Configuring Designs for Reuse

Anssi Karhinen  
Nokia Research Center  
PO Box 45,  
FIN-00211 Helsinki, Finland  
+358 9 4376 6476  
anssi.karhinen@research.nokia.com

Alexander Ran  
Nokia Research Center  
3 Burlington Woods Drive #250  
Burlington, MA 01803 USA  
+1 617 238 4904  
alexander.ran@research.nokia.com

Tapio Tallgren  
Nokia Research Center  
PO Box 45  
FIN-00211 Helsinki, Finland  
+358 9 4376 6849  
tapio.tallgren@research.nokia.com

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, and the kind of variance each is best suited for. Our analysis motivates a need for a new mechanism to deal with ad hoc variation of features found in different members of a family. We argue that higher level abstraction and parameterization techniques are not well suited for this task. We propose an alternative approach that 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.

Keywords
Design and implementation reuse, product families, configuration of designs.

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, cultural differences and fashions add variation to user interfaces, and advances in technology require frequent migration of products to new implementation and integration platforms and environments.

To reduce the development, maintenance, and support costs of similar products, companies can share effort and reuse parts between different products. Related products are organized into families or product lines to manage such sharing and reuse. Software-intensive products present an especially promising target for family-based reuse since a large part of the cost is in design and development rather than manufacturing. The idea of software product families is at least twenty years old [8], however achieving reuse in a product family still presents a considerable challenge. In this paper we concentrate on software and use 'product' to mean the software of a software-intensive product.

The main problem in development of 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 conditions for their applicability. This analysis motivates the need for a new method to manage design and implementation reuse in product families with ad hoc variation of features.

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 of program code. The variance in product features is achieved by including or excluding parts of program code. For this, developers use conditional compilation, configuration parameter files, and source code configuration management tools. However, there are often non-local dependencies between the variable parts of the program text that are not made explicit. Thus these mechanisms do not scale well.

Examining programs where configuration is determined by hundreds of interdependent conditions often gives the impression that the problems are due to inferior design choices, lack of abstraction, and limitations of old programming language technology. In some cases it is true. However in other cases feature variation across a program...
family is such that introducing abstractions that would allow treating variant programs uniformly would not simplify the design but would only unnecessarily constrain the implementation. This happens when products in a family exhibit ad hoc variation of features.

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 standard-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 (ASICs) 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 abstractions that allow treating variants uniformly lead to unnecessarily complex designs and inefficient implementations. On the other hand, program text configuration avoids artificially abstracting unstructured or unpredictable variation but 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 configuring designs. This is similar to program text configuration but is applied to software designs.

In the remainder of the paper, we will first discuss the methods that are used to manage variance in product families, with examples. We start with the most common method, implementation configuration, which relies 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 to manage certain types of variance motivates the need for another mechanism to manage variance that cannot easily be modeled. Finally we present our ideas for a solution based on configuring designs.

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 new variant program files. The first variant serves as a prototype that is always 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 documented 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, a design that does not reflect the variance does not serve the main purpose of design: it does not manage complexity. As the number of products in the family grows, managing variation through configuration of implementation 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, when variance is on a smaller scale than files, this leads to a mismatch between the granularity of variance and the granularity of configuration management as we will show in an example.

Source file preprocessing achieves finer 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.

Configuration at the file level
The first example is about implementing the communication protocols that a telephone switch must support. The protocols are usually standardized but in many cases there are national or customer specific variations. For instance, the message parameters (fields)
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 messages fields 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 as much as 500 lines of code together.

This approach would work well when the procedures are very different from each other. However often 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.

**Configuration by preprocessing**

The example in Figure 1 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 with 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, *uss* for user-to-user signaling and *tran* for transit facility. Note how the *uss* 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 difficult to read and this complicates desktop tests 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.

```c
PROCEDURE pack_setup(
  IN/OUT ccssr_data sr_internal_data,
  IN/OUT opt_part optional_part,
);)
DECLARE
# if (sub)
  facility_info facility_used_info,
# endif
# if (uss)
  uuu_segments uuu_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 (uss)
DECISION ccssr_data.cl.uuu_exist;
( T ):
  CALL pack_uuu(
    opt_part,
    uuu_segments
  );
ENDDECISION;
# else
# endif
RETURN;
ENDPROCEDURE pack_setup;
```

Figure 1: Configuration of source code by preprocessing

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 is designed to do.

**CUSTOMIZATION AND MODULARIZATION**

Managing family variance by *customization* means that all variants are supported by one “universal” product that may be adapted 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 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 that of individual variants. Customizable software incurs “software manufacturing costs” by increasing required hardware resources.

When variance in the application domain is predictable, it can also be modeled on the design level [9]. The reuse aspect of family development can be addressed by designing reusable components and domain specific frameworks. While each variant has its own design, many components in their designs can be shared between different variants. We call this modularization. We will show an example how modularization can make 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.

**Example of customization and modularization**

Let us consider an example based on the number analysis and route selection that a telephone switch must perform. 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 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.

![Routing subsystem](image)

*Figure 2: Routing subsystem*

Let us consider the diagram in Figure 2. 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, 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 arrow with a hollow head. The other arrows indicate navigability: an object at the other end of the arrow knows about the object being pointed at and can thus use it to provide
services. (We used the Unified Modeling Language [2] notation.)

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.

A customizable system has been designed to manage certain variance using an adaptation mechanism, but it does not manage all the variance in the application domain. In the example, the customizable feature is the route selection policy, and it would be difficult to make any other changes.

To make the example more 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 by attribute types. 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 to implement attribute variation would have led to a very large and slow implementation, while using source 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 is either impossible or at least more difficult.

**DESIGN CONFIGURATION**

In several families of software-intensive products that we have studied 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 transparencies 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 3.

![Figure 3: Keeping design simple and implementation efficient](image-url)
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 designs and implementations of call control protocol have been separately developed and maintained.

In the example we will consider the ISDN, GSM, and ATM call control protocols (see [4], [5] and [11]). 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 simplest 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 timeouts from them.

Creating Design Overlays
We now present the design diagrams of the three call control protocols (Figure 4). 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. The following are the three steps to create design overlays.

The first step 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 like instantiation parameters or configurations for the design template.

Figure 4: ATM, GSM and ISDN Call Setup protocols before mapping.
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.

A variant design is produced by applying the corresponding configuration to the design template.

Figure 5 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 a simplest case there exists 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. In our example, GSM and ISDN protocols share two additional parts that are colored light gray in figure 5. Clearly, to make this approach practical, tool support is needed for managing shared and variant overlays.

In order to provide automatic support for overlay design the representation of templates and configurations should be formalized. Such a formalization might be quite different for different design notations. So far we only experimented with SDL process type diagrams. In this case one possible approach to represent design templates and configurations is illustrated below. We extended standard SDL with named parameter slots that are used when defining a template diagram. Configuration diagrams are collections of SDL fragments marked by the name of the corresponding parameter of the design template.

Figure 6: Shared SDL template with parameter slots

The parametrized SDL template diagram and the parameter fragments of SDL diagrams for GSM protocol are shown on figures 6 and 7. Note that alternative paths that are identified with broken lines are not present in an
instantiation of the generic state diagram if the parameter slots of that path are empty.

There are several restrictions on the SDL fragments that can be used as instantiation parameters. For example: they must not contain any of the states of the template; they must not begin with the reception of a message that is already received at the same state in the template.

Note that a designer never needs to manipulate (or even see) the design template or configurations of design fragments. This is an example of a representation to be maintained by a tool supporting design configuration. A designer only works with complete designs of specific variants. A support tool might also automatically maintain the restrictions of the design overlays. Naturally, variant designs include the information on which parts of the design are currently shared with other variants in the family.

Design overlays provide sharing in the designs of a product family. How can we achieve sharing also at the implementation level? Since the individual design overlays are neither complete nor generic designs, they cannot usually have sensible implementations.

```
state Null
input Setup_request
  output Setup
nextstate Call_initiated
endstate Null
state Call_initiated
input Call_proceeding
  output Proceeding_indication
nextstate Outgoing_call_proceeding
input SETUP_ACKNOWLEDEMENT
  output More_information_indication
nextstate Send_message
endstate Call_initiated
state Send_message
input Call_proceeding
  output Proceeding_indication
nextstate Outgoing_call_proceeding
input Info_request
  output INFORMATION
nextstate Overlap_sending
endstate Send_message
state Outgoing_call_proceeding
input Connect
  output Setup_confirm
nextstate Active
input ALERTING
  output Alerting_indication
nextstate Call_delivered
endstate Outgoing_call_proceeding
state Call_delivered
input Connect
  output Setup_confirm
nextstate Active
endstate Call_delivered
state Active
endstate Active
```

Figure 8: ISDN implementation

The granularity at which the implementation sharing will occur is determined by the granularity at which sharing occurs in the design overlays. The elements of the design map to implementation constructs but not at the granularity that is normally used for implementation level sharing. Normally sharing in the implementation takes place at function level or module level.

Fine granularity and functional incoherence of shared program code requires tool support to make the sharing at the implementation level practical. The tools might include a graphical editor that is able to show selected variants of the source code and protects the shared code from unwanted changes. Tool-generated program text preprocessing can be used as an implementation.

Figure 8 and Figure 9 illustrate this idea for the call setup family. The granularity of code sharing is at SDL statement level. The statements are state, which begins a state definition; endstate, which ends a state definition.
input, which receives an event, output, which causes an event and nextstate, which causes a state transfer.

```plaintext
state Null
input Setup_request
  output MNCC-Est-REQ
nextstate MN-CONNECTION-PENDING
endstate Null
state MN-CONNECTION-PENDING
input MNCC.EST.CNF
  output Setup
nextstate Call_initiated
endstate MN-CONNECTION-PENDING
state Call_initiated
input Call_proceeding
  output MNCC.CALL.PROC.IND
nextstate Outgoing_call_proceeding
input D1(ALERT)
  output MNCC.ALERT.IND
nextstate CALL-DELIVERED
input Connect
  output MNCC.SETUP.CNF
nextstate Active
endstate Call_initiated
state Outgoing_call_proceeding
input Connect
  output MNCC.SETUP.CNF
nextstate Active
input D1(PROGRESS)
  output MNCC.PROGRESS.IND
nextstate Outgoing_call_proceeding
input D1(ALERT)
  output MNCC.ALERT.IND
nextstate CALL-DELIVERED
endstate Outgoing_call_proceeding
state CALL-DELIVERED
input Connect
  output MNCC.SETUP.CNF
nextstate Active
endstate CALL-DELIVERED
state Active
endstate Active
```

Figure 9: GSM implementation

The programmer is not required to work on the shared parts implementation only. Instead, the shared parts would always be seen in the context of a variant implementation. Change propagation is handled by the tool. If the programmer modifies an element that is part of many variants the system notifies of the need for and monitors recompilation and testing.

**RELATED WORK**

Domain-specific software architecture (see [12] for an example) program starts by creating various domain models, and proceeds to create a software architecture based on the actual and projected commonalities and differences of the systems. The approach assumes that the differences can be known in advance. Configurable designs is a complementary mechanism to this to deal with ad hoc variation.

Parametrized programming (see [13]) allows one to create a software architecture by defining modules and their interconnections. The modules can satisfy formal theories, be parametrized, inherit one another, and several modules can create packages. By selecting different module instantiations and connecting them differently, one can create a product family. However, this assumes that the parametrized modules and the parameters are chosen so that the modules exhibit sufficient variation. This is not reasonable when the variance is ad hoc, as the number of parameters would grow too large. Configurable designs do not assume that the variance can be parametrized.

More abstractly, parametrized programming and related approaches deal with the high-level organization of software implementation. They attempt to explain how module implementations are different in members of a product family, while the approach of configuring designs tries to show how variants are different on the design level. Parametrized programming enables one to modify the implementations in different products automatically, as there is a direct link between implementation and modules. Also, conformance checking could be automated. Since software design is a description of implementation and not a specification, it is possible to represent on the design level variation that is not intelligible in the implementation. This helps the system designers to understand the variance, even if the changes to implementation still must be done manually.

Product-line architectures have lead to great savings with product families (see, for example, [10] and [11]). The main factor has been reusing components over a range of products, supported by a strong architecture that enables this. Configurable designs can be used even when variance cannot be explicitly modeled. Thus they can be used in a product line for components that show some ad hoc variance.

**CONCLUSIONS**

The main objective of organizing similar products into families is to reduce the development and maintenance costs of each product by sharing some of the effort and parts between family members.

When building a family of software-intensive products for a fixed and well understood application domain that does not require to minimize the use of computing resources, one may consider integrating all capabilities in one customizable product. This is the approach taken by most PC software manufacturers.

When a family of software-intensive products exhibits predictable feature variation, but is growing and evolving it might be possible to use modular designs to isolate variability and to achieve good levels of reuse.

When a family of software-intensive products exhibits ad hoc feature variation, design configuration allows for simplest designs and most efficient implementation for each individual product.
Design overlays is a simple and intuitive approach to design configuration that offers a way to manage families of designs for sharing and reuse of parts without increasing complexity unnecessarily. The implementations of complete designs of different variants that are generated or managed by combining overlay designs can be efficient as there is no overhead of generic mechanisms in the implementation.

In practice, designers will need to utilize all available techniques of implementation configuration, customization, modularization and design configuration to achieve reuse with different types of variation in the same product family.

Our plans for future include trying out design overlays with other types of design diagrams. We will also investigate implementation strategies for product families managed by design configuration and try to integrate several mechanisms for achieving reuse and variation in a family. Eventually, we plan to build tools to support management of overlay designs and corresponding implementations.

ACKNOWLEDGMENTS
The authors wish to thank Juha Kuusela, Jianli Xu and Alessandro Maccari for their valuable comments. The call routing example is based on an unpublished paper by Philip Ginzboorg.

REFERENCES
5. GSM 04.08 version 4.11.0. ETSI draft prETS 300 557, May 1995.