# Configurable Designs

*Anssi Karhinen, Alexander Ran, Tapio Tallgren*

*Software Technology Laboratory*
*Nokia Research Center*

*5 November, 1996*

*Abstract: The main problem in developing software product families is how to share effort and reuse parts of design and implementation while providing variation of features and capabilities in the products. We discuss the mechanisms that are commonly used to achieve reuse and sharing in product families. The variance can be either ad hoc or predictable, and it can be managed either by making separate designs for each variant, or in other extreme by designing a single product that has the functionality of all the variants. Our analysis motivates a need for a new mechanism to deal with ad hoc variation. We propose an approach based on design configuration. It enables sufficiently detailed designs for every variant and at the same time achieves a level of design reuse without making designs unnecessarily complex or implementations inefficient.*

## 1. Introduction

Companies must address the requirements of different market segments by making products with a choice of functional features and capabilities. For example, national standards impose constraints on product functionality and implementation, cultural differences and fashions add variation to user interface design, and advances in technology require frequent migration of products to new implementation and integration platforms and environments.

Companies can reduce the development, maintenance, and support costs of similar products by sharing effort and reusing parts between different products. To manage such sharing and reuse, related products are organized into families or product lines. Software-intensive products present an especially promising target for family-based reuse since large part of the cost is in design and development, and the manufacturing is inexpensive. The idea of software product families is at least twenty years old [8]. In this paper we concentrate on software and use ‘product’ to mean the software of a software-intensive product.

The main problem in developing software product families is how to achieve sharing of effort and reuse of design and implementation while providing variation of features and capabilities in final products. In this paper we discuss the mechanisms that have been used to manage variation in product families and analyze when they should be used. This analysis motivates the need for a new method to manage design and implementation reuse in product families with ad hoc variation of features of variant products.

In the software products that we have studied, the main mechanism to manage variance has been preprocessing (using compiler flags) and configuration management of program files. This has complicated the implementation and made the programs hard to maintain. In some of the cases the situation could have been improved by creating higher-level abstractions and by using features of modern programming languages. However, in many cases the nature of variance is such that it cannot be modeled by higher-level constructs.

Software reuse in product families is often understood as sharing the program code used in variant products. The difference in product features then is achieved by having variable parts of programs included or excluded based on some condition. To control inclusion and exclusion of variable program segments developers use conditional compilation, configuration parameter files, and support of configuration management tools. Unfortunately these mechanisms do not scale well due to non-local dependencies between variable parts of the programs.

Examining programs whose configuration is determined by hundreds of interdependent conditions we often felt that the problems are due to inferior design choices, lack of abstraction, and limitations of old programming language technology. This is partly true. However, often variation of features across a program family is such that introduction of abstractions that would allow treating variant programs uniformly does not simplify the design but makes it more complex and unnecessary constraints the implementation.

For example a company that starts to produce GSM mobile phones should pay attention to other TDMA standards such as DECT, PDC (used in Japan), and IS-54 TDMA (in USA). If the developers can structure the software so that the standard-specific parts are separate from the standards-independent parts, and can introduce protocol abstractions to treat uniformly the standard-specific parts, then much effort can be saved later when the company starts to produce phones for the other TDMA standards.

However, a company that was producing mobile telephones for analog standards (AMPS or NMT) before the emergence of digital mobile communication could not have predicted the effect of digital standards on the structure of the software. This is an example of *ad hoc* variation on large scale. Ad hoc variation is often present on a finer scale as well. For example, application-specific integrated circuits (ASIC) of similar functionality may have very different interfaces. If the members of the product family use different ASICs, the software must adapt to this variability.

Attempts to deal with ad hoc variation of features using by looking for abstractions that allow to treat variants uniformly lead to unnecessary complex designs and inefficient implementations. On the other hand implementation configuration while avoiding artificial abstraction of unstructured or unpredictable variation fails to utilize design as a tool for managing complexity. In this paper we argue that ad hoc feature variation in a product family is best addressed by design configuration an approach similar to implementation configuration but applied to software designs.

We will next discuss the methods that are used to manage variance in product families, with examples. We start with the most common, implementation configuration, which relays on a single design and deals with variance on the implementation level. We will then cover customization and modularization, which both attempt to model variance. The shortcomings of these methods motivates the need for a mechanism to manage on the design level variance that cannot easily be modeled. We propose *configurable designs* as a solution, and present a method, *overlay designs*, to facilitate in configuring designs.

## 2. Implementation Configuration

Program text manipulation through conditional compilation and source code configuration management is the most common way to achieve sharing and reuse of software between different products. Product families are often created so that one variant in the product family is first designed and implemented completely. When different functionality is needed, the implementation of the first variant is modified where necessary to match the new requirements, either by using conditional compilation or by creating a new program file. The first variant serves as a prototype that always is modified first in case of improvements or bug fixes. The changes are then propagated manually to other variants in the family.

In this case, all variants in the product family have the same design. In general, we use the term *implementation configuration* when there is only one design for all products in the family that does not reflect the differences between variant products. Such a design may be relatively simple. However, when variance constitutes a major source of complexity in product development, design that does not reflect the variance does not serve the main purpose of design -- managing complexity. As the number of products in the family grows, managing variation through configuration on the implementation level becomes very complex.

Implementation configuration is often done using source code configuration management. This requires that variance is mapped to different variants of the source files. However, using complete source files to manage variance can lead to granularity mismatches as we will show in an example.

Source file preprocessing achieves smaller granularity of variance. Most programming systems today contain preprocessing facilities that allow conditional compilation and macro definitions. When software implementation is shared by several products that require non-trivial variation of features, configuring the implementation to match the features accounts for much of the implementation complexity. The mapping between the elements of product variability onto the elements of software implementation variability may be very complex. This leads to two types of problems:

1. Product configuration for delivery requires thorough knowledge of the implementation.

2. Information is replicated and non-local dependencies exist in the implementation.

We demonstrate some of the problems associated with using implementation configuration for achieving feature variation and code reuse in the second example of this section.

## 2.1 Configuration at the file level

The first example is about implementing the communications protocols that a telephone switch must support. The protocols are usually standardized but in many cases there are national or customer specific variations. The variance in protocols is mainly in the message parameters. The fields (parameters) in a specific message can have slightly different semantics. The implementation of the protocol must receive different messages and interpret the contents of the fields in the messages. It must also check that the values in the fields are valid. The protocol implementation works also in the opposite direction: it must pack information to the fields in the messages and send them outside of the switching center.

The variance in the fields of messages could be managed by implementing message unpacking and packing with two procedures for each field type. Each pack/unpack procedure pair might occupy a single source file. The variance in a field can be managed by making variants from the source file that contains the pack/unpack procedures for that field. Thus the source files can be managed by a conventional version control system.

The representation of different types of values in the message fields can be quite complicated. Often there are many semantic constraints on the allowable values that the implementation must check. The packing and unpacking procedures can be about 500 lines of code together.

This approach would work well when the different procedures are very different from each other. However, if the variance is very subtle, the unit of variance is much smaller than a source file containing two procedures. Thus there would be a severe mismatch in the granularity of variance management and the actual variance.

## 2.2 Configuration by preprocessing

The example below is adapted from a real product. It shows part of a program to fill the fields in the ‘setup’ message of the Signaling System 7 protocol (see [6]), the trunk line signaling that the switching centers use to communicate to each other. There are ten national variants of the SS7 signaling protocol that the switch must handle. In this example some of the SS7 functionality that is present in some variants is made optional by using conditional compilation.

The programming language in this example is an executable variant of SDL, the IEEE standard System Definition Language (see [3]). The procedural parts of the language are similar to Ada, Pascal or Modula-2.

There are three compiler flags for different optional properties of SS7 signaling and one flag that is used to adapt the procedure to different system environments (`m7`). The flags are `sub` for subaddressing in the network, `uus` for user-to-user signaling and `tran` for transit facility. Note how the `uus` flag is used in many places.

This kind of configuration makes it very hard to recognize and understand the different dependencies in the program. The code becomes more difficult to read and this complicates desktop testing and code reviews. In unit testing, the module must be compiled with all possible compiler flag combinations to make sure that it behaves as specified. Finally, since preprocessing does not follow the semantics of the programming language, the compiler parameters cannot be tested independently of each other.

Nevertheless, there is an important use for this kind of low level management of variability: software products that must be able to compile and link in many different programming environments. These products must be for example aware of defects and restrictions in certain compilers and environments. This kind of variance is really variance at the program text level, which is exactly what pre-processing

```
PROCEDURE pack_setup(
    IN/OUT ccsr_data sr_internal_data,
    IN/OUT opt_part  optional_part,
);
DECLARE
#if (sub)
    facility_info facility_used_info,
#endif
#if (uus)
    utu_segments utu_segments,
#endif
    send_facility bool;
BEGIN
...
#if (m7)
#else
    TASK memset(
        @facility_info,
        0,
        SIZEOF(facility_used_info)
    );
    TASK memset(
        @facility_info.subaddress_a,
        0xFF,
        SIZEOF(subaddress)
    );
    TASK memset(
        @facility_info.subaddress_b,
        0xFF,
        SIZEOF(subaddress)
    );
#endif
```

```
#if (tran)
    CALL pack_transit(opt_part);
#else
#endif
    CALL pack_lowlayer_comp(opt_part);
    CALL pack_highlayer_comp(opt_part);
#if (uus)
    DECISION ccsr_data.c1.utu_exist;
    ( T ):
        CALL pack_utu(
            opt_part,
            utu_segments
        );
    ENDDECISION;
#else
#endif
    RETURN;
ENDPROCEDURE pack_setup;
```

is designed to do.

## 3. Customization and modularization

Managing family variance by *customization* means that all variants are supported by one “universal” product that may be customized by the maker or by the customer to behave as any specific variant. Hence customization enables one to change product capabilities, supported features, and modes of operation. All possible components must be present in the delivered product, but the active set of components is selected by customization procedures. The relationships between the various components are fixed and thus the design is static. There is one design and one implementation that is customized to achieve variation in its features and capabilities.

Customization is often used because software engineers are trained to design single a single product rather than a family of products. Also most design methods only address the design of a single product. In some ways managing family variance by customization also makes the maintenance of the family simpler, as there is only one design, one implementation, and one package to ship to the customers. This also offers the best possibilities for reuse of common functionality in different variants as these are localized to the same product. However, in practice the design and implementation of the customizable product can be significantly more complex and costly than individual variants.

Providing family variation through customization is often an overkill. To produce a number of variants it is not necessary to design a universal machine in which all possible features and capabilities must be integrated and coordinated. Designs could be made simpler if the reuse aspect of family development were addressed by designing sets of reusable components and domain specific frameworks. When variation in the application domain is well understood one can achieve significant reuse between the variants in a product family through structuring the variation in design with reusable components. While each variant has its own design, many components in their designs can be shared between different variants.

When variance in the application domain is predictable, it can also be modeled on the design level. High-level abstractions can then localize the variability. Modularization goes further than customization in making the abstractions independent of each other. We will show in an example how creating abstractions and then making them independent will make the design more flexible but also more complex.

In modularization, variation is localized to structural elements of the design and variants are produced by selecting an appropriate set of components. The shared part of the design (the family architecture) is a framework that is further extended by selecting existing and specifying new components, and establishing relationships between them. Both the choice of the components and their relations may change from product to product.

Designing and developing a family architecture and generic components that may be shared by different products in a family is a complex task. Furthermore, such product implementations often require more run-time resources than simpler implementations of variant products.

## 3.1 Example of customization and modularization

Our example is the number analysis that a telephone switch performs. When a telephone user attempts to make a call, he or she dials digits that the telephone switch analyzes to determine the destination of the call. Since some destinations may be reached through several alternative routes, the switch will have to select among them. A telephone call may be characterized by a number of attributes that are used to select a possible route, such as whether the call is a voice or a data call, and the type of signaling required by the calling party.

The switch maintains an array of destination records to implement call routing functionality. The destination records list operator-selected routes to the destination. Each route may be characterized by a number of attributes, such as congestion, availability, and bandwidth.

The attributes of the call and the attributes of the routes are used to make the routing decision. For example, for calls with a minimum bandwidth requirement, such as data calls, the routes will be rated based on the their bandwidth attribute. Also, since satellite links are slow, a voice call must be routed so that it uses at most one satellite link.

In a telephone network, the range of variation in the capabilities of call routing subsystem is relatively well known. A customizable product implements all possible variants and provides the user a customization interface to select the active subset. For example, there could be four different route selection algorithms that the operator can select from. Each destination record would indicate which algorithm is used with this destination. The operator uses a management interface to customize the routing subsystem by changing the information in the destination records.

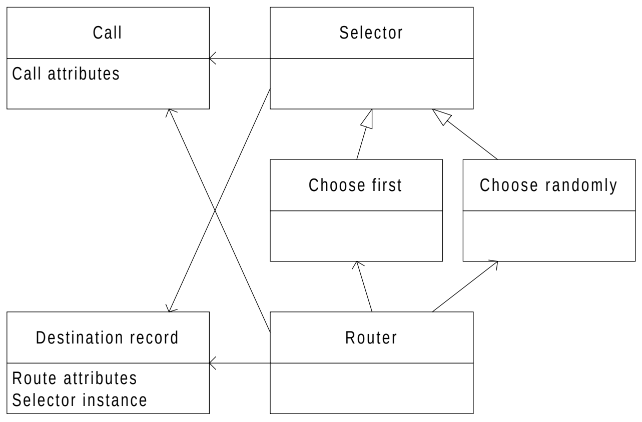


*Figure 1: Routing subsystem design*

Let us consider the design in Figure 1. It provides customization of the selection algorithm: all algorithms are always present and the operator can select any of them to be specified in a destination record.

The classes are Call, Router, different Selector subclasses for different selection policies and a Destination record class. The class Selector is a generalization of the classes Choose first and Choose randomly. This is indicated by the array with a hollow head. The other arrows indicate navigability: an object at the other end of the arrow know about the object being pointed at and can thus use it to provide services. The notation is the Unified Modeling Language [2].

A router object will look at a destination record table to find out the appropriate selector object for this destination. Then it will ask the selector object to get a possible route for the call and destination pair. The selector object will use the attributes of the call and the attributes of the routes from the destination record table. Different selectors implement different selection policies and use different attributes to do that.

A design of a customizable call routing system would include all the different attributes needed to cover the variability. Many components in the system depend on the attributes. The different selector objects all use the attributes differently. To add a selection policy that uses a new attribute, the call or destination record class will have to be changed. There is no way to extend the behavior of the selectors incrementally because there are no abstractions on the attributes.

The modularization that is done in a customizable system is only targeted to handle the customization process. It does not cover all variability in the domain. In our example the customizable feature was the route selection policy. The lack of modularization makes any other changes hard.

To make the example modular, we model the variance in the call and destination attributes. We can create new classes for call attributes and route attributes, and parametrize the selector destination records and call objects with this type. This makes the call class into a second-order concept, a parametrized class. To do that, we needed to abstract the notion of a call attribute. If we did not already have such a concept, creating the second-order class would have been much harder.

Creating abstractions allowed us to localize all predictable variation in the example to design elements. Using customization for attribute variation would have led to a very large and slow implementation, while configuration would have left the variability to the implementation. Modular design models all known variation; customization models only the variation which is visible in the product. At the same time, modularization complicated the design by creating new high-level abstractions. In many real-life cases the variance is ad hoc and there are no models for it. In those cases modularization would either be impossible or at least be more difficult.

# 4. Design configuration

We have studied several families of software-intensive telecommunications products, and in most of them source-code configuration was the main method to achieve variation of features. This was not adequate in a number of cases where support for variation was a major source of complexity. Since variation was not handled by the designs, the complexity of the implementation led to serious problems in controlling the evolution of these product families. Also, the level of reuse in these systems was very low which lead to high development and maintenance costs.

In some cases the situation could be improved by modular designs and customizable parts, using the common mechanisms of indirection, deferred binding, parametrization, and higher order abstractions. This would improve reuse and simplify implementation. However, the cost of design would grow and, in many cases, implementation would be less efficient. Often this is not acceptable.

When feature variation between variant products is not systematic and cannot be predicted, or when implementation must be optimized to minimize the use of hardware resources, a different approach is needed.

We investigate a technique based on *overlay designs.* The idea may be illustrated with a metaphor of overhead slides that may be stacked to create a composite image. Suppose you want to use a number of slides that are similar in most parts and only differ in details. One way is to create the first slide completely and then copy and modify its contents. This is fine as long as no changes need to be made later in the shared part of the slides. This is rarely the case. A better approach would be to create the shared part on one (master) transparency and use additional overlay transparencies to show the variations.

The main idea of overlay design is to have a specific design for each variant without having to consider family variation. The designs of variant products are compared for common and different parts. The common part is separated as a shared overlay and the different parts are kept as variant overlays. A complete design for a variant product is thus a simple combination of several overlays. This way we achieve design reuse without introducing unnecessary complexity associated with reconciling and abstracting differences between variants. Also the implementation is kept efficient because there is no need to implement generic mechanisms. This is illustrated in Figure 2..

| | Ad hoc variation | Predictable variation | |
|---|---|---|---|
| Single design | Customization<br>Indirection and deferred binding | Implementation configuration<br>Preprocessing and configuration management | Implementation simplicity $\downarrow$ |
| Multiple designs | Modularization<br>Parametrization and second-order abstractions | Design configuration<br>Overlay designs | |
| | Design simplicity, implementation efficiency | $\rightarrow$ | |

*Figure 2: Keeping design simple and implementation efficient*

## 4.1 Design overlays

In the following example, we consider the design of a telephone switch implementing different call control protocols. Call control connects the telephone calls going through the switch. It incorporates variance that has traditionally been very hard to manage, and consequently the different call control protocol designs and implementations have been separately developed and maintained.

In the example we will consider the ISDN, GSM, and ATM call control protocols (see [4], [5] and [1] for these). ISDN is relatively complicated because it allows the user to either dial the whole number before sending it to the switch or to dial the number digit by digit. The GSM protocol, in turn, is complicated because it cannot trust the communication medium to be reliable. The ATM protocol is the most simple one because the dialers are assumed to be computers or at least rather intelligent telephones.

We use SDL state transition diagrams to depict the protocols, as it is a standard design notation for telecommunication software. In the diagrams, the user (caller) is imagined to be on the left-hand side and the connection network on the right-hand side. The protocols are somewhat simplified: The diagrams will show only the call set-up part of the protocols, and we have dropped time-outs from them.

### 4.1.1 Creating Design Overlays

We now present the design diagrams of the three call control protocols (Figure 3). Then we proceed to identify the shared parts of the designs and generate design overlays to illustrate the common and variant parts of the designs. There are three steps in generating design overlays.

The first is *mapping* the design elements in the different designs by identifying the common elements in the diagrams and renaming them consistently. This task requires some domain expertise because corresponding states and messages have dissimilar names in different protocols.

The second step is *separation* of the common elements from the original designs on a separate overlay. This yields a kind of template for the designs of variant protocols. The variant parts in each protocol then make variant overlays. These are a kind of instantiation parameters or configurations for the design template.

The third step is *laying out* the overlays so that the shared overlay can accommodate any of the variant overlays. This way we achieve the visual effect of stacked overhead transparencies.

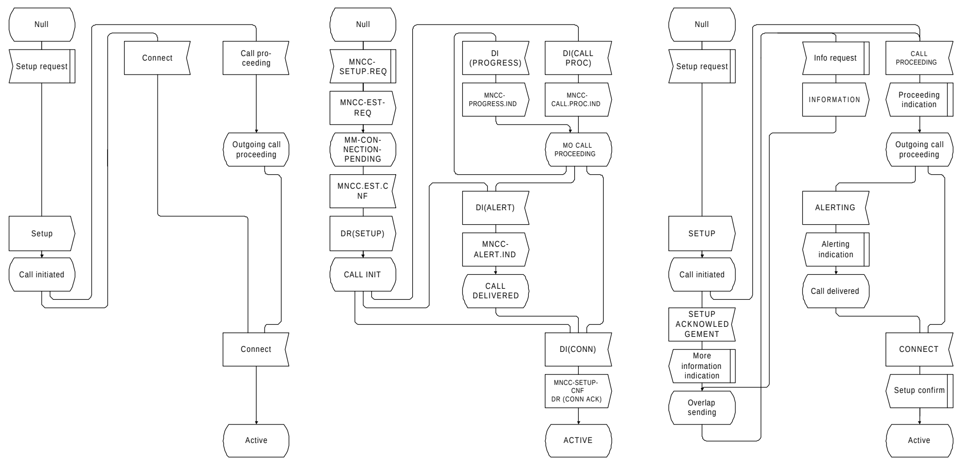
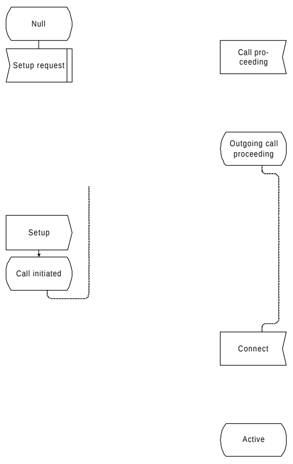


*Figure 3: ATM, GSM and ISDN Call Setup protocols before mapping.*

A variant design is produced by applying the corresponding configuration to the design template. Figure 4 shows variant designs for the three protocols produced in this way. The common elements (states and messages) have been identified and renamed. The parts shaded in gray correspond to the shared overlay. This is an illustration of how design overlays could support sharing of parts between similar designs.

In this simple example we have only one shared and one variant overlay to produce a complete design. However, since variant overlays are partial designs in their own right, the technique may be applied recursively to factor shared parts of several variant overlays. Clearly, to make this approach practical, tool support is needed for managing shared and variant overlays.


Null
Setup request
Call pro-ceeding
Outgoing call proceeding
Setup
Call initiated
Connect
Active