On Feature Orientation and on Requirements Encapsulation Using Families of Requirements

Jan Bredereke

Universität Bremen, FB 3 · P.O. box 330 440 · D-28334 Bremen · Germany
brederek@tzi.de · www.tzi.de/~brederek

Abstract. Naive feature orientation runs into problems with large software systems, such as telephone switching systems. With naive feature orientation, a feature extends a base system by an *arbitrary* increment of functionality. Information hiding helps to structure a large software system design into modules such that it can be maintained. We focus on the requirements of a software system. Requirements can be structured analogously to design modules. Naive feature orientation can violate requirements encapsulation. We survey approaches with improved encapsulation, and we show how and when families of requirements can help.

1 Introduction

A feature oriented description of a software system separates a base system from a set of optional features. Each feature extends the base system by an increment of functionality. Feature orientation emphasizes the individual features and makes them explicit. The description of one feature does not consider other extensions of the base system. Any interactions between features are described implicitly by the feature composition operator used.

Feature orientation is attractive. It meets the needs of marketing. Marketing must advertise what distinguishes a new version from its predecessors. Marketing must offer different functionality to different customers, in particular at different prices. Successful marketing also demands a short time to market. This requires that the system can be changed easily. It can be achieved by just adding a new feature. The large body of existing descriptions never needs to be changed.

But naive feature orientation runs into problems with large software systems, such as telephone switching systems. In this paper, we show where naive feature orientation can violate information hiding, and how and when *families of requirements* can help.

2 Feature Orientation

2.1 Naive Feature Orientation

With *naive* feature orientation, a feature extends a base system by an *arbitrary* increment of functionality. The increment is typically chosen to satisfy some new

user needs. This selection of user needs happens from a marketing perspective. In particular, the selection is neither particularly aligned to the internal structure of the software system nor to the organization of the system's documented requirements.

Many feature addition operators have been used in practice or proposed on theoretical grounds [1, 2, 3, 4, 5, 6]. They typically share the property that they add code in different places of the base system as needed. They are therefore operators of a syntactic nature.

A canonical example is the structure of the Intelligent Network (IN) [7, 8, 9]. The IN is the telephone switching industry's currently implemented response to the demand for new features. This example demonstrates the naive feature orientation nicely. This remains true even if the IN might be replaced by emerging architectures eventually, such as Voice over IP (VoIP).

The IN specifies the existence of a Basic Call Process (BCP) and defines sets of features. Examples of IN features are listed in Fig. 1. When a feature is triggered, processing of the BCP is suspended at a Point of Initiation (POI), see Fig. 2. The feature consists of Service-Independent Building Blocks (SIBs), chained together by Global Service Logic. Processing returns to the BCP at a Point of Return (POR). The Basic Call Process consists of two automata-like descriptions, one for the originating side of a call, and one for the terminating side of a call, see Fig. 3. In these, a feature can be triggered at a so-called Detection Point, and processing can resume at more or less any other Detection Point. This allows a feature to modify basic call processing arbitrarily.

There has been considerable research effort on feature composition operators. In particular, in the FIREworks project [10, 11] (Feature Interactions in Requirements Engineering), various feature operators were proposed and investigated. These operators successfully reflect the practice of arbitrary changes to the base system. The theoretical background is the superimposition idea by Katz [12]: one specifies a base system and textual increments, which are composed by a precisely defined (syntactic) composition operator.

A feature is inherently *non-monotonous* [15]. Most features really change the behaviour of the base system. That is, a feature not only adds to the behaviour of the base system, or only restricts the behaviour of the base system. For example, in telephony a call forwarding feature both restricts and adds to the behaviour. It prevents calls to the original destination, and it newly makes calls to the forwarded-to destination. Therefore, a refinement relation is not suitable to describe adding a feature.

2.2 Feature Interaction Problems in Telephone Switching

It turns out that severe feature interaction problems appear if one applies a naive feature oriented approach to a large software system, such as a telephone switching system. It is relatively easy to create a new feature on its own and make it work. But it becomes extremely difficult to make all the potential combinations of the optional features work as the users and providers expect. The telecom industry complains that features often interact in an undesired way [1, 2, 3, 4, 5,

Abbreviated dialling	Customized ringing
Attendant	Destinating user prompter
Authentication	Follow-me diversion
Authorization code	Mass calling
Automatic call back	Meet-me conference
Call distribution	Multi-way calling
Call forwarding	Off net access
Call forwarding on busy/don't answer	Off net calling
Call gapping	One number
Call hold with announcement	Origin dependent routing
Call limiter	Originating call screening
Call logging	Originating user prompter
Call queueing	Personal numbering
Call transfer	Premium charging
Call waiting	Private numbering plan
Closed user group	Reverse charging
Consultation calling	Split charging
Customer profile management	Terminating call screening
Customized recorded announcement	Time dependent routing

Fig. 1. The features in the Intelligent Network (version CS 1).

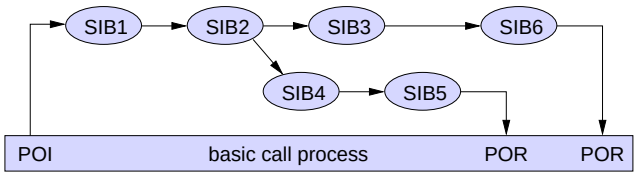


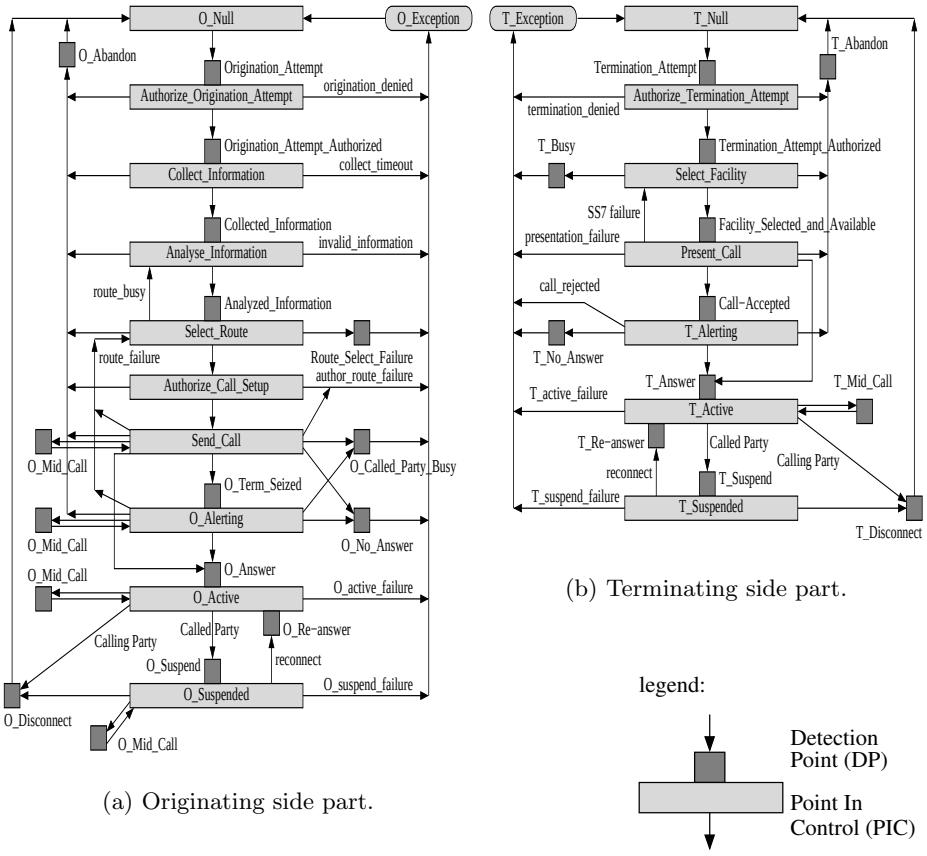
Fig. 2. feature-oriented extension in the Intelligent Network (from [13, p. 3]).

6]. There are already hundreds of telephony features. The combinations cannot be checked anymore because of their sheer number. Undesired interactions annoy the telephone users, and the users are not willing to accept many of them. The users expect reliability from a telephone system much more than from other software-intensive systems such as desktop PCs.

One typical example of a telephony feature interaction occurs between a Calling Card feature and a Voice Mail feature.

We had once a calling card from Bell Canada. It allowed us to make a call from any phone and have the call billed to the account of our home's phone. We had to enter an authentication code before the destination number to protect us against abuse in case of theft. For ease of use, we could make a second call immediately after the first one without any authorization, if we pressed the "#" button instead of hanging up.

We also had a voice mail service from Meridian at work. A caller could leave a voice message when we couldn't answer the phone. We could check for messages later, even remotely. For a remote check, we had to call an access number, dial



(a) Originating side part.

(b) Terminating side part.

Fig. 3. The Basic Call State Model of the Intelligent Network (version CS 2, after [14]).

our mailbox number and then a passcode. At the end of both the mailbox number and the passcode, we had to press the "#" button.

The interpretations of the "#" button were in conflict between these two features. The calling card feature demanded that the call should be terminated. The voice mail feature demanded that the call should be continued, and that the authorization went on with the next step. This particular feature interaction was resolved by Bell. The calling card feature required that the "#" button was pressed at least two seconds to terminate the call.

A *feature interaction* occurs when the behaviour of one feature is changed by another feature. This is a commonly accepted informal definition.

Not all feature interactions are undesired. Some features have increased value together with other features. For example, a short code to re-dial the last number dialled saves typing. This is even more helpful when one uses a (long) dialling prefix that selects an alternative, cheaper long-distance carrier. Some features are even intended to improve a system that has specific other features. (Of

course, this violates the "pure" feature oriented approach.) For example, a calling number delivery blocking feature interacts with a calling number delivery feature. The latter displays the caller's number at the callee's phone. The former prevents a caller's number to be displayed anywhere for privacy reasons.

Cameron *et al.* [16] have categorized the causes of feature interaction problems in their seminal benchmark paper: violation of feature assumptions, limitations on network support, and intrinsic problems in distributed systems. Some violated feature assumptions are on naming, data availability, the administrative domain, call control, and the signalling protocol. Limitations on network support occur because of limited customer premises equipment signalling capabilities and because of limited functionalities for communications among network components. Some intrinsic problems in distributed systems are resource contention, personalized instantiation, timing and race conditions, distributed support of features, and non-atomic operations.

A rather comprehensive survey of approaches for tackling feature interaction problems was done recently by Calder *et al.* [17]. Despite some encouraging advances, important problems still remain unsolved. The rapid change of the telecommunications world even brings many new challenges.

A new view on the causes of feature interaction problems was a main result of the recent seventh Feature Interaction Workshop [1]: in order to resolve a conflict at a technical level, we often need to look at the social relations between users to either disambiguate the situation or mediate the conflict. Zave [18] pointed out that features should be purposes, not mechanisms. Gray *et al.* [19] found that busy is a person's state, not a device's state – and the answer depends on who is asking. We [20] showed that many feature interaction problems arise because the users fail to abstract the system to the relevant aspects correctly. Other authors went into similar directions. In a panel discussion, Logrippo compared feature interaction resolution to legal issues.

3 Information Hiding Definitions

Information hiding helps to structure a large software system design into modules such that it can be maintained. We now introduce some definitions from the literature as a base for our further discussion.

A *module* in the information hiding sense [21, 22, 23] is a work assignment to a developer or a team of developers. (There are *many* other meanings of this word, we use this meaning only here.) Such a work assignment should be as self-contained as possible. This reduces the effort to develop the system, it reduces the effort to make changes to the system later, and it improves comprehensibility. A successful software system will be changed many times over its life time. When some design decision must be changed, a change should be necessary in one module only. A design decision usually must be changed when some requirement changes.

The *secret* of a module is a piece of information that might change. No other module may rely on the knowledge of such a secret. Sometimes we distinguish

between a primary and a secondary secret. A primary secret is hidden information that was specified to the software designer. A secondary secret is a design decision made by the designer when implementing the module that hides the primary secret.

The *interface* between modules is the set of *assumptions* that they make about each other. This not only includes syntactic conventions, but also any assumptions on the behaviour of the other modules. A developer needs to know the interface of a module only in order to use its services in another module.

There can be a *hierarchy of modules*. We need it for large systems. Its structure is documented in a *module guide*. The module guide describes the module structure by characterizing each module's secrets.

A fundamental *criterion for designing the module structure* of a software system is: identify the requirements and the design decisions that are likely to change, and encapsulate each as the secret of a separate module. If such a module is too large for one developer, the approach must be applied recursively. This leads to making the most stable design decisions first and those most likely to change last. The three top-level modules for almost any software system should be the hardware/platform-hiding module, the behaviour-hiding module and the software decision module. These modules must then be decomposed recursively, depending on the individual system. The structure presented in [23] might serve as a template.

An *abstraction* of a set of entities is a description that applies equally well to any one of them. An *abstract interface* is an abstraction that represents more than one interface; it exactly and only consists of the assumptions that are included in all of the interfaces that it represents. A *device interface module* is a set of programs that translate between the abstract interface and the actual hardware interface [24]. Having an abstract interface for a device allows to replace the device during maintenance by another, similar model with a different hardware interface, without changing more than one module.

Object orientation allows to use information hiding by realizing modules through the mechanism of the class.

Information hiding enables to design software for ease of *extension and contraction*. Design for change must include the identification of the minimal subset that might conceivably perform a useful service, and it must include the search for a set of minimal increments to the system [25]. The emphasis on minimality stems from the desire to avoid components that perform more than one function.

The *relation "uses"* among programs (i.e., pieces of code) describes a correctness dependency. A program A uses B if correct execution of B may be necessary for A to complete the task described in A's specification. We can facilitate the extension and contraction of a software, if we design the uses relation to be a hierarchy (i.e., loop-free), and if we restrict it as follows. A is allowed to use B only when all of the following conditions hold: (1) A is essentially simpler because it uses B. (2) B is not substantially more complex because it is not allowed to use A. (3) There is a useful subset containing B and not A. (4) There is no conceivably useful subset containing A but not B.

Information hiding is also a base for the design and development of *program families*. A set of programs constitutes a family, whenever it is worthwhile to study programs from this set by first studying the common properties of the set and then determining the special properties of the individual family member [26]. One worked-out approach is [27].

4 Requirements

We focus on the *requirements* of a software system for the discussion of feature orientation. Feature interactions often already arise in the requirements documents. When these documents are a *complete* description of the behaviours and of other interesting properties, then *all* feature interaction problems are present in the requirements documents at least inherently. Therefore, they should be tackled already there.

4.1 Families of Requirements

A *family of requirements* is a set of requirements specifications for which it pays off to study the common requirements first and then the requirements present in few or individual systems only. In particular, we are interested in families of requirements where only a subset of the family is specified explicitly in the beginning, and where more members are specified explicitly incrementally over time.

A family of requirements means that we have several versions of requirements. Requirements can and should be put under configuration management analogously to software. The "atomic objects" are properties.

The right size of the properties, when taken as the atomic objects of configuration management, depends on the size of the family. We must split up the requirements specification into small properties when a family of requirements has a large number of potentially specified members. We want to avoid to specify the same aspect A in two different properties. This can happen if we specify two aspects A, B in one property P_1 first and then need to specify A in another property P_2 again, because there is a family member that has A but not B.

In case of doubt, we should make a property in the requirements as small as possible while being useful. This is a safe strategy when we cannot overlook the entire set of family members easily. Such a specified property will be much smaller than the user of a new system or of a new feature usually thinks.

By small, we mean abstract in the above sense. A small property is part of the requirements of as many useful potential systems as possible. The goal is that each time a new member is specified, we will never need to copy and modify any existing property. The new member will only exchange one or more entire properties by one or more other properties.

When we have a large number of requirements and therefore of requirements modules, we need some additional structure. It shall help the reader of a requirements document to find easily the module he/she is interested in. The above

kind of modules is not directly suitable. The above modules are a product of the software *design*. Their secret can be a requirement or a design decision. Their structure is a software design structure. Such artefacts of the software design do not belong into the software requirements. But we can adapt the idea.

A *requirements module* is a set of properties that are likely to change together. A requirements module may be partitioned recursively into sub-modules. Each sub-module then is a set of properties that are even more likely to change together. By "likely to change together", we mean that for many potentially specified members of the family, either all the properties from the set are included, or none. The likeliness increases with the number of family members where this is true.

We propose to *organize requirements by requirements modules*. Those requirements should be grouped together that are likely to change together. It then becomes easier to specify another member of the family explicitly. It is likely that we can take an already specified member, remove one requirements module, and add another requirements module. The latter possibly also has been specified explicitly already for another member.

A criterion for the quality of the organization of the specified requirements modules is how many modules must be changed for obtaining another family member, on the average. These change costs must be weighted with the probability that the change actually occurs.

The above higher-level modules need to be decoupled. For example, there might be family members that interface to a device of kind A, and other family members that interface to a device of kind B. The behaviour of the system is similar for both sets of family members, except for the details of the device.

Our solution is similar to the idea of abstract interfaces in design [24]. We define abstract interfaces in abstraction modules. The advantage of having a device interface requirements module that declares abstract variables and/or events is that the properties of the behaviour modules need to depend only on the stable properties in this module. For example, in telephony it is preferable to base the behaviour on the abstract term of a connection request than on a hook switch being closed by lifting a handset.

We want to have consistent configurations of the requirements document only. When we construct a new configuration, we usually start with an existing, consistent configuration and add and/or remove some properties. One of the difficulties then is to take care of all dependencies among the properties. Further additions and removals may be necessary. Maybe we even need to specify some more properties explicitly. Only then we will arrive at another consistent configuration of properties.

Localizing the changes into one or a few requirements modules helps a lot, but it is not yet sufficient. A property can sometimes depend on other properties from other modules. For example, if entire modules are added or removed, these dependencies must be checked.

A property P_2 *depends* on a property P_1, if P_2 is not well-formed without the presence of P_1. In particular, declarations cause dependencies. For example, P_1

introduces a variable monitored by the system, such as the position of a button. P_2 determines the system behaviour depending on the value of this variable. P_2 would not make sense without the variable being declared.

The dependency relation must be *explicit*. Otherwise, any maintainer not knowing the entire specification by heart must check all requirements for consequences. This is not feasible for large specifications. Therefore, the dependencies should be documented when they are created.

One goal of the explicit dependency hierarchy is that the specifier tries to have as little dependencies as possible. For each dependency, the specifier should check whether it is necessary.

Each property must be formulated such that is a minimal useful increment, as discussed in in Sect. 4.1 above. The dependencies are, in our experience, a good means to check this. A property that depends on many other properties is probably not minimal and could be split up.

The requirements module hierarchy is quite different from the requirements dependency hierarchy. We must take great care to not confuse them. A particular mistake we must avoid is to force the requirements module hierarchy to be the same as the dependency relation. In general, there is not necessarily a correlation between two abstract requirements depending on each other, and being likely to change together. The relationship of requirements modules and requirements dependencies is similar to the relationship of design modules and the design "uses" relation among programs. (See Sect. 3 above.)

Another mistake to avoid is to define the dependency relation between (sub-)modules. The dependency relation is among properties, not among modules. Defining the dependency relation between modules instead of between properties would introduce additional, artificial dependencies.

4.2 Requirements Modules and Features

A *feature* is some increment relative to some baseline, and most features are non-monotonous (see Sect. 2.1). Therefore, a feature consists of a set of added properties and of a set of removed properties. In the language of configuration management [28], a feature is a "change", also called a directed delta. In our particular setting with a set of optional (or mandatory) properties, a feature consists of the set of names of properties that must be included and of the set of names of properties that must be excluded. We may say that a feature is a configuration rule.

A feature is not a requirements module. Many approaches use features as requirements modules. But this creates maintenance problems. Features and requirements modules are similar. Both concepts serve to group properties. But there are two marked differences between features and requirements modules:

1. A requirements module is a set of properties (i.e., *one* set), while a feature consists of both added and removed properties.
2. The properties of a module are selected because of their likeliness to change together, *averaged* over the entire family, while the properties of a feature are selected to fit the marketing needs of a *single situation*.

Forcing requirements modules and features to be the same is not advisable. A feature fits the marketing needs of one occasion only, even though perfectly. It is likely to not fit well for the remaining family members. A requirements module supports the construction of all family members well, even though it does not satisfy all the marketing needs of a particular occasion by itself. A few other requirements modules will be concerned, too. In contrast, adding one more feature on top of a large naively feature-oriented system will concern many other features.

A requirements module provides an abstraction, while a feature is a configuration rule for such abstractions.

An example from telephony is the following:

- The 800 feature allows a company to advertise a single telephone number, e.g., 1-800-123-4567. Dialling this number will connect a customer with the nearest branch, free of charge. This feature should be composed of properties from these three requirements modules: a module that provides addresses for user roles, a module that translates a role address to a device address based on the caller's address, and, entirely independently, a module that charges the callee. The feature removes the property that the caller is charged.
- The emergency call feature allows a person in distress to call a well-known number (911 in the U.S., 110 in Germany and in some other European countries, ...) and be connected with the nearest emergency center. This feature will include the properties from the three requirements modules above, and of a few more. For example, there will be properties from a module that allows the callee to identify the physical line the call comes from.
- The follow-me call forwarding feature allows a person to register with any phone line and receive all calls to his/her personal number there. This feature includes properties from the above module which provides addresses for user roles. The other modules are not needed. Instead, we need properties from a module that translates a role address according to a dynamic user preference. We also need properties from a further module to set user preferences dynamically.

Successful marketing needs features such as the above ones. A "user role address" feature would probably sell much worse than the ubiquitous 800 feature. But the above reuse of requirements modules would not be possible in a naive feature-oriented approach.

Two features might easily be incompatible, i.e., the features interact adversely, because one feature includes a certain property while the other feature excludes it. The features, seen as configuration rules, contradict each other.

A solution is to have different configuration priorities for the properties of a feature. We distinguish (at least) the *essential properties* and the *changeable properties* of a feature. An essential property is necessary to meet the expectations evoked by the feature's name. A changeable property is provided only in order to make the requirements specification complete and predictable for the user. For example, a call forwarding feature can be recognized no matter what

the requirements say on whether a the forwarding target can be set by pressing a button sequence or by, e.g., speech recognition. We therefore propose that the specifier of a feature documents explicitly which properties are essential and which are changeable.

5 Evaluation of Other Work with Respect to Requirements Encapsulation

5.1 Naive Feature Orientation and Requirements Encapsulation Violations

Naive feature orientation supports families of requirements, but does not organize the requirements into requirements modules as discussed above. This leads to feature interaction problems. We now show in our canonical example, the Intelligent Network, where the above encapsulation guidelines for the requirements are violated.

The specification of the Intelligent Network is oriented along execution steps. It is hard to specify a property of the IN without saying a lot about the exact sequencing of steps. The Basic Call Process consists of explicit automata with explicit triggering points, and the Service Independent Building Blocks of a feature are chained together by the explicit sequencing of the Global Service Logic. This violates the principle of making any single requirement as small and abstract as possible, and composing the base system and the features from these atomic properties.

The Service Independent Building Blocks (SIBs) provided in the standard [29] are designed to be general in the sense that they offer a lot of functionality. For example, the Charge SIB performs a special charging treatment for a call, and the Algorithm SIB applies a mathematical algorithm to data to produce a data result. Any details of the operations are controlled by run-time parameters. Any concrete system requirements document must specify which charging or calculating operations these SIBs support, respectively. From then on, it is likely that there will come up another operation not yet supported. This will require a change of the SIB concerned. This in turn threatens to break all other features using this SIB. SIBs therefore are usually not a unit of most abstract requirement.

The Basic Call Process itself violates the principle of making any single requirement as small and abstract as possible. It specifies many different aspects at the same time, as could be seen above. Instead of allowing for small requirements to be taken out and in, a monolithic specification provides hooks for changes of its behaviour. Few properties of its behaviour will be valid for all sets of features. It is hard to design a feature on top of this monolithic base system that will not break for some combination of features. If the base system would consist of smaller, explicitly stated properties with explicitly stated dependencies, then it would be easier to see which features are affected when a new feature removes a certain property.

One example is the step from the two-party call to the n-party session. The Basic Call Process is written in terms of the two-party call. Nevertheless, the Intelligent Network allows to combine several call legs. The n-party call is necessary for such features as Consultation Call, Conference Call, and Call Forwarding. Many features and SIBs are designed with the two-party call in mind, though. For example, the Screen SIB compares a data value against a list. If it is used to specify originating call screening, the screening can fail. Call Forwarding can translate the dialled number several times before making a connection. A single instance of the Screen SIB will check only one of the numbers. Even though the Basic Call Process insinuates that there is exactly one terminating side (Fig. 3), this property is not true for all systems.

The user interface is likely to change, nevertheless its concerns are spread out. This is so despite there being a User Interaction SIB that is intended to perform the user interaction for one feature. Most of the IN features need to interact with a user. This interaction must be possible through a scarce physical interface: twelve buttons, a hook switch, and a few signal tones. Ten of the buttons are used already by the base system. Physical signals must therefore be reused in different modes of operation. But the definitions of several features implicitly assume exclusive access to the user's terminal device. There is no single requirement that specifies the scheme how multiple features coordinate the access. The above interaction between a calling card feature and a voice mail feature is a consequence. Both features assume exclusive access to the "#" button. Details of the user interface are specified at the bottom of the requirements, even though they are likely to change. We discuss this in more detail in [30, 31].

5.2 Approaches with Improved Requirements Encapsulation

There are many approaches that encapsulate requirements better. We now sketch some of them in the light of requirements encapsulation. Even if some of these approaches use the word "feature", mostly they mean a module that encapsulates a secret. However, none distinguishes features and modules explicitly.

The CoRE (Consortium Requirements Engineering) method [32, 33, 34] allows to specify requirements for avionic and other safety-critical systems. A major goal is to plan for change by using information hiding for the requirements. CoRE is based on the functional documentation (four-variable model) approach [35]. It adds additional structure to the requirements document by grouping variables, modes and terms into classes. This borrows from object-orientation. A class has an interface section and an encapsulated section. Entities not needed by other classes are hidden syntactically inside the encapsulated section. The application of CoRE to a Flight Guidance System rendered valuable experience. The authors found that the requirements for the user interface should have been separated from the requirements for the essential nature of the system, since the user interface is more likely to change. Furthermore, they found in particular that planning for change in a single product is not the same as planning for change in a product family [32]. The requirements should have been organized entirely different for the latter.

The Tina initiative (Telecommunication Information Network Architecture) [36, 37], the Race project (Research and technology development in Advanced Communications technologies in Europe), and the Acts project (Advanced Communications Technologies & Services) developed and improved a new service architecture for telecommunications. These projects have added most of the interesting new abstractions explicitly to the resulting architecture. For example, the explicit distinction between a user and a terminal device splits up the host of properties that can be associated with a directory number in the Intelligent Network. Therefore, the requirements of the base system are more structured than for the Intelligent Network. The drawback is that the architecture is quite far away from the structure of current systems, and a transition would be expensive [30].

The DFC (Distributed Feature Composition) virtual architecture is proposed by Jackson and Zave [38]. It is implemented in an experimental IP telecommunication platform called BoxOS [39]. It allows to compose features in a pipe-and-filter network. The filter boxes are relatively simple. This is in accordance with the principle of small requirements. Also, several new abstractions are explicitly supported, for example multi-party sessions and the distinction among users, the different roles they play, and the different terminal devices they may use. A strong point of BoxOS is that it can inter-operate with the existing telephone network. However, part of its functionality is lost for these calls, naturally.

An Agent Architecture is outlined by Zibman et. al. [40]. It separates several concerns explicitly. There are four distinct types of agents: user agents, connection agents, resource agents, and service agents. This separates user and terminal concerns. The terminal resource agent encapsulates the user interface details, such as the signal syntax. The distinct user and connection agents separate call and connection concerns. The user agents bring the session abstraction with them. The connection agents coordinate multiple resource agents. The resource agent separates resource management from both session control and from the services. It was a design goal that the introduction of new services should not require modifications of existing software. Therefore POTS is represented by a single service agent even though POTS really comprises several distinct concerns.

Aphrodite is an agent-based architecture for Private Branch Exchanges that has been implemented recently [41]. Each entity, device and application service is represented as an agent. Agents are therefore abstractions. The often-changing details of the behaviour of an agent are specified as policies. Policies can be changed easily since they are stored as data in a table. It is an explicit goal to make features small. For example, "transfer" is no longer a feature, but made up of three different smaller features: "invoke transfer", "try transfer", and "offer transfer". Another stated goal is to make the assumptions explicit that features make. Also, many new abstractions are already incorporated in the base system as "internal features".

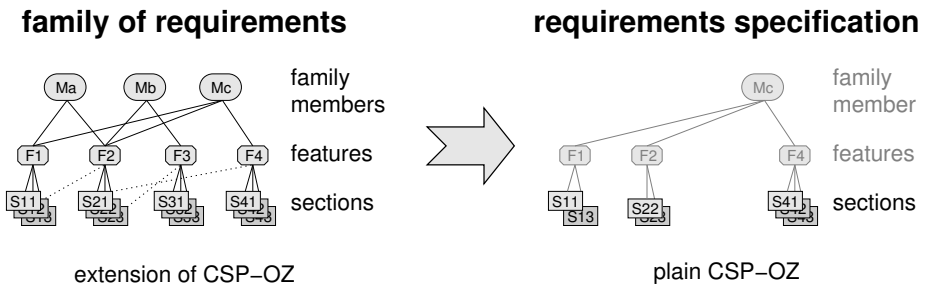


Fig. 4. Generating family members from a family document.

6 An Approach with Families of Rigorous Software Requirements

We have investigated how explicit families of software requirements can facilitate the maintenance task. We showed how the user interface can be encapsulated in a requirements specification of a family of telephone systems [31]. We applied the above requirements encapsulation guidelines in a case study on telephone switching requirements [42, 43].

In [31], we showed how the user interface can be encapsulated in a requirements specification of a family of telephone systems. We proposed to distinguish a syntactic and a semantic level of user interaction. The behavioural requirements should be specified at the semantic level only. Semantic signals should reflect a user's decision to perform some action, or a user's perception that some other user or the system has decided to perform some action. Examples for semantic signals could be "VoiceMailLogin" (for voice mail) and "ReleaseAndReconnect" (for credit card calling). These semantic signals must eventually be mapped to syntactic signals like "flash hook", "#", signal tones, and so on. The mapping should be encapsulated into only one design module, the user interface module. We sketched how such a user interface module could be integrated into the current Intelligent Network architecture.

In [42, 43], we applied the above requirements encapsulation guidelines in a case study on telephone switching requirements. We used a constraint-oriented specification style. All constraints are composed by logical conjunction. We made each constraint as small as usefully possible.

We specified the requirements in the formalism CSP-OZ [44, 45], which we extended by means to specify a family of requirements. All family members are specified in one document. A family member is composed of a list of features. A feature consists of a set of modules and of a list of modules "to remove". A module is represented in CSP-OZ by the formal construct of a section. Each module, i.e., section, holds one abstract requirement. Figure 4 gives an overview. The formalism forces the specifier of any section to state on which other sections it depends. There can be a dependency because the section uses a definition from the other section.

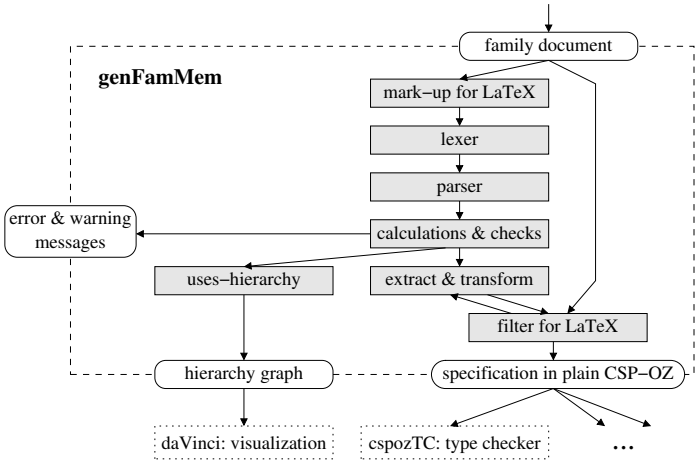


Fig. 5. Data flow structure of the genFamMem tool.

Our formalism also forces the original specifier of a feature to state whether a property is essential for the feature or not. A feature's changeable properties can be removed from the system by another feature through a suitable operator. This allows for non-monotonous changes. But only entire properties, i.e., sections, can be removed and added.

There is a formal semantics both for CSP-OZ [44] and for our additional family construct [46].

We implemented a supporting tool [42, 46]. The tool generates individual family members from the family document as needed, it extracts and displays the dependencies among sections and among features, and it performs type checks on the family constructs. Figure 5 shows its data flow structure. Our formalism imposes some type rules. For example, a section must not be removed if another section from another feature depends on it. Some kinds of feature interactions therefore become type errors. The tool also checks further, heuristic rules that indicate probable feature interaction problems.

We specified the requirements for a telephone switching system in a case study [42, 43]. The case study currently comprises about 40 pages of commented formal specification, with about 50 sections in nine features, including the base system. The communication between the users and the system was specified in terms of semantic signals entirely, as discussed above.

The specification introduces the three notions of "telephone device", "human", and "user role" explicitly and early. As a consequence, the well-known feature interaction problems between call screening and call forwarding vanish. Call screening now appears as two different features: device screening and user screening. Similarly, call forwarding is differentiated into a re-routing when a human moves to another device, and into the transfer of a user role to another

human. All combinations of screening and forwarding now work without adverse interactions.

Nevertheless, our tool found a problem between the two screening features. Its heuristic check issued a warning that both features remove the same section. And indeed, a manual inspection showed that the new constraints introduced by the two features (as a replacement for the removed section) contradicted each other. We then could resolve the issue by specifying explicitly the joined behaviour at this point.

7 Discussion

Our focus on requirements roots in the basic engineering principle of design by documentation. Engineers draw blueprints before construction, and they keep them up-to-date. Accordingly, we document requirements explicitly, including the information necessary for changing them.

The organization of the requirements affects the ease of their maintenance. Feature orientation meets the needs of marketing. But naive feature orientation does not scale. We transferred the information hiding principle from design to requirements. Modules of abstract requirements are a base for families of requirements. Families of requirements are our approach to feature orientation.

We found that a feature is not the same as a requirements module. A requirements module provides an abstraction, while a feature is a configuration rule for such abstractions, chosen for marketing. Without this distinction, it becomes harder to express abstractions with long-term value.

A *policy* is very similar to a feature in the sense that it is a kind of configuration rule (see, e.g., Reiff-Marganiec [47] in this book). The difference is that a feature typically is provisioned statically by a service provider, while a policy is intended to be defined dynamically by a user at run-time. Reiff-Marganiec [47] does not elaborate on the structure of the underlying communications layer of his policy architecture. It would be interesting research to extend our work to dynamically configured policies.

Legacy systems pose a challenge for the application of our ideas on requirements structuring. We proposed concrete improvements for the encapsulation of the user interface in the Intelligent Network [31], see the start of Sect. 6. But a general prerequisite is that rigorous requirements are documented explicitly. Already this can be difficult for a legacy system. However, the current migration to Voice over IP now offers the chance to conceive the requirements for such new systems as a family from the start.

A requirements module is a useful abstraction of the family of requirements. This returns us to our observation at the recent Feature Interaction Workshop (see the end of Sect. 2.2): it is important to consider the abstract purposes, not only the concrete mechanisms. Formulating good common abstractions explicitly is crucial.

Common abstractions need a domain with bounded change. These bounds can be hard to determine in the telephony domain. Take the example of the

UMTS mobile network a few years ago. Everybody in the field would have agreed vigorously that the bandwidth downstream should be much higher than upstream. Today, many envisioned services require a symmetric bandwidth distribution.

A prerequisite for any information hiding approach is our ability to predict the likeliness of changes to some degree. This holds both for information hiding in design and for information hiding in requirements. For requirements, we must put the most stable properties at the bottom of the requirements dependency partial order, and those most likely to change at the top. We don't know how to prepare for completely unanticipated changes.

References

[1] Daniel Amyot and Luigi Logrippo, editors. "Feature Interactions in Telecommunications and Software Systems VII". IOS Press, Amsterdam (June 2003).

[2] Muffy Calder and Evan Magill, editors. "Feature Interactions in Telecommunications and Software Systems VI". IOS Press, Amsterdam (May 2000).

[3] Kristofer Kimbler and L. G. Bouma, editors. "Feature Interactions in Telecommunications and Software Systems V". IOS Press, Amsterdam (September 1998).

[4] Petre Dini, Raouf Boutaba, and Luigi Logrippo, editors. "Feature Interactions in Telecommunication Networks IV". IOS Press, Amsterdam (June 1997).

[5] Kong Eng Cheng and Tadashi Ohta, editors. "Feature Interactions in Telecommunications III". IOS Press, Amsterdam (1995).

[6] L. G. Bouma and Hugo Velthuijsen, editors. "Feature Interactions in Telecommunications Systems". IOS Press, Amsterdam (1994).

[7] ITU-T. "Q.12xx-Series Intelligent Network Recommendations" (2001).

[8] James J. Garrahan, Peter A. Russo, Kenichi Kitami, and Roberto Kung. Intelligent Network overview. *IEEE Commun. Mag.* **31**(3), 30–36 (March 1993).

[9] José M. Duran and John Visser. International standards for Intelligent Networks. *IEEE Commun. Mag.* **30**(2), 34–42 (February 1992).

[10] Stephen Gilmore and Mark Ryan, editors. "Proc. of Workshop on Language Constructs for Describing Features", Glasgow, Scotland (15–16 May 2000). ESPRIT Working Group 23531 – Feature Integration in Requirements Engineering.

[11] Stephen Gilmore and Mark D. Ryan, editors. "Language Constructs for Describing Features". Springer (2001).

[12] Shmuel Katz. A superimposition control construct for distributed systems. *ACM Trans. Prog. Lang. Syst.* **15**(2), 337–356 (April 1993).

[13] ITU-T, Recommendation Q.1203. "Intelligent Network – Global Functional Plane Architecture" (October 1992).

[14] ITU-T, Recommendation Q.1224. "Distributed Functional Plane for Intelligent Network Capability Set 2: Parts 1 to 4" (September 1997).

[15] Hugo Velthuijsen. Issues of non-monotonicity in feature-interaction detection. In Cheng and Ohta [5], pages 31–42.

[16] E. Jane Cameron, Nancy D. Griffeth, Yow-Jian Lin, et al.. A feature interaction benchmark in IN and beyond. In Bouma and Velthuijsen [6], pages 1–23.

[17] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comp. Networks* **41**, 115–141 (2002).
[18] Pamela Zave. Feature disambiguation. In Amyot and Logrippo [1], pages 3–9.
[19] Tom Gray, Ramiro Liscano, Barry Wellman, Anabel Quan-Haase, T. Radhakrishnan, and Yongseok Choi. Context and intent in call processing. In Amyot and Logrippo [1], pages 177–184.
[20] Jan Bredereke. On preventing telephony feature interactions which are shared-control mode confusions. In Amyot and Logrippo [1], pages 159–176.
[21] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15**(12), 1053–1058 (1972). Reprinted in [48].
[22] David M. Weiss. Introduction to [21]. In Hoffman and Weiss [48], pages 143–144.
[23] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Trans. Softw. Eng.* **11**(3), 259–266 (March 1985). Reprinted in [48].
[24] Kathryn Heninger Britton, R. Alan Parker, and David L. Parnas. A procedure for designing abstract interfaces for device interface modules. In "Proc. of the 5th Int'l. Conf. on Software Engineering – ICSE 5", pages 195–204 (March 1981). Reprinted in [48].
[25] David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.* **SE-5**(2), 128–138 (March 1979). Reprinted in [48].
[26] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.* **2**(1), 1–9 (March 1976). Reprinted in [48].
[27] David M. Weiss and Chi Tau Robert Lai. "Software Product Line Engineering – a Family-Based Software Development Process". Addison Wesley Longman (1999).
[28] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Comput. Surv.* **30**(2), 232–282 (June 1998).
[29] ITU-T, Recommendation Q.1223. "Global Functional Plane for Intelligent Network Capability Set 2" (September 1997).
[30] Jan Bredereke. Maintaining telephone switching software requirements. *IEEE Commun. Mag.* **40**(11), 104–109 (November 2002).
[31] Jan Bredereke. Avoiding feature interactions in the users' interface. In Kimbler and Bouma [3], pages 305–317.
[32] Steven P. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In "Second Workshop on Formal Methods in Software Practice", Clearwater Beach, Florida, USA (4–5 March 1998).
[33] Steven P. Miller and Karl F. Hoech. Specifying the mode logic of a flight guidance system in CoRE. Technical Report WP97-2011, Rockwell Collins, Inc., Avionics & Communications, Cedar Rapids, IA 52498 USA (November 1997).
[34] Stuart R. Faulk, Jr. James Kirby, Lisa Finneran, and Assad Moini. Consortium requirements engineering guidebook. Tech. Rep. SPC-92060-CMC, version 01.00.09, Software Productivity Consortium Services Corp., Herndon, Virginia, USA (December 1993).
[35] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Sci. Comput. Programming* **25**(1), 41–61 (October 1995).
[36] Marcel Mampaey and Alban Couturier. Using TINA concepts for IN evolution. *IEEE Commun. Mag.* **38**(6), 94–99 (June 2000).
[37] C. Abarca et al.. Service architecture. Deliverable, TINA-Consortium, URL http://www.tinac.com/ (16 June 1997). Version 5.0.

[38] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Trans. Softw. Eng.* **24**(10), 831–847 (October 1998).

[39] Gregory W. Bond, Eric Cheung, K. Hal Purdy, Pamela Zave, and J. Christopher Ramming. An open architecture for next-generation telecommunication services. *ACM Transactions on Internet Technology* **4**(1) (February 2004). *To appear.*

[40] Israel Zibman et al.. Minimizing feature interactions: an architecture and processing model approach. In Cheng and Ohta [5], pages 65–83.

[41] Debbie Pinard. Reducing the feature interaction problem using an agent-based architecture. In Amyot and Logrippo [1], pages 13–22.

[42] Jan Bredereke. A tool for generating specifications from a family of formal requirements. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, "Formal Techniques for Networked and Distributed Systems", pages 319–334. Kluwer Academic Publishers (August 2001).

[43] Jan Bredereke. Families of formal requirements in telephone switching. In Calder and Magill [2], pages 257–273.

[44] Clemens Fischer. Combination and implementation of processes and data: from CSP-OZ to Java. PhD thesis, report of the Comp. Sce. dept. 2/2000, University of Oldenburg, Oldenburg, Germany (April 2000).

[45] Clemens Fischer. CSP-OZ: a combination of Object-Z and CSP. In Howard Bowman and John Derrick, editors, "Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)", volume 2, pages 423–438. Chapman & Hall (July 1997).

[46] Jan Bredereke. "genFamMem 2.0 Manual – a Specification Generator and Type Checker for Families of Formal Requirements". University of Bremen (October 2000). URL http://www.tzi.de/~brederek/genFamMem/.

[47] Stephan Reiff-Marganiec. Policies: Giving users control over calls. In this book.

[48] Daniel M. Hoffman and David M. Weiss, editors. "Software Fundamentals – Collected Papers by David L. Parnas". Addison-Wesley (March 2001).