Families of Formal Requirements in Telephone Switching*

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

Abstract.
We introduce to the idea of families of requirements documents. Requirements families are sets of requirements whose common properties are so extensive that it is advantageous to study the common properties in the requirements before analyzing individual members. We present how we can apply this idea with a formal description technique, how our approach helps to avoid feature interaction problems in telephone switching, and how it supports the detection of remaining problems.

1 Introduction

1.1 Feature Interaction Problems

Digital telephone switching systems already comprise hundreds of features, and the market forces the providers to incorporate an ever increasing number of new services and features into the switches. In recent years, *feature interaction (FI) problems* have become a serious obstacle to adding more features to these systems. Adding one more feature, even when it behaves as desired for itself, may cause undesired behaviour of other features, even when these behave as desired for themselves. The number of possible feature combinations has become so large that it is not possible anymore to check all combinations manually.

One standard example of such a feature interaction occurs between the Terminating Call Screening (TCS) and the Call Forwarding (CF) feature (Fig. 1). Terminating Call Screening allows a subscriber C to specify a list of callers by whom he never wants to be called. Call Forwarding allows a subscriber B to specify another destination to which all calls to B should be forwarded. Now suppose that C has subscribed to TCS, and has put A on his screening list. Furthermore suppose that B has subscribed to CF and forwards all calls to C. Finally, A happens to attempt to call B. Will the phone of C ring? Should it ring? This depends very much on *how we define the notion of "caller"* for the two features. When CF consists of two calls "glued" together, then B is the caller for C, the screening condition is not satisfied, and the phone will ring. When we take into account that C probably does not want to be disturbed by A, the phone should not ring. Zave discussed such "secrets of call forwarding" in depth [1]. We will come back to this example later, too.

*This work was partially funded by the Leibniz Programme of the German Research Council (DFG) under grant Ol 98/1-1.

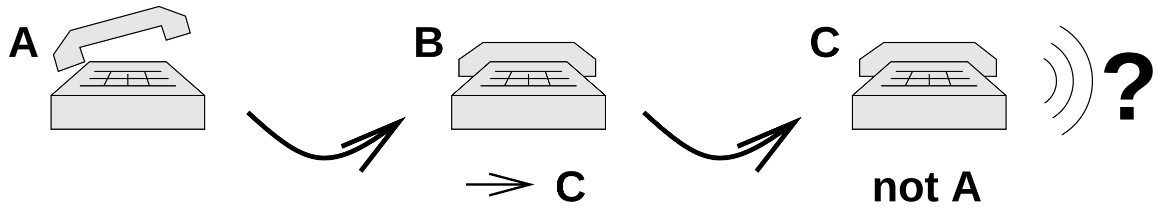

Figure 1: Feature interaction between Terminating Call Screening and Call Forwarding.

1.2 Requirements

The point we want to make at the above example is that feature interactions already arise in the requirements documents. When these documents are a *complete* description of the behaviours (and of other interesting properties), then even *all* feature interaction problems are at least inherently present in the requirements documents. Therefore, they should be tackled already there. (Of course, we still have to ensure that any implementation conforms to its requirements. But this is a different question from that of the interaction of features which work perfectly well on their own.)

The notion of complete requirements leads us to formal, i.e., precise, requirements. Considerable work about formal requirements specifications exists. But an aspect of them that deserves more attention is the maintenance of formal requirements documents. How can we handle many changes to them over time, and how can we handle a large number of variants for these documents when different customers of a successful product have different needs?

In Section 2, we introduce to the idea of families of requirements documents. In Section 3, we introduce to the formal description technique that we use to explore this idea, and in Section 4, we show how we can specify a family of telephone switching requirements. We discuss the advantages of this approach for the feature interaction avoidance and detection in Sections 5 and 6. Section 7 discusses some related work and Section 8 concludes the paper with a summary.

2 Families of Requirements

When we develop a (single) program, we take a requirements document, make design decisions and write them down, for example in the form of program code, and finally arrive at the complete program. The visible behaviour described by the documents remains the same (except maybe for a reduction of the internal non-determinism). This assumes that the requirements document indeed was complete, and that accordingly the implementer is not concerned with guessing the details of the customer's wishes.[1] We call adding such design decisions an *internal refinement* of the system. We distinguish it from an *external refinement* of a requirements specification, which adds more desired properties (i.e., features) to it, and which renders a requirements specification, again.

Sometimes, we want to have variations of the program. Either we need to make some design decisions differently (for example in order to buy different hardware), or we need to withdraw or modify some of the extensions we made to the requirements

[1] Of course, in practice the requirements are often changing already before the first implementation phase has been completed. This is a maintenance problem, as discussed below, and does not contradict the assumption of a complete requirements specification.

or add further extensions (for example think of different sets of features for different customers).

For internal refinement, Parnas's idea of *program families* offers relief. Parnas [2] defines program families as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members. He proposes to keep and document the results of the intermediate (internal) refinement steps of the first program and to use them as starting points for further programs. Thus, the family of programs becomes the set of leaves of a refinement tree. Furthermore, he proposes that design decisions that might be changed easily are postponed by using the concept of information hiding modules. An information hiding module is the precise description of the behaviour of a part of a program, without revealing how this behaviour is achieved. Thus, the module encapsulates a design decision as a secret, and the rest of the system cannot be affected by changing this decision later. When we make different design decisions inside a module for different programs, we get different branches in the refinement tree.

The above approach is intended for the *design* of software, but we can apply it to the maintenance of software *requirements*, too. When dealing with telephone switching systems, we are concerned with

> *requirements families*, which are sets of requirements whose common properties are so extensive that it is advantageous to study the common properties in the requirements before analyzing individual members.

As with families of programs, it is crucial to identify early what are the common parts of the family, on which most other parts will depend. Accordingly, we have to do an *SCV analysis* (scope, commonality, and variability) of the family, as proposed by Weiss and others [3, 4, 5] for families of programs: we do not already specify any conceivable telephone switching system, but we determine which requirements aspects will be common to all conceivable systems, which aspects will be common to only some versions, and which aspects are highly special to the particular system we want to build next. Furthermore, we find out (as far as possible) which requirements aspects are likely to change together.

When performing internal refinement for a family of programs, one encapsulates those implementation details into one information hiding module that are likely to change together; and one first makes those implementation decisions that are most unlikely to change. When performing external refinement for a family of requirements, we encapsulate those requirements variant choices into one module that are likely to change together; and we first state those requirements that are most unlikely to change. Different variants become different sub-nodes in the tree of requirements options. The secret of any requirements module is the choice of a particular variant set from the variant options inside it.

3 Overview of Formalism: CSP-OZ

In order to specify requirements precisely, we need a notation with a formal semantics. It must be suitable for specifying pure requirements, without forcing the specifier to already make any implementation decisions. Thus, the requirements document must be formulated entirely in terms of the environment of the system that is going to be constructed. In previous work [6, 7], we used the Functional Documentation approach [8, 9] to requirements specification, also known as "Parnas tables". This formalism

telNums : $\mathbb{P}(\mathbb{N})$	[Declare the (fixed) set of valid phone numbers. . .]
telNums $= \{1, 2, 3\}$	[. . . and specify a concrete set.]

SimplePhone

chan offHook chan dial : [$n?$: telNums] chan onHook [Input events.]
chan connect : [$n!$: telNums] chan disconnect [Output events.]
main = offHook $\rightarrow$ dial $\rightarrow$ connect $\rightarrow$ onHook $\rightarrow$ disconnect $\rightarrow$ main

dialledNum : $\mathbb{N}$	[Variable that remembers the last number dialled.]
dialledNum $\in \{0, 1, 2, 3\}$	[An invariant constraint on it.]

Init
dialledNum $= 0$

enable_dial
$n?$: telNums
true

effect_dial
$n?$: telNums
dialledNum$' = n?$

enable_connect
$n!$: telNums
dialledNum $\in \{1, 2, 3\}$

effect_connect
$n!$: telNums
$n! =$ dialledNum

Figure 2: A CSP-OZ example.

satisfies the above conditions, and additionally it provides a tabular kind of notation that is both compact and readable, even to customers (with some technical background).

In order to investigate the family idea, we need the notion of refinement. Since we did not want to re-invent it, we now use the formalism CSP-OZ, which already provides a well worked-out notion of refinement (but not the readability-improving tabular notation). CSP-OZ [10, 11, 12] is based on the process algebra CSP (Communicating Sequential Processes) [13, 14] and therefore allows to specify the control and communication aspects of a distributed system. The CSP part is combined with Object-Z [15, 16], an object-based extension to Z [17, 18]. Object-Z and Z are a formalization of set theory, and they are particularly suitable to specify data aspects of a system. For example, they provide the notions of abstract data types and state invariants which are not present in CSP.

In future work, we plan to investigate the use of SDL [19], too, due to its widespread use in the telecommunications community. Currently, we postponed this plan because it would require considerable work on, e.g., a suitable notion of refinement for SDL.

CSP specifications are defined over events. For example the "main" line in Figure 2 describes a process that repeatedly does the five events named there. Processes can be combined by operators like parallel composition and hiding. The semantics of a system consists of its traces, failures, and divergences. Numerous algebraic laws for behaviour refinement have been derived.

An Object-Z class consists of the specification of a state space and operations on this state space. (Multiple) inheritance of classes is available. Numerous algebraic laws for data refinement have been derived. In class SimplePhone in Figure 2, dialledNum is declared as a variable ranging over the natural numbers, with the additional constraint that it always must be between 0 and 3. In the Object-Z view, the operation named effect_dial has "$n?$" as a parameter and sets dialledNum. The Object-Z operation named enable_dial does not change any variable, for reasons discussed in the following.

In the semantics of CSP-OZ, the operations of the Object-Z part are interpreted as CSP events and impose additional constraints on the CSP behaviour. As an extension to Object-Z, CSP-OZ allows to mark operations as either controlled by the system or as monitored by the system only. Monitored events have "true" as the enabling condition.[2] Thus it is possible to distinguish between requirements of the system and requirements of its environment. In Figure 2, there are two operations called enable_connect and effect_connect in the Object-Z part. The former further restricts the conditions in the CSP part about when a connect event may happen. Now, the variable dialledNum additionally must have the right value. (Actually, we can prove that this constraint is always fulfilled since the dial event must happen first, setting the variable appropriately.) The decoration "?" denotes an input parameter, "!" denotes an output parameter, and "′" denotes the after-state. The variable dialledNum is not part of the semantics of this specification, it is merely an auxiliary construct to restrict the range of output values associated with the connect event. The semantics of CSP-OZ is defined entirely in terms of (parameterized) events.

There is tool support available for CSP, in particular there is the powerful, commercial model checker FDR (Failures-Divergence Refinement) [20]. There will be a translation tool for CSP-OZ to FDR until the end of June 2000 [21, 22], and there exists the type checking tool cspozTC [23] for CSP-OZ. (We used it to type check the specification in Figure 2 and of course that in our case study.)

4 Specifying a Family of Requirements

We now present an approach for how to structure a document that comprises a family of formal requirements for telephone switching. We use a simplified example that we adapted from case studies which we have performed with different formalisms [24, 25].

Figure 3 shows the chapters of the requirements family document. The document does not contain the requirements for one complete system, but for all the complete systems SpecificationA, SpecificationB, SpecificationC, SpecificationD, ... Each is described in a separate chapter. These chapters do not contain the entire specification, instead they just reference the features which they use, in a way described below. Each feature is described in a separate chapter, too, it encapsulates the details of one requirements variant.

The name of a chapter of a complete system is prefixed with the keyword `familymember`, and the name of a feature's chapter is prefixed with the keyword `feature`. These keywords are part of the formal specification, they serve to identify formally and to extract the specific complete system, or family member, that we want to build next, with exactly the right set of features.

The document in Figure 3 starts out with the specification of three features, and then composes them to a first, simple complete specification A. The document goes on

[2]And they may not be refused at any time by the CSP part, which is the case for *no* event in our mini-example due to space constraints. We omitted the necessary explicit asynchronous input buffer process.

1.	Introduction
2.	`feature` UserSpace
3.	`feature` BasicConnection
4.	`feature` VoiceChannel
5.	`familymember` SpecificationA
6.	`feature` ScreeningBase
7.	`feature` BlackListOfDevices
8.	`familymember` SpecificationB
9.	`feature` BlackListOfUsers
10.	`feature` FollowHumanConnectionForwarding
11.	`familymember` SpecificationC
12.	`feature` TransferUserRoleToAnotherHuman
13.	`familymember` SpecificationD
⋮	⋮
	Indices / Bibliography

Figure 3: The chapters of a requirements document.

with two more features and a composition of all these features into a complete specification B. The same pattern goes on, with the exception that each complete specification usually comprises only some of the features defined previously. For example, specification D does not include the screening features. Please note how different requirements specifications share common parts by referencing the same chapters. (We will show below how we do this precisely.)

4.1 The Features in the Case Study

In this section we briefly sketch the features whose names appear in Figure 3.

The UserSpace feature introduces the basic notions of *devices*, *humans*, and *user roles*. We distinguish these concepts, in particular the latter two. For example, John Smith, sitting in his office near his phone, is a human, as well as Mary Taylor in the office next door. The service hot-line of their company is a user (role). This user role can be played by John. During his lunchtime, the user role can be played by Mary. During peak times, both may play this user role.

The BasicConnection feature introduces the notion of a *connection* between two users, and the VoiceChannel feature associates a *voice channel* to a connection. The next chapter comprises the composition of all these features to a complete specification A. This is the first member of the family of requirements.

The ScreeningBase feature introduces the notion that connections may be prevented by *screening*. Building on this, the BlackListOfDevices feature allows to set up a list for each telephone device of other devices that will be prevented to successfully establish a connection. The BlackListOfUsers feature is similar, but works for users instead for devices.

The FollowHumanConnectionForwarding feature allows a human to move from one telephone device to another while continuously playing her associated user roles. The effect is that she can be reached at whichever device she currently is located. The TransferUserRoleToAnotherHuman feature allows a human to transfer one of her user roles to another human. If a transfer is done during a connection, the new human inherits the connection with all rights and obligations, possibly he becomes even the originating side of the connection.

4.2 Incremental Specification

Telephone switching requirements are special compared to other software requirements in that they have *very few events* at the user interface, compared to the functionality they provide. A typical telephone handset has twelve buttons, a hook switch, and can generate a few signalling tones like the dial tone or the busy tone. Similarly, the current mode of a connection (or "call state") is relevant for the behaviour of most features.[3] Thus it is not possible to slice a system into independent features by the controlled events, as it has been done successfully for other kinds of software systems.

Therefore, we instead impose partial, self-contained constraints incrementally on the same few events and modes, and thus use a *constraint-oriented specification style.* Each constraint describes some desired property, and a composition by a kind of logical conjunction ensures that adding one more (invariance) property cannot invalidate other (invariance) properties.

Figure 4 shows some more details of the BasicConnection feature chapter mentioned in Figure 3. The CSP-OZ class ConEssUserAvailable[4] introduces the notion that a user v either can be busy, with respect to some other user u, or that u can engage v in a new connection. The CSP-OZ class UserResSimple[5] then states the conditions when exactly this is the case (the actual, complex expression is left out for brevity here). The class employs the `inherit` construct of CSP-OZ to include all declarations and predicates of the class ConEssUserAvailable. The declarations are concatenated[6], and the predicates are composed by a logical conjunction. In this simple case, the parent class provides a type declaration, the declaration of userAvailable as a variable, and some type restrictions on it. The class UserResSimple then adds further constraints to the inherited variable userAvailable.

We use the CSP-OZ `inherit` construct in this restricted fashion only, and ignore the more powerful replacement operations that it offers, in order to keep the monotonicity of the logical conjunction.

The class (template) BasicConnectionSys collects all properties that we defined for (the system part of) the BasicConnection feature. By default, the `default_properties` construct has exactly the same semantics as the `inherit` construct, thus we use the `inherit` and the `default_properties` construct together to conjoin all classes, i.e., properties, that are defined for this feature. The semantics of the `default_properties` construct changes as soon as another feature *removes* one or more of the properties. An example will illustrate this.

Figure 5 presents a part of another feature, the ScreeningBase feature, which relies on some of the definitions and properties of the BasicConnection feature. Formally, we express this by the `inherit` construct, again. The property ConScreenEss inherits

[3] The situation can be improved when we distinguish between syntactic and semantic events [26], and map a larger number of semantic events to the few syntactic ones. Nevertheless, the problem with the commonly used call state remains.

[4] "Con" stands for "connection". "Ess" stands for "essential", as discussed below in Section 4.3.

[5] "Res" stands for "resources".

[6] CSP-OZ requires that all inherited declarations have distinct identifiers, in order to avoid name clashes.

...

3. `feature` **BasicConnection**

...

ConEssUserAvailable

`inherit` ...

userAvailableType ::= userAvail | userNotAvail

$userAvailable : userIds \times userIds \rightarrow userAvailableType$

...

UserResSimple

`inherit` ConEssUserAvailable ...

$\forall\, u, v : users \bullet$
$\quad (userAvailable(u, v) = userNotAvail) \Leftrightarrow \ldots$

...

BasicConnectionSys

`inherit` ... ConEssUserAvailable ...
`default_properties` ... UserResSimple

...

Figure 4: Incremental constraints on the variables.

the property ConEssUserAvailable and then adds its own constraints on top of it about when a user should never be available. It does *not* inherit the property UserResSimple, which imposed similar, but incompatible, constraints. Instead, the feature *removes* the property UserResSimple. The next property UserResScreen then puts a new precise constraint on the variable denoting when a user is available, depending on the current screening mode. The third property ConNoScreen states that screening never happens. (Because the different actual screening schemes will be introduced by further features.)

When we compose all features into a complete system, the according construct takes care that the explicitly removed property UserResSimple is disregarded in the BasicConnection feature, even though it would be included by default through the `default_properties` construct. Figure 6 shows in detail how a complete requirements specification is composed of features, using the `features` construct. Such a chapter is marked by the `familymember` construct, and then must start with a `features` construct that selects the features to include and that processes the `remove` constructs, as described below. We perform the actual composition of the properties of this family member separately; Figure 6 shows how we employ the usual `inherit` construct to compose (the system part of) the complete requirements specification.

The seven constructs `feature`, `feature_end`, `familymember`, `familymember_end`, `default_properties`, `remove`, and `features` are an extension to plain CSP-OZ, for

...

6. `feature` **ScreeningBase**

ConScreenEss

`inherit` ConEssUserAvailable ...

$\mathsf{connectionScreenType} ::= \mathsf{connectionScreenOk} \mid \mathsf{connectionScreenBlock}$

$\mathsf{connectionScreen} : \mathsf{userIds} \times \mathsf{userIds} \to \mathsf{connectionScreenType}$

$\forall u, v : \mathsf{users} \bullet$
$(\mathsf{connectionScreen}(u, v) = \mathsf{connectionScreenBlock})$
$\Rightarrow (\mathsf{userAvailable}(u, v) = \mathsf{userNotAvail})$

`remove` UserResSimple

UserResScreen

`inherit` ConScreenEss

$\forall u, v : \mathsf{users} \bullet$
$(\mathsf{userAvailable}(u, v) = \mathsf{userNotAvail})$
$\Leftrightarrow (\ldots \vee (\mathsf{connectionScreen}(u, v) = \mathsf{connectionScreenBlock}))$

ConNoScreen

`inherit` ConScreenEss

$\forall u, v : \mathsf{users} \bullet$
$(\mathsf{connectionScreen}(u, v) = \mathsf{connectionScreenOk})$

...

ScreeningBaseSys

`inherit` ConScreenEss
`default_properties` UserResScreen ConNoScreen ...

...

Figure 5: Changing some constraints.

the purpose of specifying families of requirements. Their semantics are defined as a syntactic preprocessing step on the specification document. The constructs `feature` and `feature_end` mark the start and end of one feature, and the constructs `familymember` and `familymember_end` mark the start and end of one complete specification. They do not nest. The end marks are optional and may be substituted by the next start mark of either kind. The start marks may be preceded by some chapter numbering, and they are supposed to be typeset as a chapter heading. We wrote a preprocessor that

...

8. `familymember` **SpecificationB**

```
features
    UserSpace
    BasicConnection
    VoiceChannel
    ScreeningBase
    BlackListOfDevices
```

```
SpecificationBSys
inherit
    UserSpaceSys
    BasicConnectionSys
    VoiceChannelSys
    ScreeningBaseSys
    BlackListOfDevicesSys
```

...

Figure 6: Composing a complete specification from features.

generates for a specified `familymember` chapter a corresponding complete specification in plain CSP-OZ, including a customized version of the features used. The `features` construct in this chapter determines the features which are included. The generated requirements document then can be given to an implementer for this specific system.

In the preprocessing step, a `default_properties` construct which names multiple properties is equivalent to an according number of `default_properties` constructs with just one property. The preprocessing step translates a `default_properties` construct with one property either to an `inherit` construct or to nothing. The latter is exactly then the case when in the features referenced by the `features` constructs there is any `remove` construct that names the same property.

Similar as with the `inherit` construct, a `remove` construct, too, must name a class that is defined in the current subset of text referenced by the `features` constructs. A class definition must precede any `remove` construct naming it. A `remove` construct may appear inside a `feature` chapter only, and only outside of any class. Any class named in a `remove` construct must not be named in an `inherit` construct (and thus it may be inherited only via a `default_properties` construct). This ensures that any `remove` operation will completely remove any reference to this class.

4.3 Documenting What is Essential

A feature usually consists of several properties, as for example the BasicConnection feature in Figure 4. Some of them are essential to this feature, and some are just defaults that need to be provided in order to make the behaviour description complete. In the example, the notion of userAvailable is essential to a connection, but the actual condition for user availability, that the user is not already connected to some other user, is not essential. Accordingly, this condition is removed and replaced by the ScreeningBase feature. In turn, the ScreeningBase feature specifies in the property ConNoScreen that screening never happens, and this again is replaced by any of the actual screening features.

When a specifier employs the `remove` construct, he must make sure that the removed property was not essential to the feature it belonged to. Unfortunately, that feature probably was written by someone else and/or at a different time, such that the specifier first needs to understand that feature, too. This is costly and error prone. Therefore, we plead that the original specifier of a feature shall document which of its properties are essential to the feature, and which are not. If anybody, he is the one to know.

Syntactically, we document this information in the class that composes all properties of one feature; in Figure 4 this is BasicConnectionSys. In this class, we use the `inherit` construct for the essential properties, and we use the `default_properties` construct for the changeable properties. In the example, the property ConEssUserAvailable is essential, and the property UserResSimple is changeable.

The syntactic distinction allows us to perform automated syntactic checks whether only changeable properties are removed. As stated in Section 4.2, the `remove` construct may name only classes that are referenced by a `default_properties` construct, and not by an `inherit` construct.

5 Support for FI Avoidance

The goal of the requirements family approach is to avoid feature interaction problems already in the requirements documents. The general idea is to modularize the requirements documents in the information hiding sense. Besides this methodological approach, we have some language support, too. We presented already the explicit inclusion of referenced modules, and the syntactic markup of essential properties of a feature. We discuss further support in this section.

5.1 Further Syntactic Support

In Section 4.2, we demanded that any class named in a `remove` construct must not be named in a `inherit` construct. An automated type check reveals this. Furthermore, the type check performed by our preprocessor already issues a warning when a class is not (yet) named in a `remove` construct, but is named somewhere in a `default_properties` construct, thus marked as changeable, and named in an `inherit` construct, thus marked as essential.

The `remove` construct renders further support against unintended side effects. When a class B inherits a class A, and a class C removes the class A, but when the class B still belongs to the generated CSP-OZ specification, then we have a syntactic error that is flagged already by the type check of the preprocessor tool. Accordingly, when a specifier decides to remove class A (and maybe to replace it by a different class D), then he gets feedback from the tool on whether he has overlooked B.

Similar support exists for detecting an insufficient feature list. The `features` construct lists the features of a `familymember`. We define it such that it must contain explicitly the entire list of features. If there is an `inherit`, `default_properties`, or `remove` reference to a property of a feature not selected, this is a syntactic error, again, that is checked automatically. There is no implicit inclusion via the latter three constructs. This helps the specifier to understand the dependences between the features by automated warnings that are issued when he overlooks a dependence.

5.2 Documenting the Uses Hierarchy

The gist of the notion of "feature" is that it is defined relative to something else. Therefore, it can only be understood based on the definitions that it uses from elsewhere. When such definitions are changed, this potentially affects all features that use it. Performing such changes is difficult since one must be sure to find all features affected, and to understand the consequences of the changes.

Our approach supports such a task by syntactic means. The declarations and predicates of a class A can be relevant to a class B only if B explicitly inherits A. Thus we get a directed acyclic graph of the inherit relation for each complete specification that we generate. If we replace every `default_properties` construct by an `inherit` construct and if we disregard all `remove` constructs for a moment, we get the maximal graph of potential dependence for any family member. We call it the *uses hierarchy* (in the spirit of [27]). This uses hierarchy is extracted from the requirements document by an automated cross-referencing tool, in order to support the maintainers.

When writing a feature, it is important to keep the uses hierarchy graph as loosely connected and as bushy as possible, by stating minimal dependences only, in order to facilitate later modifications. An example in the case study is that the screening features are defined on top of the BasicConnection feature, and *not* on top of the VoiceChannel feature. No changes to the VoiceChannel feature will require us to revisit the screening features. We could even replace the voice channel by a video channel. When we started our work, we wrote a draft with a monolithic voice connection feature. But in a careful analysis we factored out the BasicConnection feature (which in itself is separated into further properties, not shown in the examples in this paper). A general rule of thumb for tending the uses hierarchy graph is to introduce one concept only per feature, and in particular to separate the introduction of a new concept from its first use. Another example for this in our case study is the ScreeningBase feature with the two separate actual screening features.

5.3 Explicit Change Interface

Many approaches provide a way to add features to a base system in some way. In particular operational approaches usually allow more or less arbitrary changes to the base system. An example is the Detection Point (DP) concept in the Intelligent Network standard. The extended processing starting at a DP may have global effects: it can modify arbitrary data, and it can terminate by jumping back to an arbitrary DP.[7] This renders sufficient expressive power to introduce any desired feature. But we argue that this approach also makes it very hard to understand the consequences of changes.

> We argue that *the system to modify should have an explicit interface* where it can be modified.

This is the case in our approach. Each feature has a list of default properties which may be removed and replaced. It is syntactically not allowed to touch the essential properties of a feature, and any default property may be removed in its entirety only. This prevents changes that are too fine-grained. Allowing to make a "small change"

[7]Since Capability Set 2, for each triggered DP the DP to return to can only be one of a fixed list.

inside one property would increase the complexity significantly that the specifier has to manage. The idea of having "properties" is that they encapsulate some requirements detail, and they should be chosen such that one either uses them entirely or not at all. Then, the specifier need not understand the encapsulated details in order to use (or not use) the property.

5.4 FI Avoidance Example in the Case Study

We now present an example where we avoided feature interaction problems in our case study.

In Section 4.1, we introduced to the two connection forwarding features FollowHumanConnectionForwarding and TransferUserRoleToAnotherHuman. The former feature allows a human to move from one telephone device to another while continuously playing her associated user roles. The effect is that she can be reached at whichever device she currently is located. The formal specification consists of two parts. First, it removes a default property (DevAssocNoRem) of the UserSpace feature that states that there is a fixed association of telephone devices and humans during the lifetime of the system. Second, it adds a similar, but substantially weaker (default) property that states that such associations cannot change only while the humans concerned are involved in a connection.

The TransferUserRoleToAnotherHuman feature allows a human to transfer one of her user roles to another human. If a transfer is done during a connection, the new human inherits the connection with all rights and obligations. Again, the feature removes a default property (UserRolNoRem) of the UserSpace feature, which states that there is a fixed association of user roles and humans during the lifetime of the system. Several new default properties state that user roles can be transferred only when a connection exists, and how this is done.

The definitions of both features employ the concepts of a telephone device, a human, and a user role, which we introduced already in the UserSpace feature, even though the default properties of the UserSpace feature state that there is a fixed association between the members of all three of them, as in POTS.

As a first point, we note that both features do not mutually interact. On a syntactic level, both replace different default properties of the UserSpace feature. On a semantic level, the first is concerned with the association of devices and humans, while the second is concerned with a different association, that of user roles and humans. This is due to the careful introduction of the concepts of human and user role in the beginning.

Since we did not plan to specify a base system only, *we planned ahead for the entire family* and identified these concerns that would become important to possible future features. The actual *initial overhead* in the specification consists in stating the separate concepts explicitly, in stating that they are identified by default for now, and in using the right ones of them for defining further features.

For a second point, we look at what has become of the feature interaction problems between Terminating Call Screening (TCS) and Call Forwarding (CF), presented in the introduction. In our case study, we have not only two forwarding features, but also two screening features, BlackListOfDevices and BlackListOfUsers. Both screening features clearly state whether they screen calling devices or calling user roles. A subscriber of BlackListOfUsers will be protected against connections from all screened users from all devices, at the device where she is registered currently.[8] A subscriber of

[8]The distinction of humans and devices is not easy to implement: it requires to enforce the correct

BlackListOfDevices will not expect to be protected from either connections forwarded from other devices, nor from humans that are able to use different devices. Therefore, all the feature interaction problems between TCS and CF, as presented in the introduction, do not arise in our case study. We just don't have the ambiguous notion of "caller".

6 Detection of Remaining Feature Interaction Problems

Even though our approach supports the specifiers to avoid many feature interaction problems, some will always be left in a practical, real-sized system. Therefore we need feature interaction detection, too. Any detection method applicable to a requirements document like ours should be applicable. Besides this, our approach provides the means for some specific checks for remaining interactions.

Syntactically, our formalism allows that two `remove` constructs from different features remove the same property. But an extended automated type check is helpful for the specifier which flags such situations as a warning. When a feature removes some property of another feature, then it usually adds a new property as a replacement. Two such replacements are conjoined, by the usual conjunction semantics, but there is some chance that they are contradictory, and that the resulting complete specification has an empty set of allowed behaviours. Therefore, the specifier should look at such situations immediately as they arise.

Such a case of double removal occurs in a case study of ours. Based on the ScreeningBase feature, we define two different screening features BlackListOfDevices and BlackListOfUsers independently. Of course, both remove the same changeable property ConNoScreen (shown in Figure 5). They are the respective essential properties of the two screening features. A manual analysis shows immediately that the two replacing, simple properties are not contradictory. But a further look reveals that both features have a changeable property each that states that its respective feature is the only screening feature, such that otherwise no screening happens. This feature interaction is resolved easily, since both conflicting properties are marked as changeable, and since the necessary replacement property is obvious. We specify this replacement property as just another feature which we select whenever both above features are selected for inclusion into a system.

Potentially contradictory specifications are a general problem in a constraint-oriented specification style. This is the reason why our formalism allows to express invariance properties only, and no required progress. General progress properties are known to be not compositional in the way we need it. Their composition is still an area of research, in particular for real-time properties.

If there exists any initial state for a specification written in our formalism that satisfies all specified properties, then there also must be some system behaviour that satisfies these properties. (The simplest is that where just nothing happens.) In this case, we can prove the absence of an empty set of behaviours by determining some valid initial state.

Some restricted progress property checks are nevertheless feasible by model checking tools: we can prove by exhaustive state space exploration that no deadlocks occur, i.e.,

registration of humans at devices. Assuming that everybody stays at home may not be enough. Furthermore, allowing two users to register at the same device simultaneously requires at least something like two different ring tones.

that always at least some progress is possible. Such checks depend on a limited size of the state space, in order to avoid the state space explosion problem.

7 Related Work

Related work investigates "feature" constructs for different specification and programming languages, in particular in the context of the FIREworks project [28]. The aim of this project is to define a feature construct for several languages, and then to evaluate it in case studies. The notion of *feature-oriented programming* shall emphasize the importance of features in the system structure. The approach bases on the superimposition idea of Katz [29], where we specify a base system first, and where we then specify textual increments by a suitable operator, for example as done by Plath and Ryan [30]. One problem of the superimposition idea is that, even though the textual increments have a defined interface, the base system does not. Therefore, a superimposition can invalidate arbitrary assumptions about the base system and thus cause feature interaction problems. This effect is known sufficiently from the Intelligent Network, of which the Detection Points together have about the same potential for modification.

There are many approaches in the literature that allow to extend a system through (external) refinement in a restricted way, in order to preserve certain desired properties of the system. For example, removing traces only is an operation that preserves all invariance properties, and adding (deadlock-free) traces only is an operation that preserves the absence of deadlocks. Unfortunately, such severe restrictions are often not practical in our application area. Now, our distinction of essential and changeable requirements allows for more realistic kinds of changes while still preserving those invariance properties marked as essential.

8 Summary

We introduced to the idea of families of requirements documents, which we deem particularly suitable for the application area of telephone switching. Successful switches are long-lived, and during a long period of customizations and extensions, many family members are created that still have extensive common properties, but are not exactly alike.

The idea of families allows us to keep all requirements variants together, and to encapsulate variant choices. It enables *information hiding* for requirements documents, similar to how it is known for programs. Furthermore, it *encourages to plan ahead* for future members of the family, as we have seen in the above feature interaction avoidance example from our case study.

We presented how we can apply this idea with the formal description technique CSP-OZ, extended by some constructs that structure the family's features and compose them to family members. We support such modularization by syntactic restrictions. For example, a feature has an explicit interface for change, preventing arbitrary fine-grained changes and the entailing difficulties in understanding their consequences. These syntactic restrictions are checked by automated tools.

Another core idea is to let the original specifier *document explicitly* as much of the information as possible that another specifier of an incremental feature will need; Examples are the explicit, syntactic distinction between essential and changeable properties of a feature, the explicit uses hierarchy, and the change interface. The specifier of an incremental feature can use this explicit information either for avoiding feature interaction problems, or for at least detecting feature interaction problems. For example, the explicit uses hierarchy can serve as a cheap *syntactic filter*. When one feature removes a parent's default property that is still required by another feature, this is a type error. Another syntactic rule detects potentially contradictory properties and thus identifies interaction-prone feature combinations.

Further information that is explicit in our approach, but not discussed in this paper, is the distinction between the requirements for the system and for its environment, the distinction between monitored and controlled events, and the extraction of common "modes" from the definition of controlled events.

Finally, we stress some methodological aspects which help to avoid feature interaction problems. In long-lived systems, such as telephone switches, it pays off to invest up front in order to prepare for change. An example is the separation of the introduction of a new concept from the introduction of its first use. When we specified the requirements of a voice connection in the example in this paper, we separated the introduction of a basic connection and of a voice channel, thus facilitating a later introduction of a video channel. When we specify requirements incrementally, it is important to specify those requirements (modules) first that we expect to be most stable, and to specify the ones most likely to change last and closest to the leaves of the uses hierarchy; thus minimizing the probable impact of a change. In order to make such decisions, it is important to analyse the scope, commonality and variability (SCV) of the family early, as it was proposed for families of programs [3, 4, 5].

Currently we continue to extend our case studies, in order to gain more practical experience with the approach, and we work on improving the tool support.

References

[1] Pamela Zave. Secrets of call forwarding: A specification case study. In Gregor von Bochmann, Rachida Dssouli, and Omar Rafiq, editors, "Formal Description Techniques VIII". Chapman & Hall (1996).

[2] David Lorge Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.* **2**(1), 1–9 (March 1976).

[3] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software* pages 37–45 (November/December 1998).

[4] David A. Cuka and David M. Weiss. Engineering domains: Executable commands as an example. In Premkumar Devanbu and Jeffrey Poulin, editors, "5th Int'l Conf. on Software Reuse", pages 26–34, Los Alamitos, CA, USA (2–5 June 1998). IEEE Computer Society.

[5] Neeraj K. Gupta, Lalita Jategaonkar Jagadeesan, Eleftherios E. Koutsofios, and David M. Weiss. Auditdraw: Generating audits the FAST way. In "3rd IEEE Int'l Symposium on Requirements Engineering", pages 188–197, Annapolis, MD, USA (6–10 January 1997). IEEE Computer Society Press.

[6] Jan Bredereke. Modular, changeable requirements in Functional Documentation for telephone switching. *Formal Methods in System Design* (2000). Special issue on Tabular Notation. *Submitted.*

[7] Jan Bredereke. Requirements specification and design of a simplified telephone network by Functional Documentation. CRL Report 367, McMaster University, Hamilton, Ontario, Canada (December 1998).

[8] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Sci. Comput. Programming* **25**(1), 41–61 (October 1995).

[9] A. John van Schouwen, David Lorge Parnas, and Jan Madey. Documentation of requirements for computer systems. In "IEEE Int'l. Symposium on Requirements

Engineering – RE'93", pages 198–207, San Diego, Calif., USA (4–6 January 1993). IEEE Comp. Soc. Press.

[10] 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).

[11] Clemens Fischer. Combining Object-Z and CSP. In Adam Wolisz, Ina Schieferdecker, and Axel Rennoch, editors, "Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch", GMD-Studien Nr. 315, pages 119–128. GMD, St. Augustin, Germany, Berlin, Germany (19–20 June 1997).

[12] Clemens Fischer. "Combination and Implementation of Processes and Data: from CSP-OZ to Java". PhD thesis, University of Oldenburg, Oldenburg, Germany (January 2000). *Submitted*.

[13] C. A. R. Hoare. "Communicating Sequential Processes". Prentice-Hall (1985).

[14] A. W. Roscoe. "The Theory and Practice of Concurrency". Prentice-Hall (1997).

[15] Graeme Paul Smith. "The Object-Z Specification Language". Kluwer Academic Publishers (2000).

[16] Roger Duke, Gordon Rose, and Graeme Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards & Interfaces* **17**, 511–533 (1995).

[17] Jim Woodcock and Jim Davies. "Using Z – Specification, Refinement, and Proof". Prentice-Hall (1996).

[18] John Michael Spivey. "The Z notation: a reference manual". Series in Computer Science. Prentice-Hall, New York, 2nd edition (1995).

[19] ITU-T, Recommendation Z.100 (11/99). "CCITT Specification and Description Language (SDL)" (November 1999).

[20] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, "A Classical Mind, Essays in Honour of C.A.R. Hoare", pages 353–378. Prentice-Hall (1994).

[21] Clemens Fischer and Heike Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, "Proc. of the 1st Int'l. Conf. on Integrated Formal Methods – IFM'99". Springer (1999).

[22] Boris Wirtz. Übersetzung des Daten- und Verhalten-integrierenden Spezifikationsformalismus CSP-OZ nach CSP. Masters thesis, University of Oldenburg, Dept. of Comp. Sci. (2000). *To appear*.

[23] Jens von Garrel. Parsing, Typechecking und Transformation von CSP-OZ nach Jass. Masters thesis, University of Oldenburg, Dept. of Comp. Sci. (July 1999).

[24] Jan Bredereke. Modular, changeable requirements for telephone switching in CSP-OZ. Tech. Rep. IBS-99-1, University of Oldenburg, Oldenburg, Germany (October 1999).

[25] Jan Bredereke. Requirements specification and design of a simplified telephone network by Functional Documentation. CRL Report 367, McMaster University, Hamilton, Ontario, Canada (December 1998). ISSN 0381-9337.

[26] Jan Bredereke. Avoiding feature interactions in the users' interface. In Kristofer Kimbler and L. G. Bouma, editors, "Feature Interactions in Telecommunications and Software Systems V", pages 305–317. IOS Press, Amsterdam (September 1998).

[27] David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.* **SE-5**(2), 128–138 (March 1979).

[28] Gunter Saake and Can Türker, editors. "Proc. of the 1st FIREworks Workshop – FIREworks '98", number 10 in Preprint-Reihe, Magdeburg, Germany (15–16 May 1998). Fakultät für Informatik, Universität Magdeburg. URL http://wwwiti.cs.uni-magdeburg.de/iti_db/veroeffentlichungen/98/SaaTue98.html.

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

[30] Malte Plath and Mark Ryan. Plug and play features. In Saake and Türker [28], pages 78–105. URL http://wwwiti.cs.uni-magdeburg.de/iti_db/veroeffentlichungen/98/SaaTue98.html.