Multi-level Component Composition

Merijn de Jonge
Utrecht University, P.O. Box 80089,
3508 TB Utrecht, The Netherlands
mdejonge@cs.uu.nl

Abstract

Software components are a popular means to improve the practice of software engineering. They serve to increase productivity by promoting software reuse and increasing software quality. Component composition is the process of assembling software components. There are many forms of components, each demanding a specific form of composition. These are all orthogonal notions of the same principle of component-based software engineering.

We will explore the vision that software components are themselves compositions of components. This gives rise to the idea of composition levels, where compositions at level i serve as components at level $i+1$. *As a consequence, components at different levels are not so independent as they are usually treated. One significant aspect of components that crosses component levels is variability.*

In this paper, we will discuss preliminary work on multi-level composition. This includes general requirements for level-specific composition and a framework that supports multi-level composition.

1 Introduction

Composition plays a prominent role at different levels of software systems. Each level has a different notion of a component. For instance, at run-time components may exist in the form of COM or EJB components, at deployment-time a component has the form of a package (e.g., an RPM [1] package), and at compile-time components exist in the form of modules or classes. Components at different levels have different characteristics, such as: component granularity, composition time, variability mechanism, composition manager, component definition language (CDL). Components at different levels are therefore often handled as orthogonal entities, although in reality they are not.

Considering different component levels as independent has several consequences:

- No uniform model of all components in a system. At each level live different components, which are usually not captured in a single component model.
- No uniform model of all variability. Variation points at different levels are usually not integrated in a single model.
- No flexibility in binding time. That is, binding variability parameters is not transparent across composition levels, but dedicated to a particular (usually low) level.
- Strong coupling between component model and source code. Since bindings are often created in source code, composition tools are highly language-specific, and source code becomes dependent on a component model due to coding conventions.

There where full control over the development process and components exists, and where the software development activity can relatively easy be managed, the aforementioned consequences might not form serious problems. However, with third party components or components written in different languages, or when software systems are huge and developed at many product divisions, they certainly do.

To improve this situation, we propose to use a single component model for multiple composition levels, where the notion of a component is variable. Further, we define how concepts from the component model can be represented at each level, and we define a mapping of variability from level $i+1$ to level i. A composition at level i can then be handled as an atomic entity at level $i+1$. Component levels can even be mixed, which allows compositional optimizations at the component-model level. We call this *multi-level* composition. The expected benefits of this approach are:

- Composition of source and binary components.
- Compositions with third-parties are enabled.

- Arbitrary compositions can be made (i.e., that compositions can be made of subsystems, applications, groups of applications, etc.).
- Multi-lingual compositions are enabled.
- Composition moment is controllable.

In this paper we describe preliminary work on multi-level composition. We use Koala as uniform component model where a *module* serves as atomic entity at level *i*. At each level, a composition backend performs a level-specific composition. We identify what requirements a component should fulfill in order to be subject to multi-level composition. Finally, we describe a framework that supports multi-level composition.

The paper is structured as follows. Section 2 discusses components and composition-levels. Section 3 motivates multi-level composition. Section 4 discusses the Koala component model. Section 5 describes how Koala's composition model can be generalized to support multi-level composition. Sections 6 and 7 describe an architecture for multi-level composition. In Section 8 this architecture is further generalized to support cross-level composition. Section 9 summarizes results and discussed future work.

2 Components & Composition

A component is an abstraction: it hides an implementation behind interfaces; all component access occurs via interfaces. A component is also a unit of reuse and subject to third-party composition [12]. The purpose of a component is to provide functionality that can be used in different contexts. This functionality is accessible through a component's provides interface. Components may have multiple provide interfaces. A component can depend on functionality offered by other components. The functionality that is required by a component forms a component's requires interface. A component may also have multiple of these. A component with a requires interface can be bound to any component that implements this interfaces. Thus, by specifying functionality in terms of interfaces, no dependencies on concrete components are introduced. This property makes components independently deployable. However, non-functional properties of components, such as performance characteristics, may yield such component dependencies. These are ignored in this article.

To improve reusability of a component it can often be adapted for specific needs. The ways in which components can be adapted is called variability (or diversity) [5]. Adapting a component is called configuration. Components are adapted through configuration (or diversity) interfaces.

Components are used as building blocks to form larger software entities. Assembling components is called composition. Composition involves putting components together and connecting provided functionality to required functionality. Composition can be static or dynamic. With static composition the collection of components that form an application is statically known. With dynamic composition the composition of components is determined dynamically, e.g., at run-time. For instance, the COM component model [2] is concerned with dynamic component composition. In this article we only consider static composition.

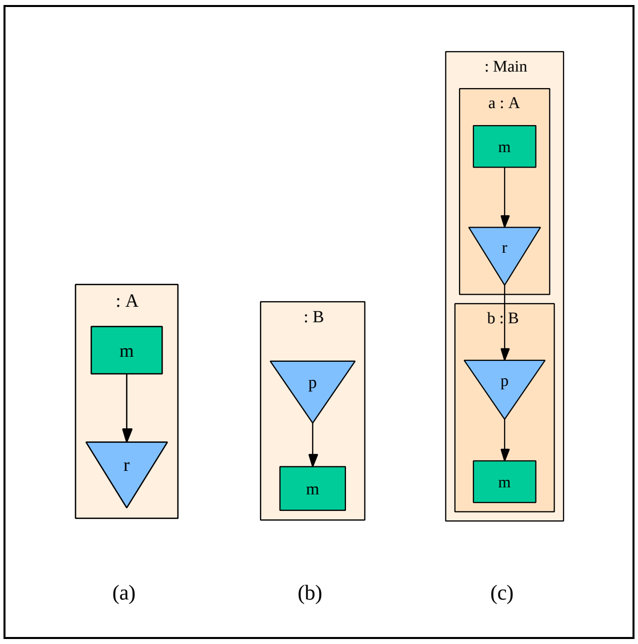

Figure 1. Components and composition.

Components and compositions can be graphically represented. In this paper we use a graphical representation that is based on Koala [11]. Figure 1 shows some examples. The picture depicts interfaces as triangles, connections as edges, modules (which correspond to atomic entities) as small boxes, and components as large boxes. Figure 1(a) depicts a component with a requires interface, Figure 1(b) a component with a provides interface, and Figure 1(c) depicts a composition of the two.

Components exist as abstraction for different kinds of entities. For instance, they exist as abstraction for source modules, libraries, applications, RPMs, classes, and so on. Different forms of components may have a different granularity. For instance, source modules are fine-grained components, whereas RPMs are more coarse-grained. As a result of different component forms, variability and composition mechanisms exist in many flavors.

The different forms and granularity of components give rise to the idea of composition levels, where subsequent levels have increasing granularity. We can classify components according to component levels, as depicted in Table 1. Ob-

Table 1. Different composition levels.

level	*atomic entity*		*composite entity*
0	function+	⇒	module
1	module+	⇒	program
2	program+	⇒	package
3	package+	⇒	bundle
	...		

serve that composition at level 0 and 1 is language-specific, whereas composition at the other levels does not depend on the programming language. We can have different names for components and compositions. For example, a composition of applications can also be called a subsystem. The exact names do not really matter. In this article we will use the names as indicated in Table 1.

With a package we mean a directory structure containing the source files of a collection of programs (i.e., a source tree), together with a build process. A bundle is a collection of such packages with a build process that integrates the individual ones. Both notions are further discussed in [6].

At each level there is a different notion of component and a different composition mechanism (or tool). E.g., the module system of a programming language, the RPM package manager, source tree composition, Koala. There are also different mechanisms to deal with dependencies and with variability [4]. There is no uniform component model that captures all levels which makes multi-level composition and variability hard to implement.

3 Multi-level Composition

The different levels of composition that we discussed in the previous section suggests that life does not end after a composition at a particular level. We can make the following observations:

- A composition at level i forms a component at level $i+1$. E.g., modules serve as composite entities at level 0, whereas at level 1 they serve as atomic entities that can be composed to form applications.
- At different levels different component models are used. For instance, the language's module system at level 1.
- Variability is not transparent between composition levels because each level has its specific mechanisms for dealing with variability. E.g., at level 1 we may use function arguments, conditional code, macro's etc., at level 2 we use command-line switches.

In this article we will explore the idea to break the barriers between different component levels by using a single component model to define components and compositions. The benefits are that composition and variability become transparent across composition levels. That is, compositions at level i can be used as composition or as component at level $i+1$ and unbound variability at level i propagates upwards to level $i+1$. We call this multi-level (or vertical) composition and variability. Level-specific composition tools are responsible for performing a composition of level-i components.

A composition performed at level i yields the composition itself and a component definition for level $i+1$. This component definition consists of requires interfaces for functionality that is needed but not contained in the composition, provides interfaces for offered functionality, and variability interfaces for unbound variability parameters. The composition process consists of a level-independent composition process that operates on component definitions, and a level-specific process that performs concrete composition actions for a particular level.

Below we discuss requirements for composition and variability mechanisms.

3.1 Requirements for multi-level composition

To support composition at a particular level it should be possible to map constructs from the component model to that level. To that end, there must be a representation at each level for the following concepts:

Provides interface A component implements the functionality that is declared in one or more provides interfaces. Each level should therefore be able to make this provided functionality available. In C, this is accomplished by means of function definitions.

Requires interface Similarly, there must be a way to represent needed functionality, which is declared in requires interfaces. In C, this is accomplished by function prototypes that are defined as extern.

Interface bindings The task of a composition tool is to bind required functionality to provided functionality. Each level should therefore support the ability to bind a component that needs a function f to a component that implements function f. In C this is typically achieved by the linker.

3.2 Requirements for multi-level variability

Multi-level variability demands that at each level variation points can be expressed, bound, and mapped to a lower level.

Variability interface At each level we need to be able to define what the variation points of a component are.

For instance command-line switches form the configuration interface of an application.

Variability binding Each level should provide a binding mechanism for variation points. For instance, command line switches are a means to bind variation points at application startup time. E.g., `ls -a -l`.

Variability mapping For every level above zero, a binding of a variation point might have to be mapped transitively to the lower level that implements the variation point. For instance, if `ls -a -l` is executed, then the variability bindings `-a` and `-l` at level 2 have to be mapped to level 1, such that the `ls` program operates according to these bindings.

In Section 5 we discuss how these requirements for composition and variability can be met at different levels when we develop an architecture for multi-level composition.

4 Koala Component Architecture

In this section we discuss the Koala component model [11]. We explain its component definition language (CDL), its diversity mechanism, and its mapping to level-1 components implemented in C. In the remainder of this article we will use the Koala CDL for multi-level component composition. Later we show how the mapping to C can be generalized to general level-i components.

Koala is a component model developed by Philips Research and currently used by the product divisions Consumer Electronics (CE) and Semi-Conductors. Koala serves to achieve effective software reuse and to manage diversity in product families.

Koala components are abstractions for C modules. A composition expressed in Koala therefore reduces to a composition of C modules, i.e., a level-1 composition. The Koala compiler produces header files with name bindings and a build process consisting of a Makefile, that knows how to build the product. The following concepts can be identified in the Koala component model:

Functional interface A functional interface in Koala is a named entity that consists of a list of C prototypes and/or definitions. Prototypes can be function or constant prototypes. Since they are prototypes they need to be defined somewhere in the composition. Functions and constants may also be defined by specifying a C expression as body. Koala follows the C syntax for function prototypes and expressions.

```
interface I {
   int c1;
   int f(int x);
   int c2 = 10;
   int g(int y) = y*2;
}
```

Koala supports requires and provides interfaces. The signature of an interface forms its type.

Diversity interface A diversity interface is an interface that captures variation points of a component. It is not a first-class concept of the language, but mimicked by means of interfaces consisting solely of constants or constant prototypes.

Module Modules are atomic entities in Koala. There are two forms of modules: virtual modules and concrete modules. Concrete modules represent C files as they exist in the file system. Virtual modules only exist in the Koala domain and serve to specify bindings for prototypes in interfaces.

Cable Cables serve to connect interfaces to interfaces, modules to interfaces, and interfaces to modules. Only correctly-typed interfaces can be connected. This means that two interfaces have the same type or that the source interface is a sub-type of the target interface.

Component A component groups modules, interfaces, cables, and other components. Hence, a component is not an atomic entity, but a container that serves as abstraction mechanism.

Switch Switches serve functional diversity. During the composition process they create cables between interfaces depending on the evaluation of switch expressions.

Figures 2 and 3 depict a small Koala composition. The first shows the textual representation, consisting of an interface definition *I* and three component definitions. Modules declared as 'present' denote modules with functions that are called outside the Koala domain. For instance, a module containing the 'main' function should be declared as 'present'. Figure 3 depicts the corresponding graphical representation of the composition.

The three modules `m1`, `m2`, and `m3` correspond to C files. The cables between the modules specify the direction of function calls. The interface definition `I` specifies what functions are available. Thus, module `m1` invokes the function `f`, which is implemented by module `m2` and `m3`. For the Koala compiler to be able to generate correct bindings for the C-modules (without name clashes), special name conventions are used. Basically, the convention is that functions are prefixed with the corresponding interface instance name. Thus, in module `m1` the function `f` from interface

```
interface I {
  ATerm f(ATerm x, ATerm y);
}
component Main {
  contains component A a;
           component B b;
           component C c;
  connects
    a.r1 = b.p;
    a.r2 = c.p;
}
component A {
  requires I r1, r2;
  contains module m1 present;
  connects
    m1 = r1;
    m1 = r2;
}
component B {
  provides I p;
  contains module m2;
  connects
    p = m2;
}
component C {
  provides I p;
  contains module m3;
  connects
    p = m3;
}
```

Figure 2. Textual representation of a Koala composition.

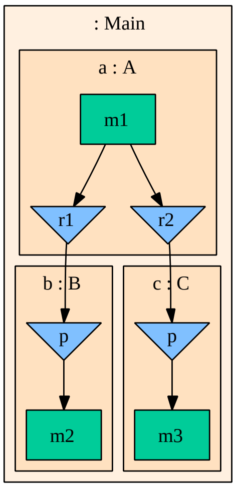

Figure 3. Graphical representation of a Koala composition.

instance `r1` is used as `r1_f`. In module `m2`, the function `f` is called `p_f`. These names are local to the C-module and not externally visible. The Koala compiler synthesizes global names for these functions and generates mappings from local to global names. These mappings are contained in generated header files. For instance, with the mappings `#define r1_f Main_b_p_f` for module `m1` and `#define p_f Main_b_p_f` for module `m2`, the function call `r1_f` is bound to `p_f` of module `m2`. With a similar mapping the function `r1_f` is bound to `p_f` of module `m3`.

In addition to C-code generation the Koala compiler performs several optimizations and creates a Makefile for compiling a product. We refer to [11] for further details.

The Koala architecture that operates on C modules clearly meets the requirements of multi-level composition. Function definitions and function prototypes represent provided interfaces, function calls represent requires interfaces, and the Koala function binding mechanism represents interface bindings.

Koala was not developed for multi-level composition or for producing half-fabricates. As a result, the compiler only generates code for compositions that do not contain unbound variability parameters. Hence, the requirements for multi-level variability are not met by the original Koala compiler. That is, the compiler does neither produce a variability interface, variability bindings, nor variability mappings. As a consequence of this, Koala is restricted to level-1 compositions: the result of a composition is either a single application or a library. For libraries, the compiler does not generate a corresponding component definition, which can be used for higher-level compositions.

5 Generalizing Koala

Koala is a component model that performs level-1 composition for the C programming language. In the remainder of this article we discuss how the Koala composition model can be generalized. The idea is to allow compositions to be created which still have unbound variability parameters (something that is not possible with plain Koala), and to use the same component model to perform compositions at different levels. Unbound variability parameters of level i may get bound at a level above i.

To achieve this, we generalize the notion of a Koala module. It is no longer a representation of a C module, but rather, a representation of an atomic entity at level i. A composition at level i can therefore function as a module at level $i+1$. Furthermore, we create a new Koala compiler consisting of a level-independent frontend and several level-specific backends. The frontend normalizes a Koala composition by transitive wiring, constant propagation, component pruning and more. The result is a normalized specification describing the ingredients of a composition and its (partial-evaluated) configuration. A backend generates from this specification a composition for a particular level.

Level-1 composition is programming language-specific. For instance, in the previous section we saw how level-1 composition is performed by Koala for C. Composition of level 1+ is language independent. As we will see shortly, it seems that multi-level composition converges. That is, from a certain level we can use the same backends for that level and above.

6 Technology

To implement multi-level composition we need a language-specific level-1 backend and language-independent backends for the levels above. In this article we will discuss a level-1 backend for the Stratego programming language [13]. For the language-independent levels we will develop backends that make use of Autoconf/Automake [9, 10] and XTC [14]. Below we will first introduce these technologies, in the upcoming sections we discuss how they serve multi-level composition.

6.1 XTC

XTC [14] is a composition framework for tool components (i.e., components have the form of executable programs). It is based on repositories in which reusable artefacts are registered and on APIs that bind identifiers to artefacts by querying repositories. XTC is developed to provide a flexible solution for run-time composition of tool components.

With XTC it is no longer needed to store the locations of components within an application. Instead, these locations are registered in an XTC repository and can be queried at run-time. The application only references components by an identifier, the XTC API binds such identifiers to component locations and takes care of proper component invocation. Component locations can therefore safely be changed and the exact bindings can be changed at any time.

6.2 Stratego

Stratego is a programming language for developing program transformation systems [14]. Stratego finds its origin in term-rewriting, but makes the way that rewriting is performed programmable. Basically, a Stratego program consists of rules that transform one term into another and strategies that define how these rules have to be applied and how the original term should be traversed (e.g., bottom-up, topdown, etc.).

Stratego programs operate on a uniform data type, called ATerms [3]. It is a very simple but expressive prefix term format. Compared to XML it is simpler but better suited to represent structured data. Because every Stratego program operates on ATerms, they can easily be composed (e.g., in a pipe line) to form complex applications. Moreover, since such programs are only connected through ATerms, programs written in different languages can be part of a composition. This yields a component-based framework where ATerms function as common exchange format leading only to data-coupling between programs [7].

Having ATerms as only data format simplifies our approach because we do not need to deal with typing at different levels. All function arguments, for instance, are ATerms. Consequently, there is no need to marshal data from one level to another.

6.3 Autotools

Autotools is a collection GNU tools for software construction. In this article we will only use the Makefile generator Automake and the configuration script generator Autoconf.[1]

Automake forms an abstraction for low-level Makefiles. It makes build process definitions declarative, describing what needs to be build, not how it should be built. This results in build process definitions which are much shorter and better portable. Autoconf is a configuration script generator that provides a uniform way for static (i.e., compile-time) configuration. Automake and Autoconf work closely together. Together they promote build-level component-based software engineering [8].

7 A Framework for Multi-level Composition

In this section we discuss a framework for multi-level composition. We will show how the technology discussed in the previous section can be used to fulfill the requirements for multi-level composition and variability. Table 2 enumerates the different techniques used at each level.

7.1 Level-1 composition with Stratego and XTC

Level-1 composition is concerned with composing individual Stratego modules and compiling them into executable applications. This composition yields a set of generated Stratego modules with strategy bindings and a Makefile defining how to compile a program from Stratego modules.

Provides/requires interfaces In Stratego, functionality is made available via strategy definitions which have

[1] In [8] we explained that the dependence on Autotools is not strict.

Table 2. Table showing how the different concepts from the Koala component model map to different composition levels.

Concept	*level-1*	*level-2*	*level-3*
Provides interface	Strategy definitions	XTC repository	XTC repository
Requires interface	`imports`	Configuration switch	Configuration switch
Interface binding	Import chasing / Name bindings	XTC import	`AB_CONFIG_PKG`
Variability interface	`ArgOption`	`AC_ARG_WITH`	`AC_ARG_WITH`
Variability binding	Command line switch	Configuration switch	Configuration switch
Variability mapping	`get-config`	XTC binding	`AB_CONFIG_PKG`

the form: `f = <<body>>`, where `<<body>>` contains the implementation of the strategy. A strategy `f` is applied to a sequence of arguments `a1`,`a2`,...,`an` as: `<f>(a1,a2,...,an)`. Strategy definitions are defined in named modules. Such modules can be used in other modules with Stratego's 'imports' construct. Thus, a provides interface can in Stratego be represented as a named module containing strategy definitions, a requires interface as a declaration of an imported module.

We will use a similar convention as in C (see Section 4) to prevent name conflicts. Namely that strategy names are prefixed with the module name and the interface instance name. Furthermore, we require that module names have unique names. For instance, consider Figure 2. The function `f` of interface `I` is defined in module `m2` and `m3` and used in `m1`. A very simple definition in `m2` might look like:

```
m2_p_f = ?(x1,x2); !x1
```

The symbol `?` denotes term matching and assigns the two arguments of `f` to `x1` and `x2`, respectively. The symbol `;` denotes sequential composition of strategies. The symbol `!` denotes term construction. This strategy thus returns its first argument. In module `m3` the strategies might then be used as:

```
... = <m1_r1_f>(x1,x2); <m1_r2_f>(x1,x2)
```

Because of the name convention, the strategy definition and strategy call use different names. These will be bound at composition time (see below).

Binding The Stratego module system assembles a Stratego specification by concatenating all imported modules transitively. It is the task of the level-1 back-end to take over the need to specify imports in Stratego modules because compositions are already defined in Koala. The back-end therefore produces Stratego code containing proper import constructs and name bindings between modules. For instance, in case of Figure 2 bindings between `m1` and the modules `m2` and `m3` are created. For module `m1` the following code is produced:

```
module m1_bindings
imports m2 m3
strategies
   m1_r1_f = m2_p_f
   m1_r2_f = m3_p_f
```

This module contains all needed imports as well as mappings for each function in an required interface to a corresponding implementation. The module `m1` only needs to import this generated module to get access to the strategies it needs.

If a requires interface is not connected (see Figure 4 where `r2` is unconnected), then strategy bindings are generated that make use of the XTC API to invoke functions of the interfaces as level-2 components. For instance, for Figure 4 the following binding is generated:

```
m1_r2_f = xtc-call(!"m1_r2_f")
```

This strategy definition calls the level-2 component that is registered in the XTC repository under the name `m1_r2_f`. During level-2 composition, this identifier `m1_r2_f` will be bound to a concrete level-2 component (for instance to `m3_p_f`).

Each function of an unbound provided interface will compile into a separate program. All such programs are registered in an interface-specific XTC repository to make them available as level-2 components. To that end, the generated Makefile is extended and for each function a Stratego module is generated containing the 'main' strategy. For instance, for `m3` the module `m3_p_f.str` will be generated:

```
module m3_p_f
strategies
   io-m3_p_f = io-wrap(m3_p_f)
```

This module defines the main strategy `io-m3_p_f`. This strategy will be invoked when the program `m3_p_f` is executed. It performs option wrapping (see below) and then calls the strategy `m3_p_f` that implements the function `f` of interface `I`.

Variability interface For unbound variability, we generate command-line option switches and use a run-time configuration

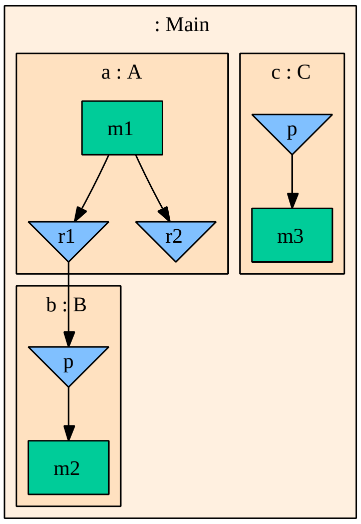

Figure 4. Composition with unconnected provided and requires interfaces.

environment that can be queried with the strategy `get-config` and manipulated with `set-config`.

To create a variability interface, we use Stratego's powerful mechanism for handling command line options. This mechanism is activated by the `io-wrap` strategy which parses command line switches and performs proper actions accordingly. Command-line switches can be defined with the `ArgOption` strategy. A mapping for `var` can be defined as follows:

```
ArgOption(
   "--var",
   <set-config>("var", <id>),
   !"--var v    Bind var to v")
```

This fragment declares the command line switch `--var` and stores the associated value in the run-time environment. It also declares a short usage message that is displayed when the program is executed with the `--help` switch. Multiple switches can be combined using the + operator.

The level-1 backend creates a configuration switch for every member of an unbound variability interface. These are combined with the + operator and passed as argument to the `io-wrap` strategy. E.g.,:

```
io-m3_p_f = io-wrap(options, m3_p_f)
options = ArgOption(
             "--var",
             <set-config>("var", <id>),
             !"--var v    Bind var to v")
          +
          ...
```

Variability binding The set of command line switches forms the level-2 variability interface of the resulting level-2 component. Variability parameters can be bound via the command line. For instance, by executing `m3_p_f --var value`.

Variability mapping The level-1 backend generates for each bound variability parameter a strategy definition in the generated bindings module that returns the corresponding value:

```
var = value-of-var
```

For unbound variability, we generate command-line option switches and strategies that map bindings to the Stratego program level. For instance, for `var` the following mapping is generated:

```
var = <get-config>"var"
```

This strategy queries the run-time configuration environment for a binding of the identifier `var`.

7.2 Level-2 composition with Autotools and XTC

Level-2 composition is concerned with composing executable programs into packages. This form of composition makes use of XTC repositories, Automake, and Autoconf. Level-2 composition consists of binding requires interfaces that remained unbound during level-1 composition to provides interfaces of level-2 components. Similarly, unbound variability parameters of level-1 compositions are bound. If the resulting composition contains unbound provides or requires interfaces they propagate upwards to level-3 interfaces. The level-2 backend produces an Autoconf configuration script and an Automake Makefile. The latter iterates over the Makefiles of the constituent level-2 components.

provides interfaces Like in level-1 composition, unbound provides interfaces at the border of a composition propagate to provides interfaces at the next level (level 3 in this case). For each of these interfaces we create a separate XTC repository. All programs that correspond to such an interface are registered in the repository.

requires interfaces Requires interfaces at level 2 are represented as Autoconf configuration switches. These can be bound at level-3 composition by specifying the location of the XTC repository that represents the interfaces. A configuration switch for a required interface 'IFoo' can be defined as follows:

```
AC_ARG_WITH(
   [IFoo],
   AS_HELP_STRING(...),
   [IFOO=${withval}])
AC_SUBST([IFOO])
```

This fragment declares a configuration switch `--with-IFoo` and sets the variable `IFOO` accordingly. This variable is then substituted in a Makefile, where it can be used to import the XTC repository for IFoo (see below). Also a usage string is specified.

Interface binding Binding a requires interface of a level-2 component to a provides interface (which is represented as an XTC repository) can be accomplished easily with XTC. First, the XTC repository of the provides interface is imported in a newly created repository that represent the requires interface. Then name mappings are registered in the new XTC repository. For instance, for Figure 4 a mapping from `m1_r2_f` to `m3_p_f` is registered.

Variability interface A variability interface of a level-2 composition is represented as an Autoconf configuration interface. It consists of configuration switches for each unbound variability parameter.

Variability binding A variability parameter can be bound by running `configure` (which is generated from an Autoconf configuration script) and specifying variability bindings. For instance, to bind the parameter `foo` to `bar` the tool is invoked as `configure --with-foo=bar`.

Variability mapping A binding of a variability parameter at level 2 is mapped to level 1 using XTC bindings. Remember from Section 7.1, that level-1 components use command-line switches to bind variability parameters. For instance, to bind `foo` to `bar`, a component `f` is invoked as `f --for bar`. Mapping a variability binding from level 2 to level 1 is accomplished by registering the binding in the corresponding XTC repository. That is, together with the tool location, also its variability bindings are registered with XTC.

7.3 Level-3 composition with Autotools and XTC

Level-3 composition assembles packages into bundles. This is the last distinguishing form of composition. Subsequent composition levels (e.g., composition of bundles) can use the same composition backend. Level-3 composition is almost similar to level-2 composition. The backend produces an Autoconf configuration script and a top-level Makefile that invokes the build processes of all bundled packages.

Provides interfaces Like level 2, provides interfaces are represented as XTC repositories. Hence, for every provides interface a new XTC repository is created.

Requires interfaces Like level-2 composition, requires interfaces are represented by Autoconf configuration interfaces. Every requires interface will be represented by a configuration switch.

Interface binding Binding requires interfaces of a level-3 component implies defining bindings for the corresponding configuration switches and executing the component's configuration process. This is accomplished with a special Autoconf macro `AB_CONFIG_PKG`. For instance, to bind the requires interface `IFoo` of component `C` to `bar`, the following code is generated:

```
AB_CONFIG_PKG(
   [C],
   [--with-IFoo=bar]
)
```

When running the Autoconf configuration tool for the bundle, the configuration tool of `C` is also executed and is passed the binding `--with-IFoo=bar`.

Variability interface Level-3 composition makes use of the configuration interface of Autoconf configure scripts, similar to level-2 composition.

Variability binding Also similar to level-2 composition is the binding of variability parameters. This is performed by specifying configuration switches.

Variability mapping Mapping variability bindings to level-2 bindings is accomplished with `AB_CONFIG_PKG`. Like interface binding, this leads to an invocation of the level-2 configuration tool with appropriate configuration switches.

8 Cross-level Composition

With the composition backends for level 1—3, we can make arbitrary static compositions. It turns out that compositions above level 3 can be created with the backends discussed thus far. For instance, creating a composition of bundles is no different than a composition of packages. Observe that backends are tailored for a composition at one specific level. What needs to be done is to allow composition levels to be mixed. That is, to allow that a Koala composition can contain components of arbitrary level. We call this *cross-level* composition.

Cross-level composition works by normalizing a Koala composition to a state where all components have the same level. To reach this situation, the koala compiler is first run on all components with lowest level and the corresponding level-specific backends are executed. This yields new component definitions of the next higher level, as well as component representations on the file system. This process is repeated until all components have the same level. Then, the intended composition can be performed on the remaining components. In order to be able to determine what the level of a component is, each component definition is annotated with its level. The process of cross-level composition is depicted in Figure 5. The picture shows how a level-1 component is first transformed into a level-2 component by running the level-1 composition tool. The resulting composition consists solely of level-2 components. This composition is then performed with a single level-3 component as result.

9 Concluding Remarks

Discussion In this paper we explained that different notions of components exist. Although in practice they are often handled as orthogonal, independent entities, they are in fact related. This forms the motivation for multi-level composition, where compositions at one level may function as components at a level above. This approach has many benefits:

- Components and variability of all components of all levels can be modeled uniformly.

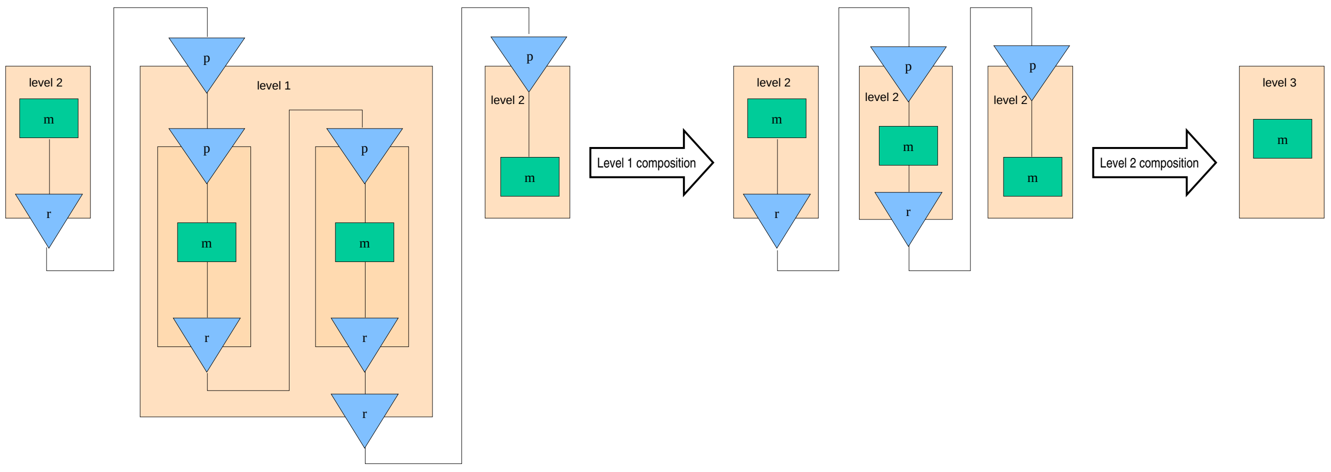

Figure 5. Cross-level composition

- Variability can cross composition levels.
- More control over the composition moment of components.
- Uniform integration of multi-lingual and third-party components.
- Uniform notion of composition, giving rise to different compositions, such as applications, bundles, half-fabricates, etc.

There are also some difficulties. One is that not all variability is truly cross-level. For instance, once compiled, the size of an array can usually no longer be changed. Somehow we need to be able to restrict the moment where a variability parameter can be bound.

Contributions We adopted Koala as uniform component model. It serves to define components and compositions at different levels. Multi-level composition requires that concepts from the component model can be mapped to the level. We identified three essential concepts. We discussed how level-specific backends perform a composition by mapping the composition concepts to a particular level. We developed a framework for multi-level composition, based on XTC as composition framework, Stratego as programming language, and ATerms as uniform exchange format.

Future work There are many directions for future work. These include: i) implementing all parts of our architectue; ii) incorporating other programming languages than Stratego. This requires developing XTC APIs for these languages; iii) performing case studies to check the feasibility of our approach; iv) investigating whether multi-level composition can also be used for dynamic composition; v) investigating whether the dependency on a uniform exchange format (such ATerms) is essential.

References

[1] E. C. Bailey. *Maximum RPM*. Red Hat Software, Inc., 1997.

[2] D. Box. *Essential COM*. Addison-Wesley, 1998.

[3] M. G. J. van den Brand, H. A. d. Jong, P. Klint, and P. A. Olivier. Efficient Annotated Terms. *Software, Practice & Experience*, 30(3):259–291, 2000.

[4] E. Dolstra, G. Florijn, M. de Jonge, and E. Visser. Capturing timeline variability with transparent configuration environments. In *Proceedings: ICSE Workshop on Software Variability Management*. ACM, May 2003.

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

[6] 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, Apr. 2002.

[7] 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, Jan. 2003.

[8] M. de Jonge. Decoupling source trees into build-level components. In J. Bosch and C. Krueger, editors, *Proceedings: Eighth International Conference on Software Reuse*, volume 3107 of *Lecture Notes in Computer Science*, pages 215–231. Springer-Verlag, July 2004.

[9] D. Mackenzie, B. Elliston, and A. Demaille. *Autoconf: creating automatic configuration scripts*. Free Software Foundation, 2002. Available at `http://www.gnu.org/software/autoconf`.

[10] D. Mackenzie and T. Tromey. *GNU Automake Manual*. Free Software Foundation, 2003. Available at `http://www.gnu.org/software/automake`.

[11] 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, Mar. 2000.

[12] C. Szyperski. *Component Software: Beyond Object-Orientated Programming*. Addison-Wesley, second edition, 2002.

[13] 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.

[14] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specifi c Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.