components (see Section 4.8). The top-level configuration process implements a top-level configuration interface that is synthesized from the configuration interfaces of the components (see Section 4.9).

A *bundle definition* formalizes a software bundle. We defined the *Bundle Definition Language* (BDL) to declare bundle definitions. Its context-free syntax is depicted in Figure 6. Observe that the syntax of BDL reuses some constructs of PDL by importing the module Package defined in Section 3.2. Like package definitions, bundle definitions consist of sections. BDL distinguishes the following sections:

Identification Similar to PDL. The fields are automatically synthesized from a bundle's components.

Configuration interface The collection of unbound variability parameters (see Section 4.6).

Requires A collection of all unbound dependency parameters (see Section 4.2 and 4.7).

Bundles The ingredients of the software bundle as a list of instantiated package definitions.

Figure 7 depicts an example of a bundle definition consisting of the 'CobolSQLTrans' package from Figure 2 and the packages that it transitively requires. To save space, we slightly stripped the bundle definition (i.e., we do not show all bundled packages and dependency parameter bindings, and the identification sections are only partly shown).

Build-level composition consists of two steps. First, a composition is defined by synthesizing a bundle definition from the desired composition of build-level components (e.g., the bundle definition of Figure 7 when CobolSQLTrans is to be bundled). Then, a composition is performed by producing a directory hierarchy, build process and configuration process from the bundle definition. These steps are further discussed in the upcoming sections.

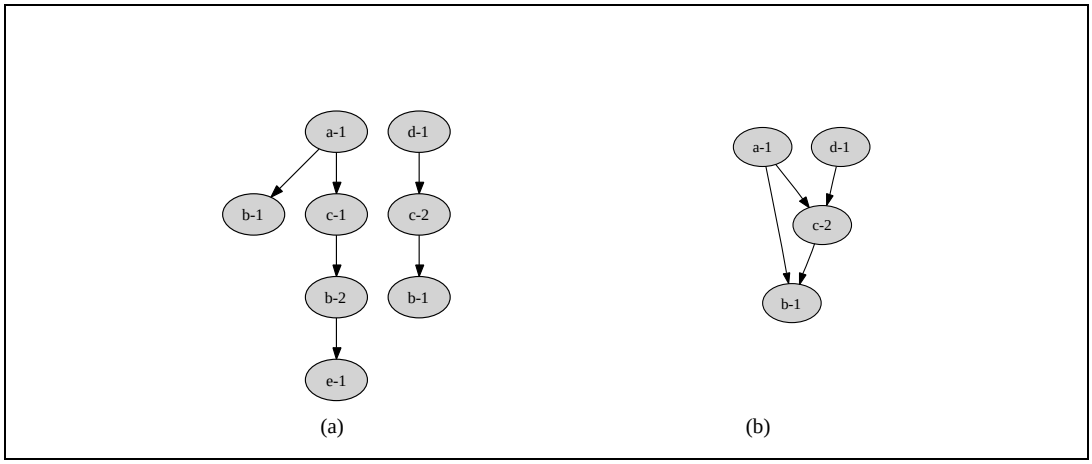

Figure 8. Version resolution.

4.2. Package normalization

The first step in synthesizing a bundle definition is called *package normalization*. It involves collecting all package definitions that are needed and unifying packages when multiple versions of a single package are required.

To collect all required package definitions we calculate the transitive closure of package dependencies. That is, given one or more packages, we recursively collect the package definitions for all packages that are contained in requires sections. Package definitions are retrieved from package repositories (see Section 3.3). The result is a package graph where nodes correspond to packages and edges to dependency relations. For instance, in Figure 8(a), the package graph is depicted for a composition of top-level packages ‘a-1’ and ‘d-1’. This package graph contains multiple versions of components, each having different dependencies. For instance, ‘b-1’ has no dependencies, while ‘b-2’ requires ‘e-1’. If a package definition is missing for a required package, than the corresponding node in the package graph is marked as an unresolved package dependency.

A bundle cannot contain multiple component instances. As a consequence, multiple versions of a component are not allowed. The package graph of Figure 8(a) clearly does not satisfy this constraint and the packages ‘b-1’ and ‘b-2’, as well as ‘c-1’ and ‘c-2’ need to be unified, such that only one instance of ‘b’ and ‘c’ is contained in the composition. We call this process *version resolution*. Recall from Section 3.1 that packages are backwards compatible. Consequently, if a package graph contains the packages p_i and p_j, where $i > j$, then we bundle p_i since it is compatible with the older version p_j.

If we apply this rule to the package graph of Figure 8(a), then we obtain the graph of Figure 8(b), which no longer contains the packages ‘b-2’ and ‘c-1’. Observe that although ‘b-2’ has a higher version than ‘b-1’, it is not contained in the graph of Figure 8(b). This is because ‘c-2’ replaces ‘c-1’, removing the only one dependency on ‘b-2’. Observe further that package ‘e-1’ is no longer contained in the

```
version-resolver(root,pkg-list) =
  deps ← strongest(pkg-deps(root))
  do
    pkgs ← ∅
    for each d ∈ deps
      pkgs ← pkgs ∪ resolve(d,pkg-list)
    deps' ← ∅
    for each p ∈ pkgs
      deps' ← deps' ∪ pkg-deps(p)
    deps' ← strongest(deps')
    if( deps ∩ deps' ≠ ∅)
      deps ← deps'
      continue
    else
      deps ← ∅
      for each p ∈ root ∪ pkgs
        deps ← deps ∪ pkg-deps(p)
      deps ← strongest(deps)
      if( deps ∩ deps' = ∅ )
        return pkgs
      else
        continue
```

strongest(*deps*) =
$\left\langle (n,v) \in deps | \forall_{(n,v') \in deps} v \geq v' \right\rangle$

resolve((*n*, *v*),*pkg-list*) =
Return the package *n* from *pkg-list* that meets version requirement *v*.

pkg-deps(*pkg*) =
Return the list of package dependencies of *pkg* as defined in the 'requires' section of its package definition.

Figure 9. Version resolution algorithm.

graph either. This is because 'e-1' is a dependency of 'b-2' and not of 'b-1'. We call a unified package graph, such as the one depicted in Figure 8(b), a *package dependency graph*. A package dependency graph gives full information about the ingredients of a software bundle and their inter-relations. For instance, the package dependency graph of the CobolSQLTrans bundle is depicted in Figure 10.

The version resolution algorithm is depicted in Figure 9. It is a fixed-point algorithm that traverses a package graph in a top-down fashion and determines the strongest version requirement of each component. The algorithm starts with the dependencies of a virtual root package that has dependencies to all top-level packages of a composition. For instance, for the composition of Figure 8, the root package would have a dependency on 'a-1' and 'd-1'. The algorithm continues until the set of dependencies reaches a fixed-point. This means that all edges of the package graph have been traversed and that no stronger version requirements exist. After reaching this fixed-point, the algorithm restarts once again, to make sure that unreachable nodes will disappear. As explained before, nodes may become unreachable as a consequence of different package dependencies.

4.3. Package linearization

A bundle definition contains a list of the packages that are bundled. This list follows from the package dependency graph by linearizing the nodes of resolved packages. Linearization implies flattening a package dependency graph and ordering the nodes such that only back references exist. For example, 'a-1' in Figure 8(b) depends on 'b-1' and 'c-2'. Consequently, 'b-1' and 'c-2' come before 'a-1'. Since

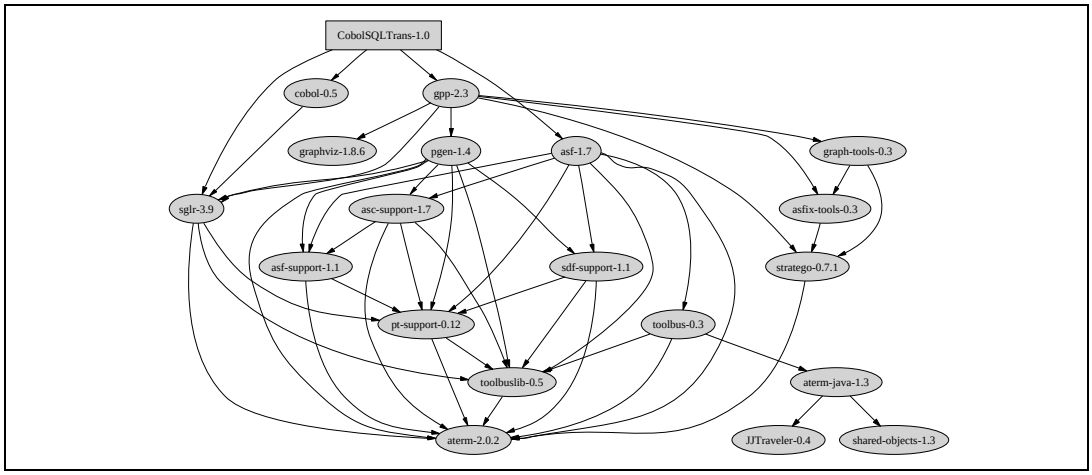

Figure 10. Complete package dependency graph for the bundle of Figure 7. The dependencies of a component are denoted by its outgoing edges.

'c-2' also depends on 'b-1', the three nodes will be ordered as 'b-1, c-2, a-1'. The packages 'a-1' and 'd-1' come in arbitrary order because there is no (transitive) dependency between them.

Linearization is important when referencing other components because a component must exist when it is referenced. For instance, 'a-1' in Figure 8(b) may refer in its build process to 'b-1' and 'c-2'. By linearization we can ensure that 'b-1' and 'c-2' are built *before* 'a-1'.

4.4. Dependency parameter binding

We store build results relative to some top-level directory with binaries in the subdirectory `bin`, libraries in `lib`, header files in `include`, etc. (according to the GNU Coding Standards [21] and the Filesystem Hierarchy Standard (FHS) [48]). This top-level directory is called the component's *installation prefix*, or *prefix* for short. For dependency parameter binding we need to know what the installation prefix of each component will be. Observe that the installation prefix of a component should not be a component property but a composition property. It should therefore be determined at composition time and bound by the composition mechanism/tool. There are several locations where components of a composition can be installed:

	relative to source tree	*relative to installation tree*
relative to component	`src/coboltrans/aterm`	`/usr/coboltrans/aterm`
relative to composition	`src/coboltrans`	`/usr/coboltrans`
composition independent	`src`	`/usr`

Component location Installation relative to a component (e.g., under `/usr/coboltrans/aterm` for the `aterm` component) has the advantage that build results of different components are nicely separated. Consequently, build results can clearly be distinguished and name clashes, due to different components having equally named build results, are prevented. The main drawback is that it is more difficult to reference build results because one has to know exactly by which component certain functionality is provided. If all references are synthesized automatically this is no problem, but if users make manual references (e.g., by executing a program), they may encounter difficulties in finding the correct locations. The advantage of installing the components of a software bundle composition independently (e.g, under `/usr`), is that user configuration is simple. That is, no special user configuration is necessary to use software installed under `/usr`. The drawbacks are that name clashes are more likely to occur than when software is installed in composition-specific directories (e.g., `/usr/coboltrans`) and that multiple versions of a system cannot simultaneously be installed. These drawbacks form a strong motivation in this article for installing components relative to a composition. This is not a fundamental decision and can easily be changed, as we did in different implementations of our composition tool.

Build-time installation As we explained in Section 4.3, a component needs to exist when it is referenced. Hence, components need to be installed in a standard location during the construction process. If components are installed relative to a source tree, then the construction process can be completed and the system can be verified before installing it in a production environment. Consequently, this environment will not be affected when the construction process fails. Conceptually, the separation between a construction and a production environment is nice, but it requires that a software system is *relocatable*. This means that after building a software system, it can be moved to another location without rebuilding.

A system without references to components is already relocatable, whereas a system with references is only relocatable if these can be changed after building the system. For instance, a system containing a component with a hard reference to another component (e.g., `/usr/bin/foo`) is not relocatable because when moving the system to another location, the reference is no longer valid. If the component only contains the reference `foo` and uses the `PATH` environment variable to find `foo`, then the system is relocatable. After changing the environment variable the component `foo` can be found.

Relocation is needed because after installation, the location of the components of a software system will have changed. Since a component may be referenced both at build-time and at run-time, references to a component differ at build-time and at run-time. Consequently, when installing a software system, all references to components need to be changed to point to the installation prefix.

Making components relocatable requires special preparation for which no general technique exists. Relocation via the `PATH` variable is only one possible way. Another technique is provided by libtool [40]. This technique makes components using shared libraries relocatable. Other techniques include substitution, where the correct references are substituted in build results, and dynamic querying, where component locations are determined at run-time by querying a composition manager.

If it can be guaranteed that all components of a composition are relocatable, then build results can be stored local to the source tree and later installed in an installation tree. Since this cannot be guaranteed in general, we will always install components relative to an installation tree. This approach requires no extra preparation of components and makes our techniques general usable. The main drawback of our approach is that software building and software installing are no longer separate build actions. Another

drawback is that we cannot easily distinguish between components that are only needed at build-time and components that are used at run-time. All build results get installed regardless of whether they are needed after building or not. Despite the decision for genericity, we also experimented with separate building and installing of relocatable components (see Section 8.3).

To summarize, we will install components relative to a composition and relative to an installation directory tree. The name of the installation prefix will be specific for the composition and may include a version number. The name of the prefix will be determined when configuring the software bundle. Dependency parameters of components in the composition will be bound to this prefix. Hence, if a component 'foo' has a dependency on component 'bar', then the dependency parameter 'bar' will be bound to the variable 'prefix'. This variable will be instantiated at configuration time to hold the installation prefix. Component 'foo' can then reference `libbar.a` from component 'bar' as `$(BAR)/lib/libbar.a`.

4.5. Variability parameter binding

Package definitions may bind configuration parameters of required packages. For instance, in Figure 2, the package CobolSQLTrans binds the parameter 'lang-ext' of the package 'cobol' to 'SQL'. The composition process of build-level components involves collecting all such parameter bindings for all included components. The set of parameter bindings for a package, forms a (partial) configuration of the package and, together with dependency parameter bindings, constitutes the package's 'configuration' section in a bundle definition (see Figure 7).

Since we assume that each component has a standardized configuration interface conforming to Table I(b), the parameter bindings for a component can be passed to the component's configuration process via `--with-` and `--enable-` switches. Bindings of the form `x=y` will be passed to the configuration process as `--with-x=y`. Bindings of the form `x` will be passed as `--enable-x`. Thus, the bindings constitute a set of switches that instruct how the component should be configured.

Variability parameters for which no bindings exist propagate upwards to the bundle's configuration interface and become variability parameters of the bundle (see Section 4.6). If multiple incompatible bindings exist for a variability parameter, then the composition cannot be performed because it is not defined which binding to use. We have plans to experiment with mechanisms to indicate how the configuration process should combine multiple parameter bindings. For instance, to combine multiple bindings in a list of values.

4.6. Bundle variability parameters

The 'configuration interface' section of a software bundle is a collection of unbound variability parameters of all the packages in the bundle. These parameters can be bound at configuration time by the user who is building the bundle.

Observe that name clashes can occur when multiple packages define the same variability parameters and they remain unbound. Name clashes can be prevented by making parameter names unique (for instance by prefixing them with the package name). This can be done by hand or automatically. Another possibility is to simply raise an error when a name clash occurs. It is important to observe though, that name clashes may be intended in case of *cross-cutting* functionality. For instance, debug support is

typically a feature that one wants to toggle for a complete system rather than for each component separately. Therefore, we do not automatically synthesize unique parameter names or raise an error when discovering a name clash. Instead, we allow name clashes to occur to support cross-cutting functionality. We consider it the responsibility of package developers to create unique parameter names when needed. We have plans to extend PDL with parameters for cross-cutting variability. Name clashes are then only allowed for such parameters.

4.7. Bundle dependency parameters

All unbound dependency parameters propagate upwards to bundle dependency parameters. A dependency parameter remains unbound when no package definition could be found that meets all version constraints, or because the package was explicitly excluded from the composition process. In either case, the bundle's configuration process allows a user to specify the location of unbound packages at configuration time. The set of unbound dependency parameters constitutes the 'requires' section of a bundle definition.

4.8. Build process composition

From the information in a bundle definition a build process can be derived. This build process combines the build processes of all constituent components into a single build process for the bundle as a whole. For the composition of build processes we can benefit from decisions and approaches that we discussed earlier:

Standard build interface Each component has a standard build interface consisting of the build actions of Table I(a). Consequently, building component 'foo' is equal to building component 'bar' in the sense that the same build actions are required. Building a software bundle thus consists of successive executing the same build actions for each build-level component.

Package linearization Packages in a bundle definition are ordered such that only back references exist. This ordering equals the required build order because it ensures that components have been built before they are referenced. Thus, the order in which build actions have to be executed is already defined in a bundle definition.

Build=install As explained in Section 4.4, we do not have distinct build and installation steps. Software building therefore implies consecutive executing the build actions 'all' and 'install'.

AUTOMAKE To form a composite build process, a top-level build process needs to be defined that is responsible for executing the build processes of all components. We use AUTOMAKE for generating this build process and we can therefore benefit from its support to recursively execute build processes in subdirectories. Furthermore, we can benefit from its compositional model of making software distributions.

Given a linearized sequence of components 'A', 'B', and 'C', the top-level build process can now be defined with the following AUTOMAKE MAKEFILE:

```
SUBDIRS = A B C
all: install
```

This MAKEFILE ensures that `make install` is issued when `make`, or `make all` is executed. The build process only consists of an iteration over the three directories 'A', 'B', and 'C'. Although this MAKEFILE is slightly simplified, it clearly demonstrates the simplicity of build process composition with build-level components. If software building and installing are separate build actions (see Section 7), then the last line in the MAKEFILE can even be removed.

4.9. Configuration process composition

A bundle definition contains variability and dependency parameter bindings for all bundled packages, as well as top-level variability and dependency parameters. This information is sufficient to compose the configuration processes of each bundled package.

Since each build-level component implements a standardized configuration interface, we can derive proper configuration switches from the configuration section of a package and execute the configuration process of that package. For instance, according to the bundle definition of Figure 7, the configuration process of the build-level component 'cobol' will be executed as:

```
./configure --with-lang-ext=SQL --with-sglr=${prefix}
```

In order to combine all configuration processes of the bundled packages, a top-level configuration process should be defined that executes all individual configuration processes and distributes top-level parameter bindings to packages. With AUTOCONF the definition of such a composite configuration process basically consists of two sections. The first section defines top-level configuration switches, the second defines calls to configuration processes of individual packages. For instance, the first section of the configuration process definition for the bundle definition of Figure 7 can be defined as:

```
AC_ARG_WITH([layout-preserving],
   AS_HELP_STRING([--with-layout-preserving],[Enable.....]))
```

The second section includes a definition of a call to the configuration process of the 'cobol' package. It can be defined as:

```
AB_CONFIG_PKG([cobol],
   [--with-lang-ext=SQL
    --with-sglr=\\\${prefix}])
```

The composite configuration process definition contains such definitions of configuration process invocations for each bundled package.[3]

The configuration script that is generated by AUTOCONF automatically distributes top-level configuration switches. Thus, if the user specifies a value for the `--with-layout-preserving` switch, than it is passed to the configuration processes of the packages.

4.10. Source tree composition

The second and final step in build-level component composition consists of constructing a directory hierarchy containing the source files of all bundled packages, as well as the top-level build and configuration process definitions.

[3] For the configuration of sub packages, we extended AUTOCONF functionality with the macro `AB_CONFIG_PKG` (see Section 5).

The source trees of the packages can be composed by creating a new top-level directory in which subdirectories are created for each constituent component. For instance, if the top-level directory for the bundle definition of Figure 7 is named `CobolSQLTrans-1.0`, then the source tree of the 'cobol' component will be contained in `CobolSQLTrans-1.0/cobol`. The top-level directory will also contain the top-level build and configuration process definitions.

The build interface of a software bundle includes the `dist` build action. This action is intended for building self-contained versioned distributions of software bundles. Such distributions include all build-level components as well as the integrated build process and configuration process. To build a distribution of a bundle one only interacts with the top-level build and configuration interfaces. A distribution of a software bundle thus hides the structuring of a system in build-level components and allows a software developer to distribute his software system as a single unit.

Since package definitions may contain pointers to packages at arbitrary locations via their 'location' fields, it is easy to make cross-institute compositions. This offers a nice opportunity for Open Source software development where software systems are typically developed at different institutes. With build-level component composition these systems can easily be assembled from their individual parts.

However, for bundling in general, we have to face copyright restrictions, which may prohibit arbitrary redistribution of build-level components. Therefore, we propose an alternative approach to self-contained distributions of bundles. In this approach the composition of source trees consists of two steps. In the first step, an "empty" source tree is produced based on the information in a bundle definition. This source tree contains the top-level build and configuration process definitions but not the build-level components (yet). Instead, it contains a tool that knows how to automatically obtain and compose the build-level components. The empty source tree is what is distributed to users. In the second step, the user of the bundle runs the tool to obtain and compose all required build-level components. The result is a self-contained source tree. He can then build the bundle in the usual way by running `configure` and `make`. This way there is no redistribution of software. Based on the specific ingredients of a software bundle, the user can decide whether or not it is allowed to create and distribute a self-contained distribution of this particular bundle. If it is allowed, he can execute the `dist` build action from the bundle's build process to produce a self-contained distribution.

5. Implementation

In this section we discuss the AUTOBUNDLE toolset which offers automated build-level composition according to the last development rule of Table II. AUTOBUNDLE is a collection of tools for producing software bundles, querying package repositories, and for handling requests from online package bases. Figure 11 depicts the architecture of online package bases and shows most of the AUTOBUNDLE tools (as grey ellipses) and their inter-relations. Below we will discuss this architecture in more detail.

Build-level component composition has been implemented as a program transformation system operating on abstract syntax trees (ASTs). These ASTs are obtained by parsing the package definition files in package repositories. The composition process consists of first transforming the ASTs of a collection of package definitions to an AST of a bundle definition, and then transforming the latter to a composite configuration and build process. The AUTOBUNDLE toolset also contains tools for analyzing ASTs of package definitions and for transforming them to HTML. We used Stratego for implementing

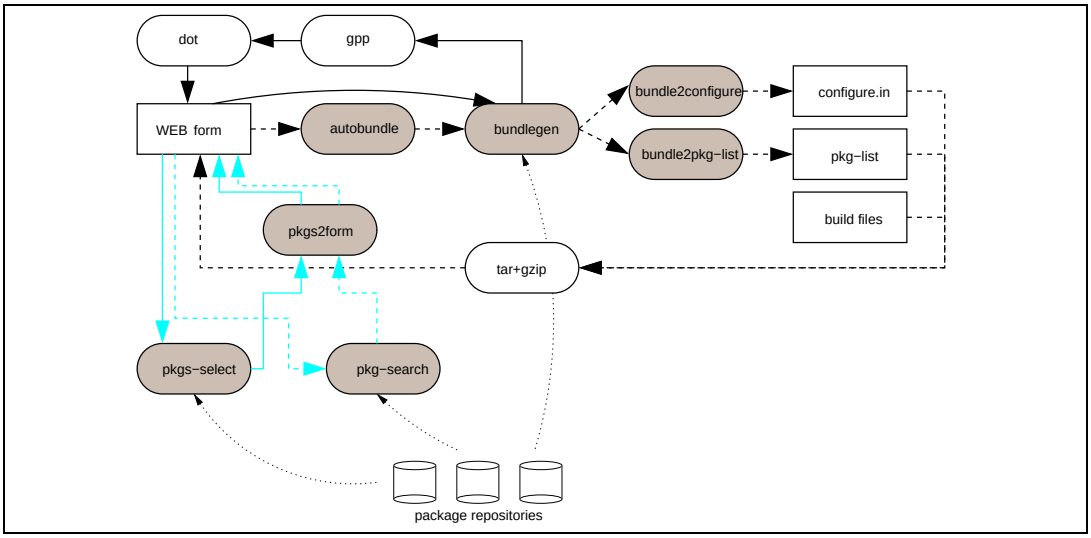

Figure 11. Architecture of the online package base.

the AUTOBUNDLE toolset. Stratego [55] is a programming language designed especially for developing such program transformation systems.

Figure 11 shows that the AUTOBUNDLE toolset consists of several components that can be connected to each other to obtain the desired functionality. Below we will discuss the most important tools of AUTOBUNDLE. The white ellipses in Figure 11 correspond to third-party tools: DOT is a graph visualization tool [22, 23], GPP is a generic pretty-printer [31], TAR a Unix archiving utility, and GZIP a compression utility. These will not be further discussed.

BUNDLEGEN This tool implements package normalization, package linearization, dependency parameter binding, and variability parameter binding as discussed in Section 4. BUNDLEGEN serves two purposes: i) synthesizing a bundle definition from a selection of package definitions; ii) generating package dependency graphs. The result of BUNDLEGEN is an AST that either represents a bundle definition (such as depicted in Figure 7), or a package dependency graph in the DOT graph format that can be further processed (see the black arrows in Figure 11) to yield a picture like Figure 10.

BUNDLE2CONFIGURE This tool implements configuration process composition as discussed in Section 4.9. The tool reads a bundle definition as input and produces an AUTOCONF configure script.

To support proper configuration of sub-components, we slightly extended the functionality of AUTOCONF by developing the macro `AB_CONFIG_PKG`. This macro calls the configurations script of a sub-component and passes it component-specific configuration switches. These switches correspond to bound dependency and variability parameters in the bundle definition (see Section 4.9 for an example).

BUNDLE2PKGLIST In Section 4.10 we motivated a two phase source tree composition process. In the first phase an "empty" source tree is produced that knows how to obtain and compose its sub-components. Expansion of this source tree by obtaining and composing sub-components is performed in the second phase. The information about sub-components is stored in a file called `pkg-list`. This file is a flat list containing a triple for each component, describing the component's name, its version, and its download location. BUNDLE2PKGLIST creates the `pkg-list` file by collecting this information from a bundle definition. The file is processed by the `collect` tool that performs the actual composition of build-level components (see below).

AUTOBUNDLE This tool performs the complete build-level component composition process. It can be used as end-user tool or as service on a package-base server (as depicted in Figure 11). It combines BUNDLEGEN, BUNDLE2CONFIGURE, and BUNDLE2PKGLIST. First, it uses BUNDLEGEN to synthesize a bundle definition. Then, it executes BUNDLE2CONFIGURE and BUNDLE2PKGLIST to construct an AUTOCONF configure script and a `pkg-list` file from the bundle definition. Finally, it creates a bundle by archiving the generated files together with a generic build process definition and some additional files that are required by the Autotools build system. This process is depicted in Figure 11 by the black dashed arrows. Archiving is performed by the tool combination TAR and GZIP.

The generic (i.e., composition-independent) AUTOMAKE MAKEFILE was already discussed in Section 4.8. In addition to the simple build process definition presented there, the actual MAKEFILE used by AUTOBUNDLE offers functionality for excluding components from the build process and for automatically installing documentation and copyright files.

The use of AUTOBUNDLE is fairly simple. One has to specify the top-level packages that need to be bundled, and (optionally) a list of package repositories where AUTOBUNDLE looks for package definitions. If no package repositories are specified, AUTOBUNDLE accesses package repositories specified in the `AUTOBUNDLE_SEARCH_PATH` environment variable. For example, if a package repository is located at `/opt/packages`, then by issuing the command:

```
autobundle -I /opt/packages -p CobolSQLTrans-1.0 -o .
```

A file called `CobolSQLTrans-bundle-1.0.tar.gz` is created in the current directory that contains the "empty" bundle for the build-level composition of the `CobolSQLTrans-1.0` component. The name and version of a bundle are automatically synthesized from the composition of packages. They can also be specified explicitely by a user with the '-n' and the '-v' command line switches. By default, AUTOBUNDLE searches for package definitions in local package repositories. Additionally, AUTOBUNDLE can serve as command-line front-end for a remote online-package base. In that case, a bundle request is constructed from the command-line switches passed to AUTOBUNDLE and sent to an online package base.

Empty software bundles contain the COLLECT.SH tool that is able to retrieve and integrate the build-level components of a bundle. For example, to unpack and obtain all sub-components of the `CobolSQLTrans` bundle, one can issue the following commands:

```
tar xfz CobolSQLTrans-bundle-1.0.tar.gz
cd CobolSQLTrans-bundle-1.0
./collect.sh
```

The tool currently supports copying ordinary files, and downloading via HTML, FTP, CVS, and SUBVERSION. Downloading via CVS and SUBVERSION is further addressed in Section 6.

PKGS2FORM The HTML forms of online package bases (see Figure 5) are created with PKGS2FORM. This tool reads a collection of package definition files and produces an HTML form, from which packages can be selected and bundled, package and bundle dependency graphs can be generated, packages can be searched, and package selections can be made (see the outgoing arrows of the HTML form node in Figure 11). The tool accepts a large number of command-line options to control the generated HTML output.

PKG-SEARCH This tool returns a list of package names from a given set of package definitions that match a given set of keywords. The output can be passed to PKGS2FORM to generate an HTML form from the matching packages (see the dashed grey arrows in Figure 11). This is how the search functionality of online package bases is implemented.

PKGS-SELECT This tool returns a list of package names by searching repositories for packages with a certain name. It serves to restrict the search space of AUTOBUNDLE tools to a subset of the packages contained in a package repository. The tool PKGS2FORM can be used to produce a (restricted) online package base from a package selection (see the grey arrows in Figure 11).

Online package bases An online package base consists of an HTML form and CGI stubs that call the aforementioned tools from the AUTOBUNDLE toolset. The CGI stubs are activated from the HTML form which is created by transforming the package definitions in a package repository to HTML using PKGS2FORM. For example, when pressing the 'bundle' button on the HTML form the activated CGI stub executes the AUTOBUNDLE tool to create the desired software bundle.

An online package base is initiated with the MKOPB tool. This tool installs the required CGI stubs, a generic MAKEFILE for creating the HTML form, and a configuration file. The configuration file serves, amongst others, to specify package repository locations and the WEB address of the online package base. After running the MKOPB tool and editing the generated configuration file, the HTML form can be created by executing MAKE. The only thing that remains to be done is to activate the online package base by making it accessible via a WEB server.

6. Integrating package development and package Deployment

Thanks to automated build-level composition, composite software systems form a unit at deployment time: they can be packaged, transfered to a target machine, and installed as a single unit. In this section we describe how build-level compositions can also be handled as a unit at *development* time. This improves development practice of component-based software systems. The reason is that the development—integration—test cycle is shortened, crosscutting development is permitted, system-wide development (within a single development environment) is allowed, and, in case of simultaneous development of a component in multiple systems, inconsistencies can be signaled soon.

6.1. Package development cycle

Recall from Section 3, that a build-level package is a unit of deployment. Packages can be developed, maintained, and released individually. They are independent of a particular Software Configuration

Management (SCM) system, and can be developed by different groups at different institutes. The development cycle of an individually developed package can be characterized as follows:

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

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

5. Select the packages that form a target software system.
6. Bundle the selected packages and those that are required by them with AUTOBUNDLE.
7. Unpack the generated software bundle.
8. Collect the ingredients of the bundle using COLLECT.SH.
9. Configure it using the generated configuration process.
10. Build the bundle using the generated build process.
11. Test the application to see whether the changed package functions as intended.

Steps 1–4 have to be performed for all packages under development. Steps 5–11 have to be executed for integration testing and have to be repeated whenever one or more packages have changed.

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

6.2. Package-based software development

The previous section explained that the package development cycle is complex. Even with the help of automated build-level composition the development cycle is not very practical. What's more, one could argue that integration testing should not be performed *after* package release, as we suggested in Section 6.1, but *before*. This suggests that the presented development cycle is not only complex, but also has serious deficiencies.

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

Integration of package development cycles should fulfill the following requirements:

- Integration should preserve component structure. That is, although less visible for a developer, the structure in build-level components remains existent.
- Integration of multiple SCM systems should be possible.
- Switching between development and deployment of packages should be easy (see below).
- Collaborative software development (by multiple groups or institutes) should be supported.

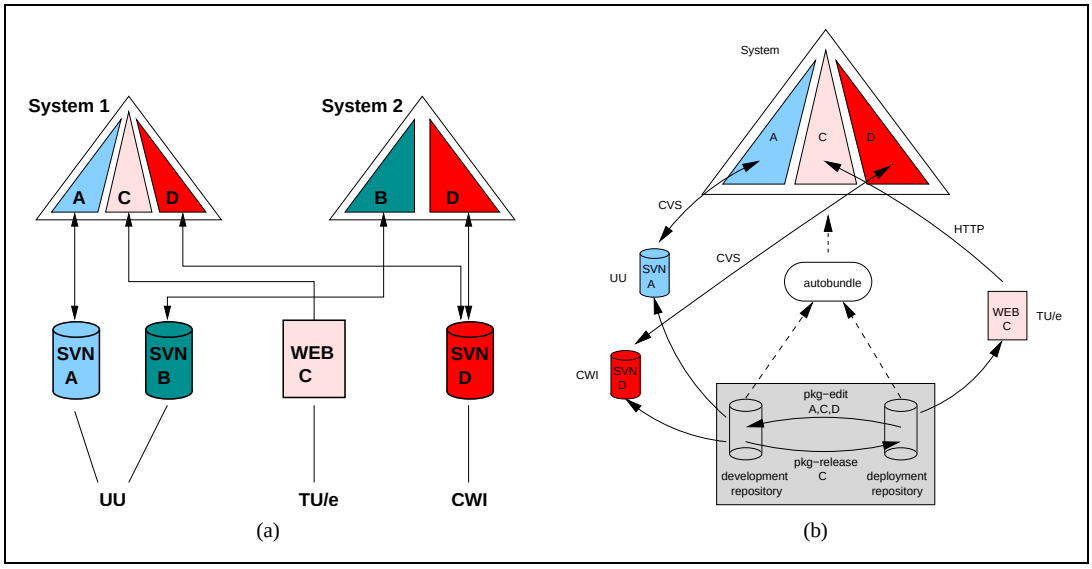

Figure 12. (a) An example of package-based software development in the context of two distinct software systems. (b) Architecture for integrated package development with AUTOBUNDLE.

- Simultaneous development of build-level components in multiple software systems.

Figure 12(a) shows an example of integrated package development. This figure shows two systems composed of 4 different build-level components (*A*–*D*). The components *A*, *B*, and *D* are obtained from an SCM system (SUBVERSION in this example). They are being developed in the context of system 1 and system 2. Component *D* is developed simultaneously in the context of both systems. System 1 also contains component *C* in the form of a released package that is obtained from a WEB-site. The components are developed at three different institutes (UU, TU/e, and CWI). The UU components are developed by two different groups, each having a separate SCM system.

The key benefits of package-based software development can be summarized as:

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

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

Package-based software development thus combines the advantages of having a system consist of components (reuse) with the advantages of having a system consist of only one piece of code (integrated development). In the next section we will discuss how the AUTOBUNDLE toolset can be used to support package-based software development.

6.3. Implementing package-based software development

To support package-based software development, we make a small extension to the package definition language (PDL) of Figure 3 and benefit from several features of build-level composition and AUTOBUNDLE. We developed some extra tools for switching between development and deployment mode. These features, the modified package definition language, and the additional tools are discussed below. Figure 12(b) shows the architecture for package-based software development where AUTOBUNDLE is used to integrate package development cycles.

Multiple download protocols In order to combine component deployment (by downloading packages from Inter/Intranet), and component development (by obtaining the sources of a package from an SCM system), we need to support different download protocols. The standard download protocol used by AUTOBUNDLE, is the HTTP protocol. It is intended for component deployment and can be specified in the location field of a package definition. In addition to the HTTP protocol AUTOBUNDLE also supports SUBVERSION [15] (or SVN for short) and CVS [11]. These are used for component development by obtaining sources files from SUBVERSION or CVS repositories. E.g.,

```
location=svn://https://svn.cs.uu.nl:12443/repos
```

The COLLECT.SH tool (see Section 5), which is used to obtain and integrate the sources of a software bundle, interprets these entries and, depending on the protocol, downloads a package from the WEB or performs a ‘checkout’ of a head revision of an SCM system.

Multiple download locations AUTOBUNDLE needs to know the location of a package’s SCM system (for development) and its download location (for deployment). In order to be able to switch between component development and deployment, both download locations have to be captured in a package definition. To that end, we slightly extended the syntax of PDL and introduced the notion of ‘primary’ and ‘secondary’ package locations. E.g.,

```
location
```

```
primary=http://www.cs.uu.nl/package-base
secondary=svn://https://svn.cs.uu.nl:12443/repos
```

If a component is used (deployed), the primary location field points to the HTTP location, while the secondary location field points to the SCM system. If a component is being developed, both fields are swapped: the primary field now points to the SCM system, the secondary points to the HTTP location. AUTOBUNDLE only considers the 'primary' field to determine the location of a package and ignores the 'secondary' field. Additional tools can swap the values of the fields to switch from development mode to deployment mode and *vice versa*.

Package search path As we already pointed out in Section 5, AUTOBUNDLE can use multiple package repositories. In order to switch between component deployment and component development, two package repositories are used (see Figure 12(b)).The first package repository contains ordinary package definitions for component deployment. We call it the *deployment (package) repository*. [4] The second repository has highest priority and is used to store package definitions for packages under development. It is called the *development (package) repository*.

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

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

Tool integration We have shown how the techniques presented earlier in this article can be used to integrate the development cycle of multiple packages. AUTOBUNDLE is used to merge source trees contained in source packages. Two package repositories are used to store package definitions for deployed packages as well as for packages under development. The COLLECT.SH tool is used to obtain a component from Inter/Intranet or from an SCM system. Finally, SUBVERSION or CVS is used by COLLECT.SH to obtain packages from SCM systems. With these tools, a developer can switch between development and deployment mode of components. To make switching more easy, we integrated these tools in two additional tools. The first tool is PKG-EDIT and turns a number of packages into development mode. The second tool is PKG-RELEASE and turns packages into deployment mode.

The PKG-EDIT tool puts a set of packages into development mode by obtaining a head revision of the packages from the corresponding SCM system. To that end, PKG-EDIT performs the following tasks:

1. It copies the package definitions to the development package repository.

[4] More deployment repositories can be used if desired (see Section 3.3), but in this section we only consider one.

2. It swaps the values of the two location fields in each package definition.
3. It removes the packages from the composite source tree.
4. It uses AUTOBUNDLE to obtain a new bundle based on the development packages.
5. It uses COLLECT.SH to obtain the sources of the packages that are now under development from the corresponding SUBVERSION or CVS repositories.

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

The PKG-RELEASE tool performs the opposite as PKG-EDIT, i.e., it switches a set of packages from development mode into deployment mode. It performs the following actions:

1. If not all changes to a package were committed to the SCM system, it makes a copy of the checked-out version, and checks a fresh version out. This ensures that only committed changes go into a new release, but also that uncommitted changes will not be lost.
2. It extracts a version number from the configuration script and checks that this version does not already exist for the package in the deployment repository.
3. It makes a new distribution of the package.
4. It releases the package by uploading it to a package repository.
5. It moves the package definition from the development to the deployment repository.
6. It uses AUTOBUNDLE to obtain a new bundle based on the deployment packages.
7. It uses COLLECT.SH to obtain the sources of the packages from their HTTP locations.

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

6.4. Package development: an example

Figure 12(b) depicts integrated package development with AUTOBUNDLE, the deployment and development repositories, and the tools PKG-EDIT and PKG-RELEASE. Below we describe the development cycles that are depicted in the figure.

We start by assembling the initial system with AUTOBUNDLE (System 1 in Figure 12(a)):

```
> autobundle -p A-1 -I /opt/package-base
```

This command creates a software bundle from the package *A-1* and the packages *C-1* and *D-1*, which are required by *A-1*. Here we use `/opt/package-base` as deployment repository. The tools PKG-EDIT and PKG-RELEASE automatically synthesize a development repository to use. Initially, this repository is empty. Hence, the bundle only contains deployed packages. The next step is to obtain and integrate the sources of these packages:

```
> ./collect.sh
Obtaining a distribution of "A" via HTTP
Obtaining a distribution of "C" via HTTP
Obtaining a distribution of "D" via HTTP
```

This yields a composite source tree and an integrated build environment for the three packages. Configuring and building can be done by running `./configure` and `make`, respectively.

Now suppose that we need to make a crosscutting change in all three packages. To that end, we use the PKG-EDIT tool to turn these packages into development mode:

```
> pkg-edit A C D
Obtaining A from Subversion
Obtaining C from Subversion
Obtaining D from Subversion
```

As a result of this command, the packages are now obtained from SUBVERSION. For instance, if the package definition for *A* contains `svn://https://svn.cs.uu.nl:12443/repos` as location field, then the source code of *A* is obtained by running the command:

```
svn checkout https://svn.cs.uu.nl:12443/repos/A
```

We can now start developing by editing the source code anywhere in the composite source tree. All SUBVERSION commands can be used in the root of the source tree, or in any subdirectory in order to commit changes, incorporate changes made by others, undo changes etc. For instance, suppose that file `f.c` in the component *C* is modified by another developer. We can bring our composite source tree up-to-date by issuing the command:

```
> svn update A C D
At revision 4929.
U C/f.c
Updated to revision 1113.
At revision 3876.
```

This example shows that a developer can manage the source tree as a unit, while in fact he is operating on components hosted in (possibly) three distinct SUBVERSION repositories.[5] All SUBVERSION commands work by accessing the specific SUBVERSION repository that hosts a particular component. Hence, the structure in components remains in tact, although developers can make changes easily anywhere across component boundaries. Furthermore, since we have all source at our disposal, development tools can be used for system-wide development. For instance, an integrated development environment can be used for doing the actual development, whereas integration tests and code analysis can be performed on the system as a whole.

After the changes to a component have been made and have been committed to the SUBVERSION repository, the corresponding package can be released with the PKG-RELEASE command. For instance, to release the package *C* we can issue the command:

```
> pkg-release C
Obtaining a distribution of "C" via HTTP
```

This command will check whether all changes made to *C* have been committed. If not, a copy of the checked-out version is saved and a fresh version is obtained from the SUBVERSION repository. This is, as we explained, because a release should only contain committed changes, but uncommitted modifications must not be thrown away. As pointed out earlier, the PKG-RELEASE command then makes a new release of the package and replaces the checked-out version by the new created release. This yields the situation as depicted in Figure 12(b), where *A* and *D* are being developed, while *C* is being deployed.

[5] In contrast to [49] we do not provide an abstraction layer for different SCM systems because it may restrict SCM functionality too much. As a consequence, a developer must know which SCM systems hosts the components that he has under development.

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

7. Case studies

Most techniques presented in this article were developed in the context of the ASF+SDF META-ENVIRONMENT (ASME) [4] and STRATEGOXT [36]. Both systems are intended for language-centered software engineering [34], a software engineering discipline for the development of programming language supporting software. The ASME is an integrated environment for developing programming languages and tools. STRATEGOXT is a tool bundle for developing program transformation systems. Typical components that are reusable in this domain are: parsers, parser generators, compilers, editors, debuggers, and pretty-printers.

By design, the ASME is a component-based system. Unfortunately, the component structure was initially not visible at the build-level because CBSE principles were not effectively used. This led to entangled components, which were, although intended differently, in practice not externally reusable. Furthermore, the system used some additional software packages, including ATerms [5] for data representation and exchange, and ToolBus [2] as coordination architecture. These packages were developed and released separately, which complicated the deployment the ASME. On the one hand, deploying everything separately complicated the installation effort for our customers too much. On the other hand, bundling all packages together yielded fragile distributions (often leading to installation problems) because the packages were not truly compositional.

The need to reuse parts of the ASME and the need to deliver robust, easy-to-install distributions, formed the motivation to split its implementation and to develop the AUTOBUNDLE toolset. Nowadays, the system consists of 19 build-level components which are available in multiple versions. Several of them are being used in different contexts by different groups and institutes around the world. The construction process of distributions is fully controlled and automated by AUTOBUNDLE.

A research group at INRIA, France, was developing another language development environment, called ELAN [3]. They too had difficulties in building robust distributions. Furthermore, they wanted to reuse some of the technology from the ASME. This was difficult because the build process of ELAN was not compositional. Combining both systems had to be performed manually and repeated once new versions of these components were released. To simplify this, ELAN was also split-up into build-level components. Consequently, ELAN can now be bundled with components from the ASME and ELAN releases are generated by AUTOBUNDLE. More recently, the integration of both systems has further been improved such that ELAN is now based on the open architecture of the ASME. This resulted in the ELAN+SDF META-ENVIRONMENT [7, 6]. Of course, in addition to the build-level, also work had to be done at the functional level in order to be able to compose both systems. ELAN therefore had to be adapted to fit in the language-centered software engineering framework. As a result of splitting-up ELAN, its own functionality became available for reuse as well. For instance, the choice point library [43] is being used by STRATEGOXT.

STRATEGOXT formed another strong motivation for splitting-up the ASME. The development of STRATEGOXT was initiated because the components of the ASME were so tightly integrated: on the one hand, the generic language technology based on SDF [26, 54] (for syntax definition and parsing) was coupled with the programming language ASF [1]; on the other hand, the components were tightly integrated in a graphical user interface. At that time a need existed to combine the SDF part with other programming languages and to have more control over the exact composition of components.

After the functionality of the ASME was made available as build-level components, the goals of STRATEGOXT could be achieved. The result is that the ASME and STRATEGOXT have the same core components. STRATEGOXT consists of 28 build-level packages, from which 8 packages are reused from the ASME and one from ELAN. Extensions to STRATEGOXT (e.g., transformation tools for different programming languages) are developed on top of the core STRATEGOXT bundle in the form of separate build-level components. The result is a large collection of components that can be used for the construction of a wide variety of program transformation systems [51].

7.1. The Online Package Base

The Online Package Base (OPB) is a collection of Open Source software packages developed at several research institutes [46]. The OPB is globally accessible and extendible. That is, anyone can obtain software bundles or contribute new software packages. Figure 5 gives an impression of the OPB.

The main motivation for initiating the OPB was to increase the accessibility of our software. The way that our software was provided used to be inconsistent and unclear. For instance, we used different FTP and WEB servers, our software packages were offered via project, as well as personal WEB pages (each having a different look and feel), and the WEB pages were difficult to maintain and often out of date. Consequently, if someone was interested in our software, it was difficult to collect all corresponding software packages. It would be a shame if inaccessibility would be the reason for not using our software. Surprisingly, many institutes still suffer from accessibility problems of their software. The OPB strives to provide a consistent and up-to-date interface for retrieving software. Other motivations for initiating the OPB where to make our software easy to install and to be able to easily share software with other groups and institutes.

Currently, the OPB contains package definitions for 72 different packages. In total, these are available in 389 versions and originate from 10 different institutes. Anyone can contribute to the OPB by filling out a package contribution form. Since the OPB does not collect software packages themselves but only package definitions, one preserves complete control over the software package, after making it available at the OPB. Via the pointer in a package definition (see Section 3.2), the software can be retrieved.

7.2. Decoupling Graphviz

In [35] we describe a project that was concerned with extracting build-level components from existing software systems. We defined a semi-automatic restructuring process that applies the development rules of Table II to a source tree in order to decouple it into build-level components. This process consists of three phases. In the first phase a source tree analysis is performed to detect candidate components and component references. In the second phase, the candidates are extracted and turned into build-level

components. In the third phase the components are turned into packages and an online package base is generated to make them available for reuse.

We applied this process to the source tree of Graphviz, an Open Source collection of tools for manipulating graph structures and generating graph layouts developed by AT&T [22, 23]. For this article we repeated the migration process discussed in [35] for Graphviz version 1.16. This versions contains 136 directories, 1,615 files, and 154 directory references. It is implemented in several languages, including C, C++, Tcl/Tk, AWK, and shell scripts. The implementation consists of more than 300,000 lines of code. The restructuring yielded 46 build-level components. From these components we reproduced the original Graphviz distribution. More interestingly, we combined the Graphviz package base with additional package bases to make compositions of Graphviz components and arbitrary other build-level components. This demonstrated build-level CBSE in practice.

8. Widening the scope of build-level components

8.1. Autotools-less Components

In this article we made use of the GNU Autotools to control the build and configuration process of build-level components. However, build-level CBSE only depends on standardized build and configuration interfaces, as we pointed out in Section 2.4. It is therefore possible to weaken the dependency on Autotools and to combine build-level components which have a non-Autotools implementation of build and configuration interfaces.

To that end, a build process has to be defined in a MAKEFILE (or in a tool MAKE if the dependency on standard MAKE should also be dropped), which implements the build interface of Table I(a). Furthermore, a configuration process should be defined in a tool CONFIGURE. This tool should understand the same syntax as depicted in Table I(b) and have a configuration switch for each build-level variability and dependency parameter.

8.2. Combining Source and Binary Components

Thus far, we only considered build-level components in source form. However, we can widen the scope of build-level CBSE to binary (i.e., pre-compiled) components. Binary components can be considered as an optimization of components in source form [17]. That is, most of the build actions are already performed for such components. Like ordinary build-level components, a binary component can be used for build-level CBSE if it contains a build process that implements the build interface of Table I(a), and a configuration process implementing the configuration interface of Table I(b). The build process of a binary component therefore typically implements the build actions `install`, `uninstall`, and `dist`, whereas the remaining build actions of Table I(a) become empty actions.

8.3. Separating software construction and installation

In Section 4.4 we motivated why build-level components are installed *before* they are used. The main motivation for this decision was to make our approach general usable. The main drawback of this approach is that software construction and software installation no longer are separate activities. That

is, the construction process consists of sequentially building *and* installing each constituent build-level component. Although this is common practice and generally accepted when installing the components of an application manually, it is a general complain about software bundles. Below we discuss when and how software construction and installation of build-level components can be separated.

The difficulty is that the location of software artefacts changes during installation. Since installed software may resist on a different file system than the corresponding source tree, and because the source tree may be removed after installation, references into a source tree may break. Consequently, references to build-level components which are valid at build-time become invalid after installation. Referencing a component can be build-time only (e.g., linking a static library), run-time only (e.g., executing a program at run-time), or both (i.e., linking and loading a shared library). Consequently, component dependencies are timed. For a build-time only dependency, there is no need to install the corresponding component because it will not be used outside the build process. A run-time dependency is not referenced at build-time and there is thus no need to install the corresponding component during the build process. So, the only situation that causes problems is when a component is referenced at build and at run time. The tool LIBTOOL, which is part of GNU Autotools, solves this for shared libraries. It provides an ingenious mechanism for making shared libraries relocatable. Build-time references to shared libraries within the same source tree are relocated at installation time to a common installation prefix. This way, references to shared libraries remain valid after installation.

If the components of a software bundle have either run-time or build-time dependencies, or if components with such dependencies are relocatable (e.g., by using LIBTOOL for shared libraries), then there is no longer a need to merge software construction and installation. To support separation of software construction and installation, we made the component binding mechanism of AUTOBUNDLE adaptable. We have used this new feature successfully for Graphviz (see Section 7.2). The only build-time/run-time dependencies of Graphviz are due to dynamic libraries, but thanks to LIBTOOL these are relocatable. We generated a composite build process with AUTOBUNDLE with only local component references. As a result, we were able to separate the construction and the installation process.

8.4. Implementing Software Product Families

In Section 3.2, we introduced the notion of concrete and abstract package definitions. In this article, concrete packages serve as abstraction for build-level packages, whereas abstract packages serve as abstraction for groups of packages.

Another use of abstract packages is to map domain features [37] to an implementation. A domain feature is then represented as an abstract package. A collection of abstract packages forms the feature space of a family of systems. The transitive closure of the dependencies of an abstract package forms the implementation of a feature. An online package base serves as a feature base from which individual products can be identified and automatically assembled. The underlying package repositories can be shared by different product families to allow effective software reuse across product families.

We performed two studies about product family development with build-level components. The first involved turning the Linux kernel into a product line architecture [30]. We proposed an architecture where dedicated kernel source trees are assembled from feature selections at an online package base. The second project involved the implementation of a commercial documentation generator as a product line driven by AUTOBUNDLE [16].

9. Related work

The work presented in this article has several similarities with the component model KOALA [45, 44]. The KOALA model has a component description language like our package definition language. Implementations and component descriptions are stored in central repositories accessible via Internet. KOALA packages are independently developed by development teams using distinct SCM systems. 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 deployment of compositions in the form of bundles is not addressed. The model does neither support composition of SCM systems, nor combined development of package compositions. Because it is tailored towards a single programming language, KOALA has more control over the composition process at the price of less genericity. For example, adopting non-KOALA components is difficult. We are in the process of merging concepts of source tree composition with the KOALA component model.

In [50], the package definition language (PDL) is adapted to formalize the notion of variability. To that end, the configuration interface section of PDL is replaced by a feature model, defining the variation points of a component. Furthermore, some support for conditional package dependencies is provided. This allows that the existence of a package dependency may depend on a feature binding.

In [27, 28], a software *release management* (SRM) 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. SRM is concerned with making software available to users, not with the development of composite systems or component integration, as is the case in this article. SRM can be combined with software deployment. For example, in [28] the SOFTWAREDOCK [25] is advocated to deploy released software systems.

This article addresses techniques to assemble software systems from build-level 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 [53, 18], even when partly automated by package managers (like RPM [20]). Although the techniques presented in this article simplify software construction, they do not make software deployment technology [10] superfluous. For instance, we strongly advocate the NIX deployment system for managing the installation of composite software systems. NIX is a recently developed software deployment system that ensures safe and complete installation of packages [18].

SCM support for component-based software development is addressed in [41, 57]. They analyze requirements of configuration management in a component-based software development process. They focus on single SCM systems, and composition of SCM systems is therefore not addressed. Also, managing reuse across project, group, and institute boundaries is not discussed. A promising SCM system supporting crosscutting development of components and collaborative software development across project and institute boundaries is STELLATION [12, 13]. Combining software deployment and software development is discussed in [49]. Their approach is restricted to JAVA components and there is no shared SCM system for cross component development, as we disccused in Section 6. Instead, deployed JAVA components are imported in distinct SCM systems.

Many articles, e.g., [9, 8, 14, 17] address solving limitations of traditional `make` [19], such as improving dependency resolution, build performance, and support for variant builds. They do not discuss composition of source trees and build processes. Gunter [24] 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. It is sometimes argued that build knowledge should not be spread across directories at all but contained in a single MAKEFILE [42]. The motivation is that only in a single MAKEFILE completeness of build dependencies can be achieved. However, this is merely due to limitations of traditional `make` implementations and completely ignores the importance of weak coupling for component composition [56]. Consequently, unless generated, build-level CBSE is hampered because components are entangled in global MAKEFILES.

10. Concluding remarks

This article adresses software reuse from a build-level perspective. We observed that modularization principles are not applied at the build-level, leading to strongly coupled build-level artefacts with weak cohesion. Consequently, files with potential reusable functionality are often entangled in source trees and their build instructions hidden in monolithic configuration and build process definitions. The effort of isolating files for reuse in other software systems usually does not outweigh the benefits of reusing the module. As a result, reuse is sub-optimal or too coarse-grained.

To improve this situation, we proposed to apply component-based software engineering (CBSE) principles to the build level. To that end, we discussed how build-level artefacts can be turned into build-level components, and how all component access can be directed via well-defined interfaces. This improves reusability because build-level artefacts become independently deployable and candidates for third-party composition. Automated composition techniques assist the construction of software systems from such build-level components.

Contributions This article forms a comprehensive overview of build-level CBSE and combines and extends parts that have been published before [32, 33, 35]. The article explores component development, component deployment, (automated) composition, and software engineering with build-level components. In addition, a number of case studies, as well as some generalizations of the presented techniques are discussed. The main contributions of this article are:

- The concept of build-level components having build, configuration, and requires interfaces. We explained how build-level CBSE is promoted by the GNU Autotools.
- Definition of 9 rules for developing “good” build-level components. By applying these rules, build-level artefacts can be turned into build-level components.
- Techniques for deploying build-level components, including the package definition language (PDL) and the notion of online package bases.
- Techniques for build-level component composition and the development of the AUTOBUNDLE toolset for automating the composition process.
- Development of techniques to combine component development and component deployment. This reduces the complexity of the software development process with build-level components.
- Case studies to demonstrate build-level CBSE. These show how build-level CBSE improves software reuse and demonstrate the feasibility of automated migration to build-level components.

Discussion The build and configuration interfaces that we used in this article are based on the GNU Autotools. Although different interfaces are possible, we think that by using Autotools the adoption of our techniques is simplified because so many software systems today are driven by Autotools. However, in Section 8 we explained how to build components without Autotools. Furthermore, the concepts of build-level CBSE are still applicable, even with totally different interfaces.

In this article we strived towards genericity and did not make any assumption about build-level components. This limits the level of control that we have on component composition. We cannot, for instance, separate component construction and installation (see Section 4.4). We have plans to further explore the concept of build-level CBSE with special-tailored components. We expect that compositions can be improved with such components. Directions for optimization include build processes, configuration processes, and source modules. As an example of the latter, we would be able to apply source code optimization techniques, crossing build-level component boundaries, if all components were implemented in a particular programming language.

In [35] we demonstrated how the techniques presented in this article can be applied semi-automatically to existsing software systems.

An application domain for build-level CBSE that we did not address is software testing. Build-level composition techniques can assist the test process of individual software components and component compositions. For instance, the components in a package repository can be tested by (randomly) selecting subsets of components. These sets of components can be automatically assembled, built, and tested via the `check` action of the build interface. By selecting component collections randomly, we can test components thoroughly in different compositions, within a fixed amount of time.

Availability The AUTOBUNDLE toolset is distributed as free software and is available at `http://www.cs.uu.nl/~mdejonge/software`. The online package base can be reached from `http://www.program-transformation.org/package-base`.

Acknowledgments We would like to thank Eelco Dolstra and Martin Bravenboer for feedback on drafts of this paper.

REFERENCES

1. J. A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989.
2. J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 75–88. Springer-Verlag, 1996.
3. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proceedings of the International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, September 1998.
4. M. G. J. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. 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 *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, April 2001.
5. M. G. J. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

6. M. G. J. van den Brand, P.-E. Moreau, and C. Ringeissen. The ELAN environment: a rewriting logic environment based on ASF+SDF technology. In M. G. J van den Brand and R. Lämmel, editors, *Proceedings: 2nd International Workshop on Language Descriptions, Tools and Applications*, volume 65. Electronic Notes in Theoretical Computer Science, April 2002.
7. M. G. J. van den Brand, P.-E. Moreau, and J.J. Vinju. Environments for Term Rewriting Engines for Free! In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *LNCS*, pages 424–435. Springer-Verlag, 2003.
8. 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*, volume 1005 of *Lecture Notes in Computer Science*, pages 81–87. Springer-Verlag, October 1995.
9. J. Buffenbarger and K. Gruel. A language for software subsystem composition. In *34th Annual Hawaii International Conference on System Sciences (HICSS-34)*. IEEE Computer Society Press, 2001.
10. A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado, April 1998.
11. P. Cederqvist et al. *Version Management with CVS*, 2002. Available at `http://www.cvshome.org`.
12. M. C. Chu-Carroll. Supporting distributed collaboration through multidimensional software configuration management. In *Tenth International Workshop on Software Configuration Management (SCM-10)*, 2001.
13. M. C. Chu-Carroll, J. Wright, and D. Shields. Supporting aggregation in fine grained software configuration management. In *Proceedings: Tenth Symposium on Foundations of Software Engineering*, pages 99–108. ACM Press, 2002.
14. G. M. Clemm. The Odin system. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, volume 1005 of *Lecture Notes in Computer Science*, pages 241–2262. Springer-Verlag, October 1995.
15. B. Collins-Sussman, B. Fitzpatrick, and M. Pilato. Subversion: The definitive guide, 2003. Available at `http://subversion.tigris.org`.
16. A. van Deursen, M. de Jonge, and T. Kuipers. Feature-based product line instantiation using source-level packages. In G. J. Chastek, editor, *Proceedings: Second Software Product Line Conference (SPLC2)*, number 2379 in LNCS, pages 217–234. Springer-Verlag, August 2002.
17. E. Dolstra. Integrating software construction and software deployment. In *11th International Workshop on Software Configuration Management (SCM-11)*, volume 2649 of *Lecture Notes in Computer Science*, pages 102–117. Springer-Verlag, May 2003.
18. E. Dolstra, E. Visser, and M. de Jonge. Imposing a memory management discipline on software deployment. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 583–603. IEEE Computer Society Press, May 2004.
19. S. I. Feldman. Make – A program for maintaining computer programs. *Software – Practice and Experience*, 9(3):255–265, March 1979.
20. E. Foster-Johnson. *Red Hat RPM Guide*. John Wiley and Sons, 2003.
21. Free Software Foundation. *GNU Coding Standards*, 2004. `http://www.gnu.org/prep/standards.html`.
22. E. Gansner and S. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, September 2000.
23. Graphviz. `http://www.graphviz.org`.
24. C. A. Gunter. Abstracting dependencies between software configuration items. *ACM Transactions on Software Engineering and Methodology*, 9(1):94–131, January 2000.
25. R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf. An architecture for post-development configuration management in a wide-area network. In *17th International Conference on Distributed Computing Systems*, pages 269–278, May 1997.
26. J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF —reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.
27. 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 *Lecture Notes in Computer Science*, pages 159–175. Springer-Verlag, 1997.
28. A. van der Hoek and A. Wolf. Software release management for component-based software. *Software – Practice and Experience*, 33(1):77–98, January 2003.
29. I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley, 1997.
30. M. de Jonge. The Linux kernel as flexible product-line architecture. Technical Report SEN-R0205, CWI, 2002.
31. M. de Jonge. Pretty-printing for software reengineering. In *Proceedings: International Conference on Software Maintenance (ICSM 2002)*, pages 550–559. IEEE Computer Society Press, October 2002.

32. M. de Jonge. Source tree composition. In C. Gacek, editor, *Proceedings: Seventh International Conference on Software Reuse*, volume 2319 of *Lecture Notes in Computer Science*, pages 17–32. Springer-Verlag, April 2002.
33. M. de Jonge. Package-based software development. In *Proceedings: 29th Euromicro Conference*, pages 76–85. IEEE Computer Society Press, September 2003.
34. M. de Jonge. *To Reuse or To Be Reused: Techniques for Component Composition and Construction*. PhD thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, January 2003.
35. M. de Jonge. Decoupling source trees into build-level components. In J. Bosch and C. Krueger, editors, *Proceedings: 8th International Conference on Software Reuse*, volume 3107 of *Lecture Notes in Computer Science*, pages 215–231. Springer-Verlag, July 2004.
36. M. de Jonge, E. Visser, and J. Visser. XT: a bundle of program transformation tools. In M. G. J. 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*, pages 211–218. Elsevier Science Publishers, April 2001.
37. 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, 1990.
38. D. Mackenzie, B. Elliston, and A. Demaile. *Autoconf: creating automatic configuration scripts*. Free Software Foundation, 2002. Available at `http://www.gnu.org/software/autoconf`.
39. D. Mackenzie and T. Tromey. *GNU Automake Manual*. Free Software Foundation, 2003. Available at `http://www.gnu.org/software/automake`.
40. G. Matzigkeit, A. Oliva, T. Tanner, and G. V. Vaughan. *GNU Libtool Manual*. Free Software Foundation, 2004. Available at `http://www.gnu.org/software/libtool/`.
41. H. Mei, L. Zhang, and F. Yang. A software configuration management model for supporting component-based software development. *ACM SIGSOFT Software Engineering Notes*, 26(2):53–58, 2001.
42. P. Miller. Recursive make considered harmful. *AUUGN*, 19(1):14–25, 1998. Available at `http://aegis.sourceforge.net/auug97.pdf`.
43. P.-E. Moreau. A choice-point library for backtrack programming. In *JICSLP'98 Post-Conference Workshop on Implementation Technologies for Programming Languages based on Logic*, June 1998.
44. R. van Ommering. Configuration management in component based product populations. In *Tenth International Workshop on Software Configuration Management (SCM-10)*. University of California, Irvine, 2001.
45. 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.
46. The Online Package Base. `http://www.program-transformation.org/package-base`.
47. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
48. R. Russell, D. Quinlan, and C. Yeoh. Filesystem hierarchy standard. Technical report, Filesystem Hierarchy Standard Group, 2004.
49. S. Sowrirajan and A. van der Hoek. Managing the evolution of distributed and interrelated components. In B. Westfechtel and A. van der Hoek, editors, *Software Configuration Management*, volume 2649 of *Lecture Notes in Computer Science*, pages 217–230. Springer-Verlag, 2003.
50. T. van der Storm. Variability and component composition. In J. Bosch and C. Krueger, editors, *Proceedings: 8th International Conference on Software Reuse*, volume 3107 of *Lecture Notes in Computer Science*, pages 86–100. Springer-Verlag, June 2004.
51. Stratego/XT. `http://www.strategoxt.org`.
52. C. Szyperski. *Component Software: Beyond Object-Orientated Programming*. Addison-Wesley, second edition, 2002.
53. D. B. Tucker and S. Krishnamurthi. Applying module system research to package management. In *Tenth International Workshop on Software Configuration Management (SCM-10)*. University of California, Irvine, 2001.
54. E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.
55. 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 *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.
56. D. Whitgift. *Methods and Tools for Software Configuration Management*. John Wiley & Sons, 1991.
57. L. Zhang, H. Mei, and H. Zhu. A configuration management system supporting component-based software development. In *Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, pages 25–30. IEEE Computer Society Press, 2001.