Variability management with feature models

Danilo Beuche
University Magdeburg
Universitätsplatz 2
D-39106 Magdeburg
danilo@ivs.cs.uni-magdeburg.de

Holger Papajewski
pure-systems GmbH
Agnetenstr. 14
D-39106 Magdeburg
holger.papajewski@pure-systems.com

Wolfgang Schröder-Preikschat
University Erlangen
Martenstr. 1
D-91058 Erlangen
wolfgang.schroeder-preikschat@informatik.uni-erlangen.de

Abstract

Variability management in software systems requires adequate tool support to cope with the ever increasing complexity of software systems. The paper presents a tool chain which can be used for variability management within almost all software development processes. The presented tools use extended feature models as the main model to describe variability and commonality, and provide user changeable customization of the software artifacts to be managed.

1 Introduction

While the development of single-system software is not a completely understood process yet, the need to develop sets of related software systems in parallel already exists and increases. The growing interest in concepts like software product lines and software families by industry and research groups substantiate this need. The first ideas and solution proposals of software families go back a long time in terms of computer science history. Widely known are the works of Parnas [17], Habermann [10] and Neighbors [16] from the 70s and early 80s. However, most of the work was done in the 90s, especially in the second half. Much of this work was related to organizational aspects, i.e. how to make developers in an organization efficiently develop software so that it can be used in several different products instead of just in a single one. Methods like ODM [19], FAST [22] or PuLSE mainly focus on this topic. The more technical aspects of the implementation of such systems are mostly left open in these approaches. Yet there are several techniques which cover these aspects. Examples are (static) meta-programming [6], GenVoca [3] and many others.

Common to all methods is that they use models to represent the differences and commonalities between the various resulting products or implementation fragments. The first model is a result of the domain analysis process and the latter the result of the domain design and implementation process. However, in most cases tool support for the transition from the high-level models of the domain analysis process to the product line implementation is missing. Some of the methods (e.g. FAST) propose the use of generators which accept a problem domain specific language as input and generate the implementations according to the input specification. However, even with generator-generators like in GenVoca this process is not easy and often too heavyweight for many software development projects.

In this paper we present a set of models and related tools that can be used in conjunction with almost any product line process that uses feature models[1] as representation for commonalities and variabilities. The goal was to develop a complete tool supported chain of variability management techniques which cover all phases from domain analysis to the deployment of the developed software in applications (products).

This paper is structured as follows: Section 2 discusses some problems of variability management and tool support. Section 3 introduces the basic concept of the approach. A more detailed explanation of some aspects of this approach is given in the fourth section. Section 5 demonstrates the extensibility of the approach using a case study. A brief introduction of CONSUL based tools is presented in the sixth

[1] Or any model which can be transformed into a feature model

section. Section 7 discusses some related work. The last section contains some concluding remarks and gives an outlook on future work.

2 Rationale for an open variability management tool chain

The definition of software variability as given in the workshop's CfP is:

> "Software variability is the ability of a software system or artifact to be changed, customized or configured for use in a particular context."

This definition is very open and broad. The openness is a key point. Variability management is a cross-cutting problem, which affects almost all more complex software projects to various degrees.

Variability in software systems can be found in the functional and non-functional attributes of the systems. Functional variability means that the system can provide different functionalities in different contexts. E.g. a variable HTML viewer component supports the configuration of the sets of HTML dialects it is able to render. Non-functional variability includes system properties such as memory consumption, execution speed or QoS of system functionalities.

These different aspects of variability can be realized in many different ways. The following list is an attempt to categorize where and how variability is expressed:

Programming language level: the variability is expressed using the programming language which is used to implement the system, for instance Java, C++ or C. This involves language features like conditional execution, function parameters and constants. Some of the variability is resolved at compile time[2], the remaining variability is resolved at runtime.

Meta language level: a meta language is used to describe variable aspects of the software artifacts. Examples are aspect oriented languages like AspectJ or AspectC++, meta programming systems like COMPOST [1], or BETA [15]. Even the C/C++ preprocessor language is an albeit simple example but nevertheless probably most widely known meta language for variability representation. The binding time of variability depends on the language concepts. In most cases the actual result of the binding is expressed in a basic (non-meta) programming language, which is then compiled or executed.

[2] If the compiler is able to optimize the resulting code based on partial evaluation, i.e. replacement of constant expressions with theirs results etc.

Transformation process level: almost every software is transformed from higher level language(s) into an executing system through several steps of transformations. For instance a C program is compiled by a compiler into an intermediate representation (.o files) which in turn is linked against a set of libraries by the linker, and is finally loaded into the memory of a particular computer system by the operating system's program loader. Most of the involved transformation tools can be parameterized so that the resulting system changes. I.e. the compiler has several levels of optimization, which may influences the memory footprint and/or execution speed of the compiled system. The transformation process is usually controlled by a tool like make [20] or ant [2] that interprets a transformation process description.

In most software systems, several levels of variability expressions are used together or independently. The small example shown in Figure 1 demonstrates such a mix of levels. It shows a small C source file and a makefile which is used to produce two different executables from the same source code. The point of variability is the second argument of the `printf` function. The preprocessor macro defines this value if the value of `HW_TEXT` is not already set by other means. The makefile includes two different transformation rules for the same source, the second uses a compile option to set the value of `HW_TEXT`.

```
#include <stdio.h>

#ifndef HW_TEXT
#define HW_TEXT "Hello, world!"
#endif

int main(int argc, char* argv[])
{
  printf("%s\n",HW_TEXT);
}
```

```
all: hw_en hw_de
hw_en: hw.c
   $(CC) -o $@ $<
hw_de: hw.c
   $(CC) -o $@ \
   "-DHW_TEXT=\"Hallo, Welt!\"" $<
```

Figure 1. A very simple example of variability management with C and make

In most cases, such a mixing of levels is needed to accomplish the goals of the software development in terms of efficiency, organization, reuse etc. However, tool support for controlling these highly complex mixes is very limited. Especially an automated coupling of high level models

of variability and commonalities (VC) with the "low-level" implementations of the variability is rarely to be found.

Several important issues have to be considered when developing a tool chain to support the complete process of variability management:

- Easy, yet universal model(s) for expressing variability and commonalities should be supported.
- Variability at all levels must be manageable.
- Introduction of new variability expression techniques should be possible and easy.

The CONSUL (CONfiguration SUpport Library) tool chain presented in the next section tries to meet all these requirements.

3 CONSUL overview

The CONSUL tool chain has been designed for development and deployment of software program families. The core of CONSUL are the different models which are used to represent the problem domain of the family, the solution domain(s) and finally to specify the requirements for a specific representative (member) of the family.

The central role is played by *feature models* which are used to represent the problem domain in terms of commonalities and variabilities. CONSUL uses an enhanced version of feature models compared to the original feature models as proposed in the FODA method [12]. A detailed description of those enhancements is given in Section 3.1.

The solution domain(s) (i.e. the implementations) are described using the *CONSUL Component Family Model* (CCFM). It allows to describe the mapping of user requirements onto variable component implementations, i.e. the customization of a set of components for a particular context. As the name suggests, this model has been newly developed for CONSUL. The CCFM is presented in detail in Section 3.2.

The *feature sets* are used at deployment time and describe a particular context in terms of features and associated feature values.

Figure 2 illustrates the basic process of customization with CONSUL. Most steps can be performed automatically once the various models have been created. The developers of variable components have to provide the feature models, the component family models, and the implementations itself. A user[3] provides the required features, the tools analyze the various models and generate the customized component(s).

[3]Here a user can be either human or also a tool which is able to derive the set of required features automatically from some input

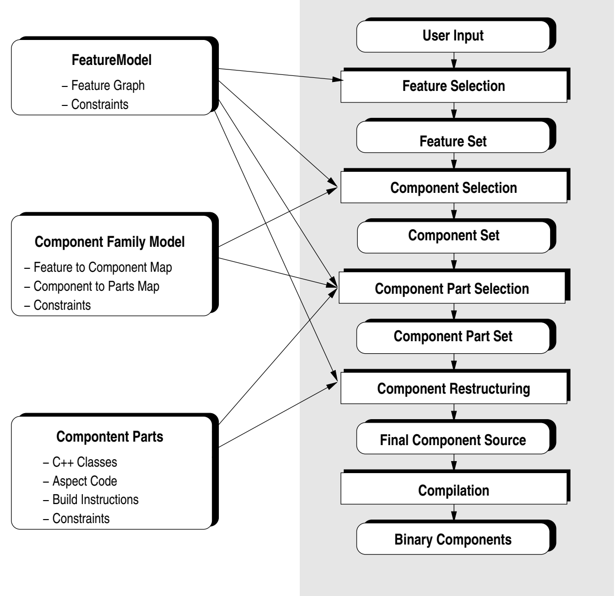

Figure 2. Overview of CONSUL process

The key difference between CONSUL and other similar approaches is, that CONSUL models in most cases only describe *what* has to be done, but not *how* it should be done. CONSUL provides only basic mechanisms which can be extend according to the needs of the CONSUL user. This flexibility is achieved by combining two powerful languages inside CONSUL and allowing the user to extend this system.

The first language is Prolog, a widely known language for logic programming. Prolog is used for constraint checking, i.e. for expressing relations between different features. The same logic engine is used for component selection and customization.

The second language is a XML-based language called XMLTrans which allows to describe the way customization (transformation) actions are to be executed. The most simple transformation is the verbatim inclusion of a file into the final customized source set. Even for this simple transformation different solutions are possible. On systems where file system links are possible, the inclusion action can be described differently in a different way than on systems without such file system capabilities. XMLTrans allows the tool users to describe similar and more complex transformations in a special XML language. Due to its modular structure, it can be extended with user supplied transformation modules. This can be used to provide seamless access to special generators or other tools seamlessly from within the tool chain.

3.1 CONSUL feature models

Feature modeling is a relatively simple approach for modeling the capabilities of a software system introduced by Kang et al. [12]. A feature model represents the commonalities and variabilities of the domain. A feature in FODA[4] is defined as an *end-user visible characteristic of a system*.

CONSUL uses feature models because on one hand they are easy to understand, but on the other hand are able to express relatively complex relations in a very compact manner. To enable modeling of more complex scenarios, CONSUL uses a slightly enhanced version of feature models compared to the original concept. The enhanced versions allows to attach typed values to features to represent non-boolean feature informations and additional relation rules called restrictions.

Features are organized in form of *feature models*. A feature model of a domain consists of the following items:

Feature description: each feature description in turn consists of a feature definition and a rationale.

The definition explains which characteristic of the domain is described by the feature, so that an end-user is able to understand what this feature is about. This definition may be given as informal text only or in a defined structure with predefined fields and values for some information like the binding of the feature, i.e. the time a feature is introduced in the system (configuration time, compile time, etc.).

The rationale gives an explanation when to choose a feature, or when not to choose it.

Feature value: each feature can have an attached type/value pair. This allows to describe non-boolean features more easily.[5]

Feature relations: the feature relations define valid selections of features in a domain. The main representation of these relations is the *feature diagram*. Such a diagram is a directed acyclic graph where the nodes are features and the connections between features indicate whether they are optional, alternative or mandatory. Table 1 gives an explanation of these terms and shows its representation in feature diagrams.[6] Additional relations can be attached to a feature. CONSUL provides a flexible mechanism called *restrictions* to enable the description of arbitrary feature relations.

[4]Feature-oriented Domain Analysis

[5]Typed features with values are not part of the original feature model proposal. However, this extension is required to describe many domains and has been proven to be very useful.

[6]The graphical notation differs from the original FODA style to allow easier drawing/generation of feature diagrams.

Feature Type	Graphical Representation
mandatory Mandatory feature B has to be included if its parent feature A is selected	A, B
optional Optional feature B may be included if its parent feature A is selected.	A, B
alternative Alternative features are organized in *alternative groups*. Exactly one feature of such the group B,C,D has to be selected if the group's parent feature A is selected.	A, B, C, D
or Or features are organized in *or groups*. At least one feature of such the group B,C,D has to be selected if the group's parent feature A is selected.	A, B, C, D

Table 1. Explanation of feature diagram elements

From the characteristics of the problem, a domain analyst derives the features relevant for the problem domain.

For example for a domain which requires a variable realization of cosine calculation functions for embedded real-time applications, the model could contain a feature that allows to specify the precision required for the results (`Precision`)[7], a feature that represents whether discrete angle values are used (`ValueDistribution`), a feature to express that fixed calculation time is required (`FixedTime`) and so on. The complete feature model is shown in Figure 3. A more detailed discussion of this example can be found in [5].

The feature model of a problem domain (in our case the cosine world) can be used by an application engineer, and she or he should be able to select the feature the application requires and if necessary to specify feature values.

[7]The names in parentheses are the feature names used in the resulting feature model, see figure 3.

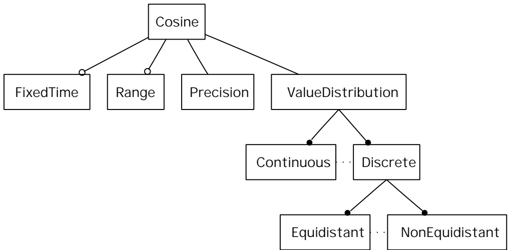

Figure 3. Feature model of cosine domain

3.2 CONSUL component family model

The component family model of CONSUL is not yet another component model in the spirit of CORBA or COM component models. CONSUL uses a very open definition of components. A component encapsulates a configurable set of functionalities. As a consequence, CONSUL cannot check interfaces of connected components itself, but allows to introduce user-definable checks appropriate for the intended framework/architecture. Figure 4 illustrates the hierarchical structure of the component based family model supported by CONSUL.

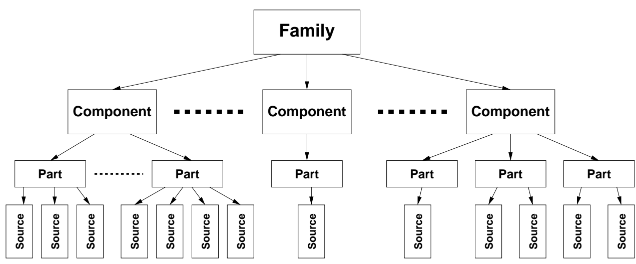

Figure 4. Structure of the CONSUL family models

This approach is reflected in the CONSUL family description language (CFDL) which mainly describes the internal component structure of a family and its configuration dependencies. The language is complementary to languages like OMG's CORBA IDL or Microsoft's COM IDL which focus on the external view of a component. The external interface of a component is merely another (possibly) configurable part of a component for CONSUL.

An small example of the language is given in Figure 5. It shows a simple component realizing the cosine example domain with just three different implementation files. Depending on the selected features one of the `cosine_?.cc` is used to implement the cosine function.

The CONSUL family model represents a family as a set of related components. The inter-component relation of these components is not fixed. I.e. both hierarchical component structures like the OpenComponent model [8] or ordinary independent components can be part of a family model. The CONSUL family description language (CFDL) is the textual representation of the model.

The following paragraphs briefly introduce the three elements of the CONSUL family model.

Components: a component is a named entity. Each component is hierarchically structured in *parts* which in turn consist of *sources*.

Parts: parts are named and typed entities. Each part belongs to exactly one component and contains any number of *sources*.

A part can be an element of a programming language like a class or an object, but also any other key element of the inner and external structure of an component, i.e. an interface description. CONSUL provides a number of predefined part types, like `class`, `object`, `flag`, `classalias` or `variable`. The introduction of new part types according to the needs of the tool users is also possible.

Section 4 gives a small demonstration of this. Table 2 gives a short description of the currently available part types in the current CFDL version.

Sources: a part as a logical element needs some physical representation(s) which are described by the *sources*. A source element is an unnamed but typed entity. The type is used by the transformation backends to determine the way to generate the source code for the specified element. Different predefined types of source elements are supported, like the `file` which simply copies a file from one place into the specified destination of the component's source code. Some source elements are more sophisticated, like `classalias` or `flagfile`, and require generation of new source code by the backends. Table 3 lists the currently available source element representations.

The actual interpretation of these source elements is handed over to the CONSUL component generator backends. To enable the introduction of custom source elements and generator rules, CONSUL allows to plug in different generators. At the moment, two different generators exist. One is implemented in Prolog and operates directly on the Prolog CONSUL knowledge database representation. The second which uses a modular transformation based approach.

The advantage of the Prolog based approach is its speed and the ability to use the power of Prolog everywhere. However, it requires a decent knowledge of Prolog to change or add source element generators. The other approach [18] uses XML to describe the transformations and allows users

```
Component("Cosine")
{
  Description("Efficient cosine implementations")
  Parts {
     function("Cosine") {
       Sources {
         file("include", "cosine.h",def)

         file("src", "cosine_1.cc",impl) {
           Restrictions { Prolog("not(has_feature('FixedTime',_NT))")}}

         file("src", "cosine_2.cc",impl) {
           Restrictions { Prolog("has_feature('FixedTime',_NT),
                                  has_feature('NonEquidistant',_NT")}}

         file("src", "cosine_3.cc",impl) {
           Restrictions { Prolog("has_feature('FixedTime',_NT),
                                  has_feature('Equidistant',_NT")}}
       }
     }
  }
  Restrictions { Prolog("has_feature('Cosine',_NT)") }
}
```

Figure 5. (Simplified) component description for cosine component

to integrate own special-purpose modules into the systems via an easy-to-use module concept. This enables users to introduce their own family specific generators without any need to change the core CONSUL tools.

Using restrictions in CFDL a key difference of the CFDL from other component description languages is the support for flexible rules for inclusion of components, parts and sources. Inclusion constraints, called restrictions, can be attached to each CFDL element.

Each element may have any number of restrictions. At least one of them has to be true to include the element into the system. If there is no restriction specified an element is always included. The CFDL itself does not specify a language for restriction description, it passes the restriction description to an external module. Currently, there is just one language model which uses Prolog as description language and allows direct access to the CONSUL knowledge database[8].

The code of restrictions can access the complete CONSUL model set (feature model, component model, feature set) to make a decision. This allows the customization of components according to the specified needs of the applications on a structural base. In combination with the ability of the backend transformation to produce specialized source elements based on arbitrary parameters and structural informations, this permits almost any customization concept to be used in conjunction with CONSUL.

[8] Although this direct access is very powerful, it has its drawbacks, since it is very easy to make mistakes in Prolog statements, without breaking the syntax. For most statements, an easier, more problem-oriented language would be sufficient. It will be included in a new release of the CFDL.

4 Closing the gap: family variation vs. family member flexibility

One of the main problems of family based software designs is that there are two levels of flexibility or variation in the design. On the one hand there is the "usual" flexibility a family member or a single application has to provide and on the other hand there is the variation inside the family to provide different family members. Both levels cannot be completely separated in a design, often the same design can represent both, family variation and member flexibility.

The following example will illustrate this problem and give an idea how CONSUL can be used to deal with it.

A very important service of any operating system is to provide access to the hardware connected to the processor. Depending on the hardware configuration and/or the needs of the software the operating system has to provide software components and interfaces to different sets of devices. Even if there is a hard disk controller device available in a system, if the software does not require disk access, a disk driver does not have to be included in the system.

The example is based on a fictitious hardware which has three different types of analog/digital converters (ADC) available. The goal is to provide a software design and implementation which adapts easily to different hardware configurations without having to implement different versions of the device drivers. The scalability shall be achieved by

Part Type	Description
`interface` (X)	represents an external component interface X.
`class` (X)	represents a class X with its interface(s), attributes and source code.
`object` (X)	represents an object X.
`classalias` (X)	represents a type-based variation point in a component. A classalias is an abstract type name which is bound to a concrete class during configuration.
`flag` (X)	represents a configuration decision. X is bound to a concrete value during configuration. Depending on the physical representation chosen for the flag, it can be represented as a makefile variable, a variable inside a class or even a preprocessor flag.
`variable` (X)	similar to a flag, but a variable should not be used for configuration purposes.
`project`(X)	represents anything which cannot be described by the part types given above.

Table 2. Overview of CFDL part types

Source element	Description
`file`	represents a file which is used unmodified.
`flagfile`	represents a C++ preprocessor flag.
`makefile`	represents a makefile variable.
`classalias`	represents a C++ typedef variable.

Table 3. Overview of CFDL source element representations

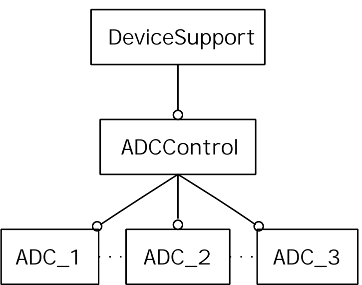

Figure 6. Partial feature model for the ADC example

using the services of CONSUL.

Figure 6 shows the relevant part of the feature model. When ADCSupport is selected, any combination of support for the three different ADC types can be requested. Thus there are seven (three single, three double, one triple) combinations of functional support for ADCs possible. In some application it is known in advance which ADC(s) are going to be used, so compile-time binding should be possible. But there could be applications which will bind an ADC at load-time, and some will defer the decision until run-time and may request access to different ADC over the time.

The drivers shall be realized within a single component. All ADC must provide the same interface to enable switching between different ADCs.

This setting seems to be a classical example for the use of an abstract base class, defining the common interface and three different subclasses which are the concrete realizations of the interface. However, in many configurations, as shown in Figure 7, the base class is not necessary since there is only one class derived from it in use. While the use of abstract base classes is appropriate for modeling and communicating interfaces to users and developers, it requires additional resources during runtime. To implement the runtime variability, C++ as well as other object-oriented languages rely on tables associated with each object derived from abstract base classes. Each table stores the location of the method implementations for the common interface of the abstract base. In C++ these tables are usually called *virtual method tables*. Use of such tables consumes memory for storing the table, and run-time since for each call to an abstract method the corresponding table is consulted.

The measurements for an abstract/concrete class pair

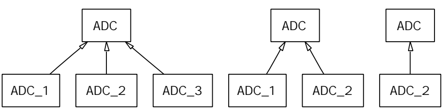

Figure 7. Class hierarchies for 3 different members

with just one virtual method (see Table 4[9]) clearly show that there is an increased memory use for the abstract class version. Especially critical is the use of data memory. Without virtual methods, no data memory is used. Many embedded microcontrollers have separate code and data memories, and often the data memory is quite small (few bytes to some kBytes) so wasting a few dozen bytes of data memory can be a real problem. A skilled embedded programmer would avoid using virtual method whenever possible[10]. To achieve the same resource usage as a hand-coded solution, the variable implementation of drive component should avoid using virtual methods whenever possible.

Hierarchy	Processor	Code	Data
non-virtual	x86	32	0
virtual	x86	206	140
non-virtual	AVR90Sxxxx	80	0
virtual	AVR90Sxxxx	284	42

Table 4. Memory consumption of abstract and non-abstract classes

To solve this problem the `classalias` of CONSUL can be used. The `classalias` part type allows description of flexible, statically changeable class relations. Figure 8 shows a new class hierarchy where the external component interface `ADC` can be mapped to any of the `ADC_?` classes.

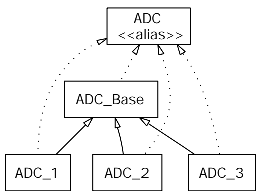

Figure 8. Variable class hierarchy for ADC component

The corresponding component description is shown in Figure 9. The concrete class to which the alias should be set is determined by the four `Value` statements given inside the classalias definition. The evaluation of the second argument of each statement is done top-down. The first argument of the first statement which evaluates to true is used to calculate the class name. In the example, one of the predefined clauses of CONSUL is used. The clause `is_single(X,_NT)` is true when only feature `X` is selected from its corresponding or-feature group. The last statement ensures that if there is more than one feature selected from the group, the abstract base class is used.

To solve the problem of having an abstract base class or not for the `ADC_{1,2,3}` class, the class `ADC_Base` has two different declarations, one as abstract class, and the other as just an empty class definition.

The description of class `ADC_1` is straightforward, it is included in the component whenever support for `ADC_1` is requested. For the other two classes, the descriptions look alike.

It is obvious that the mechanisms for variability used in this example could be used without CONSUL. Changing a class hierarchy could be accomplished using a conditional `#include` resolved by the C++ preprocessor according to a compiler argument which is defined in a makefile. However, with the CONSUL and the CFDL there is one single place to manage the customization process. The information what and how to configure is not spread out over different files in different languages. CONSUL and CFDL separate the structure of systems and components from the source files they are implemented in.

Using AOP to do the trick: the extensibility of the CFDL through its customizable backend makes the introduction of new high-level description elements very easy. Going back to the example given above, there has been some tricking around with the base class of `ADC_{1,2,3}`. It was necessary to provide a fake (empty) base class when the abstract base class should not be used.

The aspect language AspectC++ [7] allows to write aspects for the C++ language which are able to introduce new base classes to arbitrary classes. The use of that feature makes the solution for the ADC example much easier, if the CONSUL would allow a statement to set the base class similar to a class alias.

To make this available in the CFDL, it is necessary to define a new part source type named `baseclass` which takes two arguments, the name of the intended base class and the privilege level (`private, public, protected` for C++).

The addition of a new source element requires only the addition of a new transformation rule to the CONSUL generator backend library. When the XML based backend is used, this requires writing an XML transformation description. With the Prolog backend, the same can be accomplished with appropriate Prolog rules.

Figure 10 shows the modified component description and Figure 11 the generated aspect code.

Using this extension mechanisms, CONSUL can be used to control and combine arbitrarily complex tools to produce the intended customized system. It can be even used to im-

[9] Compiler: gcc 2.96 for x86, gcc 2.95.2 for avr, size values in bytes

[10] Today, most programmers avoid this problem by not even using object-oriented languages for embedded systems programming

```
Component("ADCControl")
{
  Description("ADC Controller Access")
  Parts {
     classalias("ADC") {
       Sources {
         classaliasfile("include", "ADC.h","ADC") }
       Value("ADC_1",Prolog("is_single('ADC_1',_NT)"))
       Value("ADC_2",Prolog("is_single('ADC_2',_NT)"))
       Value("ADC_3",Prolog("is_single('ADC_3',_NT)"))
       Value("ADC_Base",Prolog("true"))
     }
     class("ADC_Base") {
       Sources {
         file("include", "ADC_Base.h",def,"include/ADC_Base_virtual.h") {
           Restrictions {
     Prolog("not(selection_count(['ADC_1','ADC_2','ADC_3'],1,_NT))")
           }
         file("include", "ADC_Base.h",def,"include/ADC_Base_empty.h") {
           Restrictions {
     Prolog("selection_count(['ADC_1','ADC_2','ADC_3'],1,_NT)")
         } } } }
     class("ADC_1") {
       Sources {
         file("include", "ADC_1.h",def)
         file("src", "ADC_1.cc",impl)
           { Restrictions { Prolog("has_feature('ADC_1',_NT)") } }
       } }
     ...
    } }
  Restrictions { Prolog("has_feature('ADCControl',_NT)") }
}
```

Figure 9. CFDL for ADC component

```
aspect consul_ADC_1_ADC_Base {
  advice classes("ADC_1"):
     baseclass("public ADC_Base");
};
```

Figure 11. Aspect code generated for the CFDL `baseclass` source element

plement simple source code generators directly, as shown above.

5 CONSUL case study: Pure

To evaluate the CONSUL ideas, it was necessary to use it in a larger project. The Pure operating system family for deeply embedded systems [4] developed at the University Magdeburg, was an ideal target.

The Pure operating system family consists of about 321 classes implemented in some 990 files. Pure runs on nine different processor types from 8 bit to 64 bit processors and is almost entirely written in C++. Prior to the use of CONSUL, the configuration was done by modifying/setting several C++ preprocessor `#define` statements (about 64) and also some makefile variables. Due to its application area Pure is trimmed to use hardware resource as efficiently as possible. For every application it tries to provide exactly the features an application needs, not more.

The result of the domain modeling using feature models was a model of the PURE problem domain with some 250 features. The model allows approx. 2^{105} different valid feature combinations. The component family model representing the implementation consists of 57 components.

A feature set for a typical configuration has some 20 features. The smallest possible set contains just three features (describing the used compiler, the target cpu model and the target hardware platform), selecting 20 classes. A typical configuration supporting preemptive multitasking with time slices has 94 classes[11]

Using CONSUL reduced the risk of misconfiguration, because the feature model and the CFDL allows to express dependencies and these can be checked automatically. Prior

[11]Both configurations are for a x86 PC based target platform and the GNU Compiler, values for other target platforms may differ slightly.

```
Component("ADCControl")
{
  Description("ADC Controller Access")
  Parts {
    ....
     class("ADC_Base") {
       Sources {
         file("include", "ADC_Base.h",def,"include/ADC_Base_virtual.h") {
           Restrictions {
     Prolog("not(selection_count(['ADC_1','ADC_2','ADC_3'],1,_NT))")
           } } } }
     class("ADC_1") {
       Sources {
         file("include", "ADC_1.h",def)
         file("src", "ADC_1.cc",impl)
 // introduce new base class when not single
 baseclass("ADC_Base","public")
           { Restrictions { Prolog("not(is_single('ADC_1',_NT))") } }
       } }
     ...
    } }
  Restrictions { Prolog("has_feature('ADCControl',_NT)") }
}
```

Figure 10. CFDL for ADC component using the baseclass() source element

to the availability of CONSUL tools for Pure configuration most Pure developers used only two or three well known configurations, because finding a new working configuration was very complicated. Today, the test directory contains some 120 different base configurations. A new working configuration is typically created in a few minutes.

6 CONSUL based tools

Variability management tools have to be used by two different classes of users. The first class is formed by the developers who develop variable software artifacts, the second class by the deployers of these variable artifacts. As a complete tool chain, CONSUL supports both classes.

The modular implementation of CONSUL allows flexible combination of the required services and user interfaces to build different tools. The current application family consists of following three different tools:

Consul@GUI The main application for developers is `Consul@GUI`. Consul@GUI is an interactive modeling tool for CONSUL models. It allows to create and edit the models but can also be used in the deployment of the developed software for generating the customized software.

Figure 12 shows a screenshot of a configuration session. It shows the feature model for the cosine domain with several features selected. The configuration is not valid, since there is still an open alternative. This is indicated by the different background colors of the two features `Equidistant` and `NonEquidistant`.

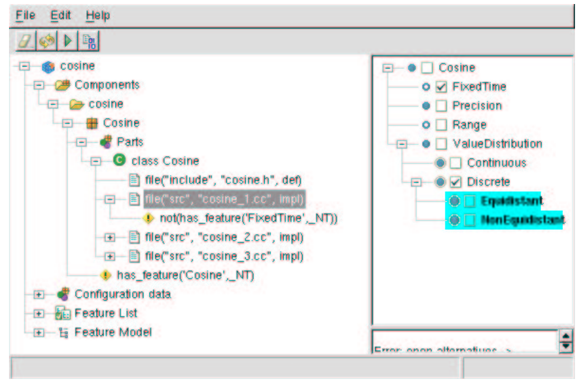

Figure 12. Consul@GUI

Once a valid configuration has been found, the generation process can be started.

Consul@CLI Based on CONSUL a customization tool with a command line interface has been built as well. This tool can be used e.g. together with make to provide automated customization when (re)building a software system.

Consul@Web It is also possible to make software customization available via web browsers. A demonstration based on a Java applet can be found at `http://www.`

pure-systems.com/consulat/. It allows the configuration, building and downloading of Pure via an Java-enabled web browser.

7 Related works

There are not many tools for language-independent, cross-level management of software variability available. The company BigLever with their product GEARS [14] is one of the few. GEARS operates on the file system level to manage variability. It allows to specify conditions for the inclusion of a specific file into a resulting system. However, there is no complete domain model, but several independent sets of parameters are used to describe the conditions. Although this might enhance the reusability, this restricts the description of cross-component dependencies.

Several other approaches use feature models for domain modeling [9, 13]. However, most of them do not use an explicit feature modeling tool which effectively limits the size of the models. In [21] a tool is described which operates on a feature model and is able to generate java class skeletons from feature models.

The transformation process in CONSUL, which produces the customized implementation from component descriptions has some similarities to frame-based source generators like COMPOST [1] or XVCL [11]. The idea of frames blends perfectly with the ideas of CONSUL. The open model of the CONSUL tools allows the integration of such a generator into the transformation process, and the parameterization of the generator is controlled vi the feature model and the component family model constraints.

8 Conclusions

This paper presented an extensible tool chain for variability management. The main model types are an enhanced feature model and a flexible component based family model which enable language independent representation of variability in software systems.

Compared to other tools for variability management CONSUL is more flexible through its extension mechanisms. The use of feature models as the model for communication between the developers of variable software and the deployers has been proven to be an effective solution.

One of the problems of CONSUL is that Prolog is not very well suited as a description language for users. Its syntax rules are to weak to detect typical typos in user defined rules, and the Prolog language system tends too produce very unpredictable results in these cases. A new language for expressing the basic restrictions is in development and will replace the use of "native" Prolog in many places.

Among the future projects based on CONSUL are an integration of CONSUL technology into integrated development environments like Eclipse or VisualStudio. To enhance interoperability with other tools the component family model will be mapped to an XMI representation, allowing direct use inside UML tools like Rational Rose or ARGO/UML.

References

[1] U. Aßmann. Beyond Generic Component Parameters. In J. Bishop, editor, *Proc. of the Component Deployment IFIP/ACM Working Conference*, volume 2370 of *Lecture Notes in Computer Science*, pages 141–154, Berlin, Germany, June 2002. Springer.

[2] Ant Project Homepage. see http://jakarta.apache.org/ant/.

[3] D. Batory, J. Thomas, and M. Sirkin. Reengineering a Complex Application Using a Scalable Data Structure Compiler. In *Proceedings of ACM SIGSOFT*, 1994.

[4] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The Pure Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *IEEE Proceedings ISORC'99*, 1999.

[5] D. Beuche, O. Spinczyk, and W. Schröder-Preikschat. Fine-grain Application Specific Customization for Embedded Software. In *Proceedings of the International IFIP TC10 Workshop on Distributed and Parallel Embedded Systems (DIPES 2002)*, Montreal, Canada, Aug. 2002. Kluwer Academic Publishers.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, 2000. ISBN 0-201-30977-7.

[7] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. AspectC++: Language Proposal and Prototype Implementation. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, Florida, Oct. 2001.

[8] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. Open Components. In *Proc. of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, Tampa, Florida, Oct. 2001.

[9] M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating Feature Modeling with the RSEB. In *Proc. of the 5th International Conference on Software Reuse*, pages 76–85, Victoria, Canada, June 1998.

[10] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[11] S. Jarzabek and H. Zhang. XML-based Method and Tool for Handling Variant Requirements in Domain Models. In *Proc. of 5th IEEE International Symposium on Requirements Engineering RE01*, pages 116–173, Toronto, Canada, Aug. 2001. IEEE Press.

[12] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, Nov. 1990.

[13] K. C. Kang, K. Lee, J. Lee, and S. Kim. Feature Oriented Product Lines Software Engineering Principles. In *Domain*

Oriented Systems Development — Practices and Perspectives, UK, 2002. Gordon Breach Science Publishers. to appear.

[14] C. Krueger. Variation Management for Software Production Lines. In *Proc. of the 2nd International Software Product Line Conference*, volume 2379 of *LNCS*, pages 37–48, San Diego, USA, Aug. 2002. ACM Press. ISBN 3-540-43985-4.

[15] M. Löfgren, J. L. Knudsen, B. Magnusson, and O. L. Madsen. *Object-Oriented Environments - The Mjolner Approach*. Prentice-Hall, 1994.

[16] J. M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, 10(5):564–573, Sept. 1984.

[17] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-5(2):1–9, 1976.

[18] S. Roemke. XML-Based Modular Transformation System. Master's thesis, Computer Science Faculty, University Magdeburg, Magdeburg, Germany, 2002. In German.

[19] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang. STARS Organizational Domain Modeling (ODM) Version 2.0. Technical report, Lockheed Martin Tactical Defense Systems, Manassas, VA, USA, 1996.

[20] R. Stallman and R. McGrath. *GNU Make Documentation Version 0.51 for make Version 3.75 Beta*, May 1996.

[21] A. van Deursen and P. Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, pages 1–17, 2002.

[22] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Approach*. Addison-Wesley, 1999. ISBN 0-201-69438-7.