Developing Product Lines with Third-Party Components

Merijn de Jonge [1]

Philips Research
Eindhoven, The Netherlands

Abstract

The trends toward product line development and toward adopting more third-party software are hard to combine. The reason is that product lines demand fine control over the software (e.g., for diversity management), while third-party software (almost by definition) provides only little or no control.

A growing use of third-party software may therefore lead to less control over the product development process or, vice-versa, requiring large control over the software may limit the ability to use third-party components. Since both are means to reduce costs and to shorten time to market, the question is whether they can be combined effectively.

In this paper, we describe our solution to this problem which combines the Koala component model developed within Philips with the concept of build-level components. We show that by lifting component granularity of Koala components from individual C files to build-level components, both trends can be united. The Koala architectural description language is used to orchestrate product composition and to manage diversity, while build-level components form the unit of third-party component composition.

Key words: Koala, software product lines, build-level components, third-party sofware, software composition.

1 Introduction

Cost reduction and time to market are driving factors for developing software product lines. Often proprietary technologies are used for managing diversity to enable quick product development. Until recently, it was common practice to develop most (if not all) software in-house, especially for industrial product line architectures. Today, the trend toward reduction of cost and time to market is continued, namely by adopting more third-party software, including open source software. The expectation is that by adopting third-party software, the software quality and time to market can be improved, while keeping the development costs down.

[1] Email: Merijn.de.Jonge@philips.com

Unfortunately, third-party software usually does not integrate well with proprietary product line technology. Product-line technology typically demands specially flavored components and fine-grained control over software artifacts. Since product-line technology is not standardized, it is very likely that third-party software does not fit, and that it is not allowed or possible to make it fit. As a result, either the use of third-party software is preferred over product-line technology, or the other way around. Clearly, this is not an optimal situation.

Koala [10,11] is such a proprietary component technology for creating product lines. It has been used successfully for about 10 years for defining and constructing a large variety of products. Its key features are diversity management by means of composition and variation, and the architectural description language (ADL) to define component compositions and to drive code generation.

Despite its proven success, Koala is currently not sufficiently capable for adopting external software. Because it is proprietary technology, it is not an industrial standard. As it assumes specially tailored component implementations, it requires effort to make third-party components fit. Moreover, since it operates at the level of individual C files, it is difficult to integrate more coarse-grained components. Consequently, in its current form Koala is not capable of orchestrating the composition for products that consist of both proprietary and third-party components.

Independently of Koala, the concept of build-level components [4] has been developed. A build-level component consists of source code together with instructions to build (e.g., compile/link) it. It provides a build and configuration interface to abstract over the component-specific build and configuration processes. As a consequence, build-level components can be composed and bound in a uniform way, similar to components in other component models. Build-level components are simple to develop and comply with standard technologies. Therefore, they are promising to improve the composability of third-party software at the build-level.

In this paper we explore the idea to combine the Koala component model and build level components to enable product line development with third-party software. We show how we can use Koala to orchestrate the composition and variability of build-level components, by leveraging the granularity of components from C modules to build-level components. This way, we can adapt to the trend to use third-party components, rather than using only proprietary software. With this approach, we can benefit from Koala's architectural language, diversity mechanism, and tooling. This way, the use of Koala for product line development can be continued, even in the heterogeneous environments of tomorrow.

This article is structured as follows. Section 2 gives a brief overview of the Koala component model. In Section 3 we zoom into Koala's module concept, which is of key importance for this article. Section 4 gives a quick overview of build-level components. In Section 5 we explain how Koala and build-level components can be combined. In Section 6, we discuss the implementation of build-level composition with Koala. In Section 7, we discuss the development of the Koala compiler as a composition of build-level components. In Section 8 we draw final conclusions.

2 Koala in a nutshell

Koala [10,11] is a component model consisting of an architectural description language (ADL) and tool support. The ADL serves to define interfaces, data types, basic components, and compositions (which are components themselves). The tooling serves to generate products from component compositions. Koala was primarily designed for resource-constrained software and is applied in the consumer electronics domain.

Koala is a hierarchical component model where larger components are constructed by instantiating smaller components. The leaves of a composition tree are formed by Koala modules, which correspond to individual C files. [2] The Koala tooling includes a compiler which creates bindings between components, creates code to manage unbound diversity parameters, and which generates a script to compile/link a composition. Below we will briefly describe a subset of the Koala ADL.

Interface definitions
In its simplest form, a Koala interface definition consists of a sequence of function prototypes, parameters, and constants in a C-like syntax. For example:

```
interface IFooBar {
   int foo(void);
   int bar(int c, int y);
   int a_constant = 10;
   int a_parameter;
}
```

This interface defines two functions (`foo` and `bar`), a constant (`a_constant`), and a parameter (`a_parameter`).

Component definitions
A component definition consists of a name and a list of sections. For example:

```
component FooBar {
   provides IFooBar p;
   requires IFoo rFoo;
            IBar rBar;
   contains
      component Foo foo;
      component Bar bar;
      module m;
   connects
      p = m;
      m = rFoo;
      m = rBar;
}
```

This component definition defines the component `FooBar`. It specifies three interface instances: one provided interface of type `IFooBar`, and two requires interfaces of type `IFoo` and `IBar`, respectively. The component definition further

[2] Conceptually, the Koala component model is not bound to the C language (see Section 3), but in practice only C is supported by Koala tooling and all existing components are written in C.

contains two component instances (one instance of `Foo` and one instance of `Bar`) and one module `m`. Observe that, in contrast to interfaces and components, modules have no type and are implicitely instantiated. Finally, the component definition specifies how interfaces are connected. The provided interface `p` is connected to the module `m`. This means that each function call to interface `p` is routed to the module `m`. This module must implement these functions. Module `m` is connected to `rFoo` and `rBar`. This means that module `m` can use the functions from these interfaces, although it is still undefined where these functions are implemented. Connections of modules are untyped. Hence, any interface can be connected to a module.

Configurations
In order to build a product from a set of components, a configuration has to be defined. A configuration is a component definition without provides or requires interfaces. It must bind all unbound requires interfaces of its constituent component instances. In the example above, this means binding the `rFoo` and `rBar` interfaces.

Modules
A module is Koala's atomic unit of composition. It typically corresponds to a C file. A Koala module cannot be instantiated. A C file corresponding to a Koala module must meet a few rules: it should have exactly one specific #include statement, and it should use a specific naming convention for implementing functions from provides interfaces and for accessing functions from requires interfaces, for example:

```
#include "FooBar.h"
void p_foo() {
   r_foo();
}
int p_bar(int x, int y) {
   return r_bar(x,y);
}
int p_a_parameter = 10;
```

As can be seen from this example, functions from interfaces are prefixed with their instance name. These are called logical names in Koala. In its simplest form, compilation of a Koala composition consists of binding C function calls to C function definitions according to the interface bindings in the Koala composition. A binding is accomplished by mapping the logical names of a function call and definition to a common physical name. These bindings have the form of:

```
#define logical_name physical_name
```

The single file that each C module must include, contains, amongst others, such bindings. This file is generated by the Koala compiler. The file `FooBar.h` in the example might include the following bindings:

```
#define p_foo Main_FooBar_foo
#define r_foo Main_Comp_Demo_foo
#define r_bar Main_Test_Bar_bar
```

Diversity
Diversity is, amongst others, supported via the switch construct. For example:

```
switch expr
   in  {i1 i2}
   out {o1,o2} on v1
       {o3,o4} otherwise
```

This switch connects the pair of interfaces `i1` and `i2` to either `o1` and `o2` in case `expr` evaluates to `v1`, or to `o3` and `o4` otherwise.

The Koala compiler performs partial evaluation in order to evaluate switch conditions whenever possible. If a switch condition can be statically determined (based on parameter bindings in interfaces), the corresponding route through the switch is lifted into a static binding, and the remaining routes are removed. This forms an important optimization, which allows defining a huge configuration space in the ADL, without having significant impact on the run-time behavior and/or size of the resulting executable, since most variation points for a product are bound statically and optimized away. For variation points which are not statically bound, the Koala compiler generates code to deal with the variation point at run time.

3 A closer look at Koala modules

A Koala module is untyped. This means that any interface can be connected to a module. The reason is that a module forms the connection between the architectural level and the realization level. Since Koala has no notion about C, it cannot type check a C file to verify that a binding to the file is correct. Hence, at the architectural level, we are concerned with interfaces and components and how they are connected. At the realization level we are concerned with C files (and to some extent, libraries) and how they have to be compiled into an executable. Since a Koala module corresponds to a concrete piece of source code, modules cannot be instantiated, because that would imply pure code duplication.

The Koala language (and concepts) and the Koala compiler (producing C code) are developed and used together. As a consequence, they are often seen as one, integral combination. Actually, there is hardly any experience with decoupling the Koala language from its compiler and from the C programming language. However, conceptually, Koala is a hierarchical component architecture, where modules are the atomic unit of composition. Thus, Koala component composition amounts to composing Koala modules. By associating Koala modules to a concrete composable type of artifact, Koala becomes a component architecture for that type of artifact. The association of Koala modules to C source files is just one possible association. It is important to understand that the Koala language is agnostic for a particular association; it is the Koala compiler which is aware of it.

Essentially, the Koala compiler consists of two parts: a front-end and a back-end. The front-end is concerned with the Koala language. This includes type checking, synthesizing interface bindings, and partial evaluation. The result of the front-end is a Koala composition normalized into some canonical form. The back-end is concerned with code generation. Given a normalized Koala composition it generates proper bindings for the target language (e.g., #include's in case of C).

`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: Build interface.

`--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: Configuration interface.

Table 1
Build interface and configuration interface of build-level components.

Thus, it is the compiler back-end which is aware of a particular module association. This split-up enables the use of the Koala language and front-end to drive the composition of other artifacts than C source files. It is this observation that gave rise to the idea of this paper to extend the application of Koala to more coarse-grained components to enable adoption of third-party software. For this purpose, we will propose build-level components as atomic units of composition.

4 Build-level components in a nutshell

The build-level is concerned with building software products from source files. This is called a software build process, and typically includes tasks like compiling and linking. The build-level is therefore concerned with (source) files, their structuring in directories, compilers (and related tools and their settings), dependencies (i.e., what other software is needed), and configuration (i.e., controlling what features need to be built, and how to build them).

The build process is inherently difficult because there are many different programming languages requiring different steps in the construction process, because there are numerous tools available, because standardization is missing, and because the build process is (most often) not designed for composition. Build processes are therefore often custom-made and extremely difficult to understand and maintain. Moreover, they often work only in very specific environments (due to implicit dependencies it is hard to reconstruct such environments). Finally, they cannot easily be integrated in another build process.

The motivation for build-level components is to improve this situation. Build-level components introduce a few development rules, which bring principles from component-based software engineering (CBSE) to the build-level [4]. In particular, a build-level component is a unit of independent deployment, a unit of third-party composition, and has no (externally) observable state [8].

The concept of build-level components promotes decomposing large software systems into smaller reusable building blocks. These can be composed in a systematic way to form complex software systems. They abstract from their internal details (e.g., how they are exactly built, and their source file organization in directories) by offering a build, a configuration, and a requires interface. A build interface

specifies the steps needed to build the component, a configuration interface specifies the variation points of the component, and a requires interfaces specifies the needs of a component (i.e., its dependencies). In [4], we proposed to comply with the open source community and adopt the syntax of autotools [12] for defining build and configuration interfaces (see Table 1a and 1b). Requires interfaces are bound via `--with-` switches of a component's configuration interface.

Below we give a small example of a build-level component which makes use of GNU Automake and Autoconf. The component is called `Foo` and consists of a single library `libfoo.a`. Its implementation is contained in the C source file `libfoo.c`. The component has one dependency on the component `Bar`. Its build interface is generated by Automake from the following build process description:

```
lib_LIBRARIES     = libfoo.a
libfoo_a_SOURCES  = libfoo.c
libfoo_a_CPPFLAGS = -I$(BAR)/include
```

This Automake makefile defines that `libfoo.a` will be the result of the build process. This library is defined in `libfoo.c`, using the single dependency `Bar`. By using GNU Autoconf, a proper build-level configuration interface can be automatically generated from the following `configure.ac` file:

```
AC_INIT(foo,0.1,you@your.organization)
AM_INIT_AUTOMAKE
AC_PROG_CC
AC_ARG_WITH([bar], AS_HELP_STRING(...), [BAR=${withval}])
AC_SUBST([BAR])
AC_CONFIG_FILES([foo/Makefile])
AC_OUTPUT
```

This configuration file specifies the name, version, and maintainer of the build-level component. Of further importance is the definition of the dependency parameter for component `Bar`. The resulting configuration script will understand the `--with-bar` switch and passes the value to the makefile via the variable `BAR`.

5 Linking Koala modules to build-level components

In this section we show how the Koala compiler front-end can be used together with a new back-end to form a composition tool for build-level components. This unites the desire to have an ADL for describing compositions and managing diversity with the desire to adopt third-party software. Since build-level components are fairly easy to construct (and the design rules of [4] are good to apply anyway), the effort of integrating third-party software at the build-level is reduced. The essential idea is to:

- Define interfaces, components, component compositions, component bindings, and diversity bindings in the Koala ADL.
- Use the Koala front-end tools to manage diversity, analyze correctness, and to bring a composition into normal form.
- Given the normalized composition specification, generate a build-level realiza-

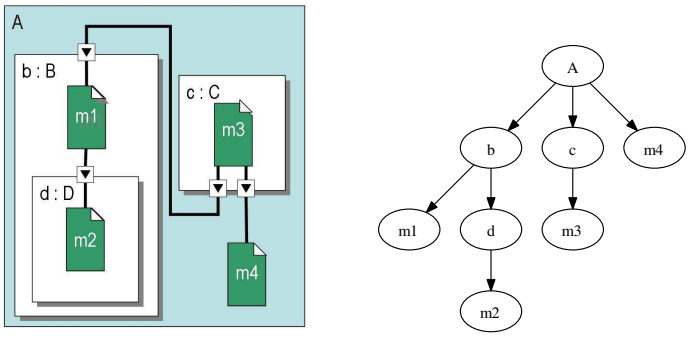

Fig. 1. Koala component configuration and corresponding composition tree.

tion of this composition.

Realizing a build-level composition involves creating a directory hierarchy containing a set of component instances, integrating build processes in a top-level build process definition, and integrating component configuration by synthesizing bindings for requires interfaces and diversity parameters. These will be described below.

5.1 A build-level composition hierarchy

A normalized composition specification defines a composition tree where nodes represent components and leaves represent modules. At the build-level we will follow this composition tree. Directories will represent components and modules. A Koala module maps to a build-level component. A build-level component is placed in the directory of the corresponding module. A directory for a component is a container which delegates all build-level operations to its subdirectories.

Figure 1 shows a Koala configuration and the corresponding composition tree. At the build level the same structure is created: the four build-level components, corresponding to the modules are placed in the directories `m1`, `m2`, `m3`, and `m4`; the directories `b`, `c`, and `d` are containers which add structure to the build-level. [3]

5.2 Build-level component instances

A build-level component instance is a directory containing the contents of a build-level component. For every Koala module a separate directory is created in which a build-level component is stored. That is, each directory that corresponds to a Koala module contains an instance of a build-level component. This means that multiple instances of build-level components may exist. The ability to have multiple instances of a build-level component, enables simultaneous use of different versions of the same component, and the use of different configurations of the same component. However, it depends on how component instances are used, whether multiple instance will work or not. For example, a single executable may typically not use

[3] In [3] we described an approach for build-level component composition where the composition tree is always flat. A flat component structure hinders real abstraction, because a component cannot embed another component instance. The structuring in (sub) directories, that we propose now, is much more flexible and allows for composite components at the build-level.

two different versions of a library at the same time.

5.3 Top-level build-process definition

A build-level composition is a build-level component itself. This implies that the composition has its own build process definition. This build process is simple because the top level directory merely serves as container. Hence, most build actions are simply delegated to the subdirectories. The `dist` and `distcheck` actions are special and cannot be delegated. These actions construct a distribution by packaging all needed files from the composition.

A composite build process is a sequential composition of build processes. Individual build processes cannot be executed in arbitrary order. They must take component dependencies into account. A sequential build process is formed from a composition tree by traversing the tree in depth first order, in such a way that if a build-level component in the subtree rooted at x has a dependency on a build-level component in the subtree rooted at y, then y is traversed before x. This ensures that a build-level component has been built before it is used. Observe that for components with circular dependencies it is not possible to determine a correct build-order. Such components have to be refactored [4].

5.4 Build-level interface bindings

Koala has three types of interfaces: provides interfaces, requires interfaces, and diversity interfaces. All three types can be connected to modules. A Koala backend takes care of realizing interface bindings. In case of build-level components, this amounts to binding requires interfaces and diversity parameters.

Binding a requires interface `r`, implies specifying a value for the `--with-r` switch of a component's configuration process. Binding a diversity interface `d`, implies specifying a value for `--with-v` for each interface element `v` of `d` (e.g., `--with-v=value`).

In case a binding is static (i.e., when its value evaluates to a constant when running the Koala compiler), binding is straight forward. It is similar to the binding mechanism described in [3], and amounts to running the configure tool with the particular binding in the form of a `--with-` switch.

In Koala, configuration can be more complex because also run-time interface binding is supported. In case of C, run-time interface binding means that at execution time a switch condition is evaluated to determine a function binding. In case of build-level components, run-time means when running the configure tool. The values to the `--with` switches are then synthesized dynamically when running `configure`. In the next section we will discuss the technical details.

The C-version of the Koala compiler requires that top-level components cannot have interfaces themselves (such components are called configurations). This is essential because the compiler produces executables and without special technology, compiling and linking an application with unbound requires interfaces will result in unresolved symbols errors. In case of build-level components, this requirement is not needed. Therefore, every build-level composition is itself a component, which

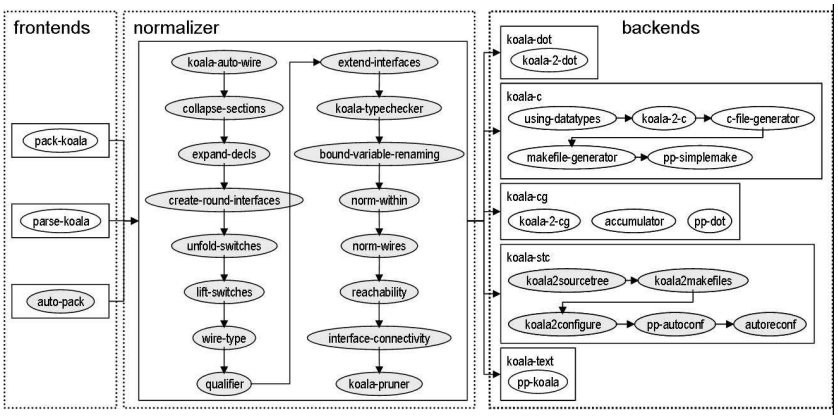

Fig. 2. Composition of run-time components of the Koala compiler (product line).

can have as many provides, requires, and diversity interfaces as needed.

By linking Koala modules to build-level components, we obtain a powerful means to construct build-level compositions. It enables the construction of software systems from coarse-grained build-level components. It forms a consistent way of constructing software systems, instead of building them from collections of individual, yet dependent, blocks, which need to be built and configured independently, or for which integration implies manually developing complex build scripts. It does not offer the fine granularity as Koala for composing C modules, though.

6 Implementation

We developed a prototype Koala compiler to experiment with the Koala component architecture and concepts. It was implemented as a pipeline of transformations using StrategoXT [2]. Each transformation takes care of a particular compilation step. This enabled us to easily experiment with new language constructs, and to adapt/extend the pipeline to our needs. The compiler is a product line where the input processing mechanism and the produced output are variation points. Figure 2 depicts the composition of run-time components of the Koala compiler. The gray-colored nodes form the pipeline for realizing build-level component compositions.

Due to space limitations, we will only discuss the code generation part of this pipeline which is performed in the `koala-stc` backend. This back-end creates a directory structure, downloads and unpacks build-level components in this directory structure, creates Automake makefiles for each Koala component definition, and it creates a top-level Autoconf configuration script.

Build-level components are unpacked in the directories corresponding to Koala modules (see Section 5). After synthesizing the build order from module dependencies, makefile generation is trivial. Each makefile consists of a single statement:

```
SUBDIRS = A B C
```

Where `A`, `B`, and `C` denote subdirectories in the correct build order. Synthesizing the build order is a recursive process, where at each node in the composition tree the sibling nodes are placed in the right order depending on the module dependencies

in the corresponding subtrees.

The more complicated part of the back-end is generating a top-level Autoconf configuration script that drives the configuration of all build-level components. In general, an Autoconf configuration script processes configuration switches, drives the execution of configuration processes in subdirectories, and generates makefiles. It also performs platform checks, but these are not relevant for our discussion. Generating a configuration script for our purposes is not trivial because it has to deal with realizing interface bindings between build-level components, and because it has to deal with managing and delegating (dynamic, configuration-time) bindings.

Unbound interfaces

Unbound requires and diversity interfaces (of the top-level component) are added to the top-level configuration interface. For unbound requires interfaces this involves adding the interface instance name to the configuration interface. For unbound diversity interfaces, it amounts to adding for each element `e` of the interface `d` the parameter `d-e` to the configuration interface. This way, elements of diversity interfaces can be bound individually. An element `e` from diversity interface `d` can be bound to `v` with the switch `--with-d-e=v`. A requires interface `r` can be bound to `v` by passing the switch `--with-r=v` at configuration time.

Realizing build-level bindings

Bindings of build-level components are realized at configuration time by passing bindings in the form of `--with-` switches to the individual configuration processes. Requires interfaces are bound to modules at the end of corresponding connection chains. These connection end-points are synthesized by the koala normalizer. Diversity interfaces are bound to the values synthesized by the Koala compiler. In case a requires/diversity interface is not bound by the Koala compiler, its binding can be specified at configuration time via the top-level configuration interface. Configuration-time bindings must be defined for mandatory requires interfaces or else an error is raised. For diversity and optional requires interfaces such bindings can be left out.

Dynamic diversity

Since connection chains may have forks due to switches, realization of bindings has to be carried out dynamically by evaluating switch conditions at configuration time. If forked chains cannot be reduced to single end-points, because switch conditions cannot be evaluated, a binding cannot be realized and an error is raised.

We implemented this behavior for GNU Autoconf via a set of M4 macro's. These macro's can express the Koala binding structure of a composition. The macro's expand to chains of function calls which represent the dynamic behavior of Koala bindings. For instance, for a simple switch statement that connects an interface `r3` to `r1` or `r2` depending on the `debug` field in an diversity interface `A_config`, the following code is generated (using the macro's `KOALA_SWITCH` and `KOALA_SWITCH_OUT`):

```
KOALA_SWITCH(
 [A_switch],
 [expr=$(A_config debug)],
```

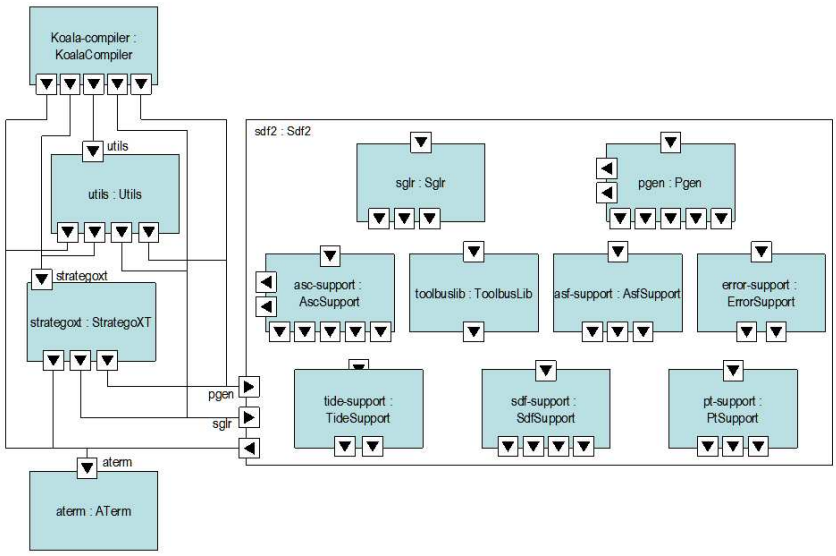

Fig. 3. The structure of the Koala compiler in terms of build-level components.

```
[KOALA_SWITCH_OUT([${expr}],["yes"],[A_r1])
 KOALA_SWITCH_OUT([],[],[A_r2])])
```

This code is then expanded by GNU Autoconf into the following shell script code that will be part of the configure script:

```
A_switch() {
 expr=$(A_config debug)
 if test "a${expr}" = "ayes"; then
  echo $(A_r1 $*)
 elif test "a" = "a"; then
  echo $(A_r2 $*)
 fi}
```

This fragment shows the function that is generated for the switch statement. A call to this function results in another function call, to either `A_r1` or `A_r2`, depending on the evaluation of the switch expression. This demonstrates how the binding of `r3` is dynamically evaluated at configuration time.

As indicated before, the result of a composition of build-level components is a build-level component itself. The generated configure script forms its configuration interface, the top-level makefile forms its build interface. The new component forms a unit of composition and hides its internal structure (i.e., its internal build-level components).

7 Case Study

In this section we demonstrate how we can define our Koala compiler in terms of itself (i.e., as a composition of Koala components) and how this composition can be realized by actually running the compiler. We show how Koala allows structuring components to our needs, how nested components can be (re-)used in larger compositions, and how we can easily integrate third-party components.

Figure 3 depicts the build-level structure of the Koala compiler. Observe that this build-time structure is quite different from its run-time structure depicted in Figure 2. Essentially, the build-level structure consists of five building blocks: i) the ATerm library [9] for data representation and exchange, ii) the sdf2 component for parse technology, iii) the StrategoXT program transformation environment, iv) a language tools package, and v) the Koala compiler itself.

There are multiple ways to organize the structure of these blocks. Our example shows a composition in terms of 4 flat components, and 1 nested component (`sdf2`). The nested component consists of a parser (`sglr`), a parser generator (`pgen`), and several utility components (to improve readability we omitted the interface bindings within component `sdf2`). Since `sglr` and `pgen` are mostly used together, they are placed within the same component. We deliberately did not export the interfaces of the utility components because we consider them as implementation components. In the rare case that they are needed elsewhere, Koala allows to create additional instances of them. This is a good example of how Koala enables structuring of components to hide component implementations. Once defined, the `sdf2` component can be instantiated as a single component. All other components of Figure 3 are also being used independently. Therefore, we did not put them into any nested structure.

Given a set of component definitions, our Koala compiler can perform a composition in three ways:

(i) By specifying a set of components, the Koala compiler synthesizes a composition by transitively finding components that implement the requires interfaces of the components that are part of the composition. This is the most implicit form because no closed set of components is specified and no interface bindings are specified.

(ii) By explicitly specifying which components should be part of a composition, the Koala compiler synthesizes correct bindings between them. In this form the set of components is specified, but the bindings between them are not.

(iii) By explicitly specifying all components, and all their bindings. This is the most explicit form because all ingredients and bindings are specified.

For the implicit form, we don't need a top-level component definition. Instead, we pass a set of mandatory components to the Koala compiler, and let the compiler synthesize a top-level composition. In our example, we can create such a composition using the following command:

```
koala-stc --parser auto-pack -I . -d out -c KoalaCompiler
```

This command specifies that the `auto-pack` tool for implicit component composition has to be used, that component definitions can be found relative to the current directory ('.'), that output should be placed in 'out', and that `KoalaCompiler` is a mandatory component in the composition. After the Koala compiler is ready, one can configure and build the composition by running `configure` and `make` in the directory `out`. To finalize the creation of a build-level component from this composition, one issues the command `make dist`. This bundles all components into

a single archive file, which is ready for distribution.

Apart for the Koala compiler, all components depicted in Figure 3, are third-party (open source) components. They have been developed at different institutes, they are implemented in different programming languages, and they are being used in many different software systems. What the components have in common is that they follow the build-level rules defined in [4]. They are therefore directly usable for build-level composition.

8 Concluding Remarks

Contributions

This paper addressed the problem that on the one hand a growing need is developing to adopt third-party (open source) software, and on the other hand, that fine control is needed over the software to enable quick and reliable product development. Most often, these are conflicting demands, because one has (by definition) little control over third-party-software, while to speed product development, technologies for e.g., diversity management and product line development are required that do need significant control.

As a consequence, a potential risk exists that a growing use of third-party software goes in hand with less control over the product development process, or, the other way around, that requiring large control over the software limits the ability to use third-party software.

We observed this trend within Philips, where third-party (open source) software did not integrate seamlessly with the successful Koala component model. In this paper, we describe a possible solution to this problem by combining the Koala component model and the concept of build-level components. We have shown that by lifting component granularity of Koala components from individual C files to build-level components, both demands can be united. The Koala ADL can be used to orchestrate product composition and to manage diversity, build-level components can form the unit of third-party component composition.

In this paper we explained how Koala modules form the mapping from the architectural level to the realization level, and that they can represent arbitrary units of composition (in addition to plain C files). We then defined how a composition of Koala modules can be mapped to a composition of build-level components. Next, we extended our Koala compiler product-line with a new back-end to automate the realization of compositions at the build-level. Finally, we showed how our approach can be used in practice, by defining the Koala compiler in terms of a composition of build-level components.

Related Work

There are several alternative composition lanagues to Koala. See e.g., Few technologies exist which address the topic of orchestrating composition and diversity of third-party (open source) components. For instance, typical package managers, like RPM [1], merely address building/installing single packages. There are two chal-

laging tools around that combine diversity, composition, and third-party software. The first is GEARS [7]. This is a software product line development tool, which explicitly supports the integration of existing (i.e., unchanged) software. This implies that GEARS can deal with software over which no control exists. The second system is the Nix deployment system [6,5]. Nix is a promising tool for safe software deployment. Underneath is a functional language that provides advanced diversity features. Nix was explicitly designed for managing diversity and safe orchestration of open source software composition.

Availability

The Koala compiler product line is distributed as open source. It can be downloaded from http://www.program-transformation.org/Tools/KoalaCompiler.

References

[1] Bailey, E. C., "Maximum RPM," Red Hat Press, 1997.

[2] Bravenboer, M., K. T. Kalleberg, R. Vermaas and E. Visser, *Stratego/XT 0.16. Components for transformation systems*, in: *ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM'06)* (2006), pp. 95–99.

[3] de Jonge, M., *Source tree composition*, in: C. Gacek, editor, *Proceedings: Seventh International Conference on Software Reuse*, LNCS **2319** (2002), pp. 17–32.

[4] de Jonge, M., *Build-level components*, IEEE Transactions on Software Engineering **31** (2005), pp. 588–600.

[5] Dolstra, E., M. de Jonge and E. Visser, *Nix: A safe and policy-free system for software deployment*, in: L. Damon, editor, *18th Large Installation System Administration Conference (LISA '04)*, USENIX, Atlanta, Georgia, USA, 2004, pp. 79–92.

[6] Dolstra, E., E. Visser and M. de Jonge, *Imposing a memory management discipline on software deployment*, in: J. Estublier and D. Rosenblum, editors, *26th International Conference on Software Engineering (ICSE'04)*, IEEE Computer Society, Edinburgh, Scotland, 2004, pp. 583–592.

[7] Krueger, C. W., *Easing the transition to software mass customization*, in: E. S. Institute, editor, *Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4)*, Bilbao, Spain, 2001, pp. 265–277.

[8] Szyperski, C., "Component Software: Beyond Object-Oriented Programming," Addison-Wesley, 1999.

[9] van den Brand, M., H. A. de Jong, P. Klint and P. A. Olivier, *Efficient annotated terms*, Softw, Pract. Exper **30** (2000), pp. 259–291.

[10] van Ommering, R., "Building Product Populations with Software Components," Ph.D. thesis, University of Groningen (2004).

[11] van Ommering, R., F. van der Linden, K. Jeff and J. Magee, *The Koala component model for consumer electronics software*, Computer **33** (2000), pp. 78–85.

[12] Vaughan, G. V., B. Elliston, T. Tromey and I. Taylor, "GNU Autoconf, Automake, and Libtool," (2006), available at http://sourceware.org/autobook/.