

Configuring Members of a Family of Requirements Using Features

Jan Brederke

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

Abstract.

We explicitly consider entire families of software requirements; this enables us to configure family members using features. Our goal is to avoid feature interaction problems by a software engineering approach. Naive feature orientation does not scale due to complexity problems. But we can structure a family of requirements into requirements modules to make it easier to maintain. We then can configure family members from these requirements modules. In this, we must distinguish the notions of a requirements module and of a feature to avoid feature interaction problems. We demonstrate our ideas by adding suitable constructs for families and for features to the formalism Z, and by then specifying a family of LAN message services and a set of features for it.

Keywords. feature interaction, software engineering, rigorous requirements, information hiding, configuration management, Z

1. Introduction

Our goal is to avoid feature interaction problems by a software engineering approach. Naive feature orientation does not scale due to complexity problems. We concentrate on software requirements. All feature interaction problems are present already in the requirements, if the requirements are sufficiently complete. We explicitly consider entire families of requirements. Different family members have different sets of features. There is a typical process pattern in which such a family evolves during its maintenance. We also concentrate on rigorous software requirements. Rigorous requirements are necessary to ensure the dependability of the software system. Telephone switching systems are an example of particularly long-lived dependable software systems.

1.1. Naive Feature Orientation Does Not Scale

A feature-oriented description of a telephone switching system is attractive but also can promote undesired feature interactions.

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.

There has been considerable research effort on feature composition operators. In particular, in the FIREworks project [1,2] (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 [3]: one specifies a base system and textual increments, which are composed by a precisely defined (syntactic) composition operator.

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 [4,5]. 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.

1.2. An Evolution Process Pattern for Maintaining Families of Requirements

We explicitly consider entire families of requirements; such a family evolves in a typical process pattern during its maintenance. Initially, we analyze the domain (more or less thoroughly), and we specify and then implement one or a few systems. Over time, customers demand new or other features. Therefore, we specify explicitly and then implement more members of the family. Over more time, we iterate this many times.

In this process, implicit family members become explicitly specified. The initial analysis of the domain determines which systems are potential family members, and which systems can never be part of the family. Also, the requirements of one family member, or of a few, are specified explicitly in the beginning. We call these the *explicit family members*. We call the other potential family members the *implicit family members*.

2. Information Hiding Requirements Modules

We can structure a family of requirements into requirements modules to make it easier to maintain. Requirements modules mean encapsulation in the information hiding sense. A requirements module is a set of properties that are likely to change together. We need a hierarchy of requirements modules to structure a large number of requirements. But existing formal languages such as the well-known formalisms Z and Object-Z do not fully support hierarchical requirements structuring. We therefore extend the formalism Z suitably; this extension is also a necessary base for our feature construct in Sect. 4 below. We demonstrate our approach on an example, a family of LAN message services.

2.1. 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. We also point out that this section is about design, not yet about requirements.

A *module* in the information hiding sense [6,7] 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.

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 [7] 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 [8]. 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.

A family of requirements needs a module structure, too, but 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.

2.2. Requirements Modules for Families of Requirements

A *requirements module* is a set of properties that are likely to change together. The likelihood of change of a property in a family is determined by its abstractness. The abstractness of a property in a family is determined by the share of the family in which the property is included in. A requirements module is an *abstract requirements specification*. An abstract requirements specification is a subset of properties of a single product. It will hold equally well for several or even all products of the product space.

Each of the properties is an *abstract requirement*. The property holds for all members of the family that include this property. This is in accordance with the definition of “abstract” above. In principle, we could define a metric for the abstractness of a property. One property is more abstract than another if it is always included when the other is included, and at least for one more family member. We would need to assign concrete numbers for a complete metrics. In practice however, we can estimate the abstractness of a property only. However, we are only interested in qualitative, relative comparisons of abstractness, anyway. This suffices for the following considerations.

We can regard the abstractness of a property as determining the *likeliness of its change*. When we step randomly from one family member to another one, the property will be added or removed with a likeliness that depends directly on its abstractness. We distinguish between the likeliness of change for a single property and the likeliness that a set of properties changes together. There can be a set of properties that is either included completely, or of which no element is included, for a large part of the family. In this case, there is a strong correlation of being included or not. Both kinds of likeliness of change are important for a suitable requirements module structure. Properties that change together should be arranged together, in the description of the family of requirements. And the likeliness of change for a single property should determine whether other properties depend on it or not. The rationale for arranging properties together that change together is the historically proven success of the information hiding approach with design modules.

2.3. A Hierarchy of Requirements Modules

We need a hierarchy of requirements modules to structure a large number of requirements. 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. We structure the requirements modules analogously to the hierarchy of design modules [7].

2.4. Support for Requirements Modules and Families in Existing Formalisms

Existing formal languages such as the well-known formalisms Z [9,10] and Object-Z [11] do not fully support structuring by hierarchical requirements modules and by requirements families. Formal languages can be used to document requirements rigorously. Rigorous requirements are necessary to ensure the dependability of the software system. The formalism Z is a well-known notation to describe properties of an information system precisely; however, we can express a module hierarchy and a family only informally. Object-Z is the most popular of several extensions of Z that add object-oriented mechanisms; however, it is not easier to express a module hierarchy in Object-Z than in plain Z, and it is *more* difficult to express a family. This is due to the fact that the standard object-oriented mechanisms have been added in Object-Z, but no package mechanism. A package mechanism would have made the ease of expression in Object-Z similar to plain Z. A host of other formal languages exists; we select Z here because it is comparably widely used. None of these other languages fully support structuring by hierarchical requirements modules, too.

Z is a formalization of set theory. Z allows to specify a *state space* and constraints over it. Powerful mathematical operators help to express constraints concisely. There are

also *operations* for specifying changes to the state space. Operations usually constrain the changes, one can express preconditions and postconditions. Operations can also have parameters. Z is standardized by the International Standardization Organization (ISO) [9]. Before this official standardization, Z was defined for many years by Spivey's reference manual [10].

Z offers basic support for modularity. A schema allows to group variables together. A paragraph is the basic formal unit of structure for Z. A Z specification document consists of interleaved passages of formal text and informal prose explanation. The formal text consists of a sequence of paragraphs. A paragraph can be, for example, one type definition, one axiomatic description, or one schema definition. Spivey's Z already uses paragraphs [10].

A section and a specification are higher-level formal units of structure. These are extensions by the ISO standard. A *specification* consists of *sections*, which in turn consist of paragraphs. The section construct allows for a constraint-oriented, incremental style of specification. Each section has a self-contained formal meaning. Any initial sequence of sections can be taken as the set of requirements for a variant of the specified system. Each further section adds more constraints on the system (and new declarations).

The parents construct serves to specify the dependency relation among sections. The parents construct is part of the section construct. It lists the names of other sections of which the current section is an extension.

There is no formal way to express a multi-level hierarchy of modules. That is, there is no way to group related lower-level modules together into higher-level modules. It can be done informally only. We can have a hierarchy of informal chapters, sections, sub-sections and so on around the formal Z sections. But such an informal hierarchy is already sufficient to arrange together the formal sections according to their likeliness of change.

We can express a family of requirements by a suitable convention. For this, we use the section construct together with the parents construct. The dependency relation is a hierarchy, i. e., acyclic. We can use the convention that each bottom leaf section in the hierarchy is one member of the family. Such a leaf section composes the desired properties from other, non-bottom sections through its parent construct.

We can extract a single member of the family into a separate document automatically, with some limitations. We must use the above convention that the leaves in the dependency hierarchy of sections each specify one family member. We then can indicate the family member desired through the name of the corresponding leaf section. A suitable tool can follow the dependency relation in order to identify all sections included in this family member. The tool then can copy the document with the family into another document, while deleting those sections that are not included. A limitation is that we cannot select the appropriate informal text automatically. In particular, if there are higher-level, informal chapters, any informal closing remarks of the first section are merged inseparably with any following informal chapter heading.

Object-Z [11] is the most popular of several extensions of Z that add object-oriented mechanisms; however, it is not easier to express a module hierarchy in Object-Z than in plain Z, and it is *more* difficult to express a family. Object-Z offers a basic support for modularity, too, but it is different from the one in plain Z. Object-Z does not provide the formal section construct from the ISO standard; Object-Z is based on the old version of Z by Spivey. Object-Z is similar to plain Z in that there is no formal way to

express a multi-level hierarchy of modules. (Note that inheritance does not group classes together.) Object-Z offers no easy way to express a family of requirements formally; it falls back behind plain ISO Z with this respect. Accordingly, there is no mechanizable way to extract a single member of a family into a separate document.

2.5. Adding Support for Requirements Modules to the Formalism Z

We extend the formalism Z by a hierarchical module structure and by explicit interfaces between requirements modules. This extension is also a necessary base for our feature construct in Sect. 4 below. Explicit interfaces between requirements modules help to control dependencies between them. A property that is likely to change should not be depended upon by properties from other modules; distinguishing between interface properties and secret properties of a module is a mechanism to avoid this.

To achieve a formal hierarchical module structure, we add a “chapter” construct on top of the existing “section” construct of Z. A chapter groups sections together. It may also group other chapters together. This allows for a full hierarchy of chapters, sub-chapters, and sections. A chapter is different from a section in that it has no parents construct. This is because the dependency relation should be defined over individual properties, i. e., sections, not over higher-level modules, i. e., chapters.

To achieve explicit interfaces, we allow to prefix a section or a chapter with the new keyword “private”. We add a suitable type inference rule to restrict the access to private sections and chapters. Otherwise, the definition of the semantics remains unchanged. We call the extended language Z_{CI} .

Syntax. We modify the BNF grammar of plain Z [9] at the top level as follows:

```
Specification = { Chapter | Section }
              | { Paragraph };

Chapter = ZED , [ private ] , chapter , NAME , END , { Section } ,
        ZED , endchapter , [ NAME ] , END
        | ZED , [ private ] , chapter , NAME , END , { Chapter } ,
        ZED , endchapter , [ NAME ] , END ;

Section = ZED , [ private ] , section , NAME ,
         parents , [ NAME , { -tok , NAME } ] , END , { Paragraph }
         | ZED , [ private ] , section , NAME , END , { Paragraph } ;

Paragraph = ...
```

We add one context-sensitive rule to the syntactic grammar: the optional NAME after the token “endchapter”, if it exists, must be the same as the NAME after the corresponding token “chapter”. The repetition of the NAME is intended to improve readability when a chapter is long and the reader might have lost a part of the context.

All specifications in Z are also specifications in Z_{CI} (with the same meaning). The only exceptions are those specifications in Z that use the three new keywords “chapter”, “endchapter”, and “private”. These specifications are not legal in Z_{CI} anymore.

Lexis and Mark-Up. The lexical analysis phase groups Z characters to tokens. The lexis specifies a function from sequences of Z characters to sequences of tokens. We add the three new alphabetic keywords and associated tokens to the lexis.

A mark-up allows to represent all of the many characters of Z on machines that have only a small character set. The definitive representation of Z characters is in 16-bit Unicode [12]. A mark-up is a mapping to the Unicode representation. As for plain Z , there is a mark-up for the typesetting system \LaTeX [13] and a light-weight one for email. In the \LaTeX mark-up of Z_{CI} , the input “`\begin{zchapter}`” is converted to the Z character `ZEDCHAR`, and the input “`\end{zchapter}`” is converted to the Z character `ENDCHAR`. The lexer then converts the Z character `ZEDCHAR` to the token `ZED` and the Z character `ENDCHAR` to the token `END`. The mark-up input “`\CHAPTER`” is converted to the string “chapter”, followed by a space. In context, a Z_{CI} chapter heading thus looks like:

```
\begin{zchapter}
\CHAPTER NAME
\end{zchapter}
```

Furthermore, the mark-up input “`\ENDCHAPTER`” is converted to the string “endchapter”.

We suggest that the \LaTeX environment `zchapter` might provide a suitable automatic, hierarchical chapter numbering scheme when typesetting. We implemented a \LaTeX style that follows this suggestion.

Type Inference Rules. We add a type inference rule for chapter names. It takes care that the chapter hierarchy is consistent, and that chapter names are different from section names. We also add a second type inference rule for chapter names. It restricts the access to private sections and chapters. The detailed formal rules can be found in [14].

Semantics. The definition of the semantics of plain Z remains unchanged; there is no formal meaning for a chapter beyond the access restrictions it imposes.

2.6. Example: A Family of LAN Message Services

We demonstrate our approach on an example, a family of LAN message services. We specified this family in the language Z_F . Z_F is a further extension of the language Z_{CI} ; we will define Z_F in Sect. 4 below. Please ignore the keyword “default” for now. The full specification is in [14]. Figure 1(a) shows some of the formal chapter and section headings. The complete specification has 28 sections on 19 pages.

The basic idea of such a service is that computer users on a local area network (LAN) can send each other short messages that are displayed immediately. These systems can have less or more functionality. A very simple version just unconditionally opens a graphical window at the receiving side and displays one line of text. This can be convenient to alert one’s colleagues on the same floor that one will cut a birthday cake in five minutes. Other family members can support individual addressing, message blocking, message re-routing, output on a text console, delayed messages, and so on. Therefore, all these aspects are likely to change from one family member to another.

At the top level, the specification is divided into the requirements on the behaviour of the software system to build and into the requirements on its environment. The environment comprises the communicating entities, the messages they want to exchange, and the existing hardware and software that that can be made use of. The specification of the behaviour of the software system describes what the system does to the communi-

<pre> 1. chapter environment 1.1 chapter computing_platform 1.1.1 chapter data_types ... 1.1.1.2 chapter text_strings 1.1.1.2.1 section text_string_base ... 1.1.1.2.2 private default section c_text_string ... endchapter text_strings 1.1.1.3 chapter graph_images 1.1.1.3.1 section graph_image_base ... endchapter graph_images endchapter data_types ... endchapter computing_platform 1.2 chapter device_interfaces ... 1.2.2 chapter communicating_entities 1.2.2.1 chapter messages ... endchapter messages ... 1.2.2.8 private default chapter user_interface 1.2.2.8.1 section user_base ... 1.2.2.8.2 private default chapter graphical_user_interface 1.2.2.8.2.1 section gui_comm_base ... 1.2.2.8.2.2 private default section gui_io_base ... endchapter graphical_user_interface 1.2.2.8.3 private chapter textual_user_interface ... endchapter textual_user_interface endchapter user_interface ... endchapter communicating_entities endchapter device_interfaces endchapter environment 2. chapter system_behaviour 2.1 chapter function_drivers 2.1.1 chapter message_delivery ... 2.1.1.2 private default section timely_message_delivery ... 2.1.1.3 private default section correct_message_delivery ... 2.1.1.4 private default section broadcast_message_delivery ... endchapter message_delivery endchapter function_drivers ... endchapter system_behaviour </pre>	<pre> feature note_to_all: + broadcast_message_delivery + text_message_base (+) one_line_message feature scroll_text_message: + multi_line_message - one_line_message + graphical_user_interface - textual_user_interface feature birthday_cake_picture: + broadcast_message_delivery + graphical_message_base - text_message_only + graphical_user_interface feature lunch_alarm: + automated_agent_interface + broadcast_message_delivery (+) text_message_base feature deskPhoneXY_hardware: - graphical_user_interface + textual_user_interface + max_lines2_message + pascal_text_string - c_text_string ... </pre> <p>(b) Some of the feature definitions in the configuration rule base.</p> <pre> The "Lunch Phone" system: lunch_alarm deskPhoneXY_hardware The "Classic PC" edition: note_to_all multi_line_text_message standardPC_hardware The "Deluxe PC" edition: lunch_alarm birthday_cake_picture note_to_all multi_line_text_message scroll_text_message standardPC_hardware </pre> <p>(c) Some queries for family members.</p>
--	---

(a) Some of the formal chapter and section headings in the requirements module base. We omitted the "parents" construct and all of the actual contents of the sections.

(c) Some queries for family members.

Figure 1. Extract of the specification of a family of LAN message services in [14].

cating entities and the messages, without referring to any details of the existing hardware or software. We expect that changes in the hardware devices will happen independently from changes to the high-level behaviour of the system. For example, a change from a textual user interface to a graphical user interface will be independent of whether there is a message broadcast scheme or an individual message addressing scheme.

3. Configuring Family Members Using Features

We can configure family members from requirements modules; in this, we must distinguish the notions of a requirements module and of a feature to avoid feature interaction problems. A feature is a set of changes, not a requirements module. A selected configuration of requirements must be consistent; we can reconcile some inconsistencies by configuration priorities.

3.1. A Feature Is Not a Requirements Module

A feature is a set of changes, not a requirements module. We mean *change* in the sense of software configuration management [15], i. e., directed delta that is a sequence of (elementary) change operations which, when applied to one version, yields another version. Features and requirements modules are similar. Both concepts serve to group properties. But there are marked differences [16]. Examples from telephony support our claim that a feature often affects different requirements modules of a well-designed requirements module hierarchy. The telephony system sketched in the examples avoids many kinds of undesired feature interactions.

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. A feature is an entire set of changes, not only one change. This is because a feature usually should be made up of changes to different requirements modules.

Examples from telephony support our claim that a feature often affects different requirements modules of a well-designed requirements module hierarchy.

- 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 (example: the above 800 number for the pizza delivery role), a module that translates a role address into 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 from 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. Of course, this feature also removes the property that the caller is charged.

- 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 module in the first example that 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 enable the user to set his or her 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. Nevertheless, structuring the system only into the above features would not have been good. We could not have reused requirements modules across features, as we have done above. This would have been the naive feature-oriented approach.

The telephony system sketched in the examples above avoids many kinds of undesired feature interactions. There is only one module that translates a role address into a device address. A consistent distinction between user roles and devices is important. For example, the well-known undesired feature interactions between call forwarding and terminating call screening disappear as soon as we make clear whether the screening acts on the device that made the connection or whether it acts on the user who initiated the session. The latter, of course, needs that users explicitly register with devices. If devices are screened, the phone should ring, if users are screened, the phone should not ring. The description of the features will make clear to the customers what they can expect. A naive feature oriented system adds the translation in the call forwarding feature only, with entailing confusion and grief.

3.2. Configuration Priorities

Differentiating the strictness of configuration rules helps to reconcile inconsistencies; we propose to distinguish between the essential properties and the changeable properties of a feature. Conflicting feature definitions that demand to both include and to exclude a property are an important special case of an inconsistent configuration. Such an inconsistency is an adverse feature interaction. For example in our family of LAN message services in Figure 1, the feature `note_to_all` demands the property `one_line_message` to be included, while the feature `scroll_text_message` demands this property to be excluded.

A solution is to have different configuration priorities for the properties of a feature. We found that not all properties of a feature are equally important. Some properties are

definitely necessary to meet the expectations evoked by the feature's name. But other properties are provided only in order to make the requirements specification complete and predictable for the user. For example, the feature `note_to_all` can be recognized no matter what the requirements say on how many lines a note can have.

We therefore propose that the specifier of a feature documents explicitly which properties are *essential properties* and which are *changeable properties*. If a feature demands the inclusion of a changeable property, but another feature demands the exclusion of this property, then the property is excluded without the configuration being inconsistent. The attribution of a priority is per feature, not per property. A property can be essential for one feature and be just ancillary for another feature. We will introduce such a distinction of priorities for the formalism Z in Sect. 4 below.

4. Adding a Feature Construct to the Formalism Z

We complete our support for families of requirements in the formalism Z by adding a suitable feature construct; it allows to specify a configuration. We illustrate our approach by presenting some features for our family of LAN message services from Sect. 2.6 above.

We could specify family members in plain Z by an informal convention, but we cannot extend the convention to features. In Section 2.4, we found that we can extract a single member of the family into a separate document automatically. We must use the convention that the leaves in the dependency hierarchy of sections each specify one family member. But the extension of this convention does not work, that the sections above the leaf sections serve to specify features. We can use this extended convention to specify the properties that a feature includes. But we cannot express that a feature excludes some other properties. There is no means for non-monotonous extensions. But a feature is inherently non-monotonous [17]. Most features really change the behaviour of the base system.

We define an extension of the formalism Z for families of requirements; it includes a construct to specify a feature. We call the language Z_F . We define the language by a transformation that maps the specification of a family of requirements to an individual family member, depending on a list of selected features. In the terms of software configuration management [15], we will use two-level intertwined intensional versioning. A feature is a list of sections added and a list of sections removed. The list of added sections is differentiated into essential and into changeable sections. The list of removed sections is not differentiated.

We define the language Z_F by a transformation from Z_F to Z_{CI} (as defined in Sect. 2.5). A description of a family member in the language Z_F consists of three parts (Fig. 2): the requirements module base, the configuration rule base, and a query. The requirements module base contains all sections that potentially can be part of a family member, grouped into chapters. The configuration rule base contains the definitions of all features. A query is a list of those features that are selected for the family member desired.

We therefore define the language Z_F in three independent parts. We leave to the discretion of the specifiers to maintain these parts. For example, they can keep the module base in a \LaTeX include file, and the rule base in another \LaTeX include file. The specifiers

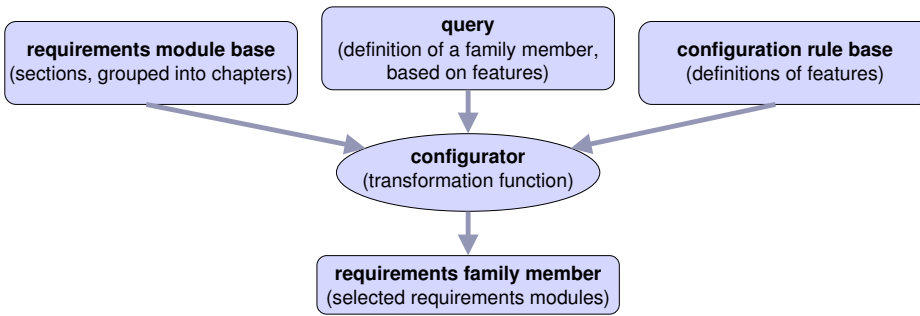


Figure 2. The three parts defining a family member in the language Z_F .

can combine the two files into a family document for printing through the include mechanism of \LaTeX . Any generated family member will contain parts of the module base, but no parts of the rule base. The latter is not interesting for the users of the family member document. The rules could even be confusing. The query can be formulated, for example, ad-hoc as input on the command line of the configurator tool.

4.1. Syntax

Requirements module base. We define the syntax by a context-free grammar in BNF. The grammar is completely separate from the grammar of Z (and of Z_{CI}). The start symbol is the family. A family consists of chapters and sections that serve as requirements modules.

```
Family = { Chapter | Section } ;
```

A chapter and a section are defined similar to the definition in Z_{CI} , with two exceptions. There can be the additional keyword “default”. And the body of a section consists of informal text only, with respect to the transformation. We have no rule for a “Paragraph”. Any Z_{CI} paragraphs are just informal text. We will associate it to the preceding token. Therefore, any Z_{CI} paragraph will be part of the section whose heading immediately precedes it.

```
Chapter = ZEDCHAR , [ private , [ default ] ] , chapter , NAME , ENDCHAR ,
         { Chapter | Section } ,
         ZEDCHAR , endchapter , [ NAME ] , ENDCHAR ;

Section = ZEDCHAR , [ private , [ default ] ] , section , NAME ,
         [ parents , [ NAME , { ,tok , NAME } ] ] , ENDCHAR ;
```

The keyword “default” serves to specify the parts of the base system. The base system consists of all interface sections/chapters and of all private sections/chapters that are marked as “default”.

Configuration rule base. The BNF start symbol is the list of features. A feature is a list of sections added and a list of sections removed. The list of added sections is differentiated into essential and into changeable sections. A feature has a name.

```
FeatureList = { Feature } ;
```

```

Feature = ZEDCHAR , feature , NAME , : , ENDCHAR ,
        ZEDCHAR , { FeatAddEss | FeatAddChg | FeatRemove } ,
        ENDCHAR ;

FeatAddEss = + , NAME ;

FeatAddChg = (-tok , + , )-tok , NAME ;

FeatRemove = - , NAME ;

```

The add list and the remove list can also refer to entire chapters instead of individual sections. This is an abbreviation. We will define below adding a chapter to be equivalent to adding all its default sections.

Query. A family member is specified by a list of feature names.

```
FamilyMember = ZEDCHAR , { NAME } , ENDCHAR ;
```

4.2. Lexis and Mark-Up

The lexical analysis phase groups Z characters to tokens. For each of the three parts of a family member description, the lexis specifies a function that maps sequences of Z characters to sequences of tokens. We only have alphabetic keywords. They are the ones in the syntax definition. For the requirements module base, the lexis also specifies a second function. This second function attributes each token with the sub-sequence of input Z characters that were grouped into this token. We will need these sub-sequences to compose the output of the translation to a family member.

The mark-up is straightforward. Details can be found in [14]. Analogous to the lexis, there is also a second mapping for the requirements module base that attributes each Z character with the sub-sequence of input \LaTeX /email characters that were grouped into this Z character. Any informal text is associated to the preceding formal Z character, again.

4.3. Type Inference Rules

Some type inference rules provide general consistency and furthermore adherence to our intended restrictions on essential sections and chapters. This enables suitable type checks on the specification of the family and on any query.

For the requirements module base, we adapt the applicable type inference rules from Z_{CI} on chapters and sections (omitting the rules on parents). Additionally, we demand that multiple appearances of a chapter name are marked consistently as “default”. This gives us some general consistency. The formal rules can be found in [14].

For the configuration rule base, we define four obvious consistency rules: C1) All features must have distinct names. C2) Feature names are in the same name space as section and chapter names; hence they all must be different. C3) Each section and chapter name may appear only once in the list of a feature. C4) All section and chapter names that appear in the list of a feature must be defined in the requirements module base.

For the query, we define one general consistency rule and one restriction rule: Q1) All feature names in the query must be defined in the configuration rule base. Q2) For all feature names in the query, no feature may remove a section or chapter that is listed among the essential sections and chapters of some feature of this query.

Note that being essential does *not* propagate from a chapter to all of its sections. A chapter may be marked as essential, but nevertheless some of its default sections can be removed. Note furthermore that a feature may remove a section that is needed as a parent by another, included section. We leave this consistency check to the definition of Z_{CI} , where it is performed anyway. Similarly, we do not check the access restrictions to private sections and chapters here.

4.4. Semantics

The semantics is the transformation that maps the three parts to an individual family member. We define the result of the transformation by the following steps. We start out with a base system consisting of everything that is either an interface or marked as default. To be more precise, the set of base sections is the set of all those sections which are not marked as “private”, and for which none of its enclosing chapters is marked as “private”. Again, the formal definitions can be found in [14].

We then add all the sections that the features in the query demand; and for the chapters that the features in the query demand, we also add all their non-“private” sections. This means adding both the essential and the changeable sections/chapters.

We next remove all the sections that the features in the query demand to be removed; and for the chapters that shall be removed, we remove all their sections. The result is the set of sections S of the family member.

After having determined the set of sections of the family member, we can define the set of chapter headings C of the family member. We keep only those chapter headings for which at least one section exists.

The final step is the definition of the output sequence of Z characters (and mark-up). The output is the same as the requirements module base, but with some parts removed. The order of the output characters is otherwise the same as in the requirements module base. We need the association of the syntax grammar non-terminals “Chapter” and “Section” to their corresponding input from the lexis and from the mark-up. We remove all Z characters associated to syntax grammar non-terminals “Section”, where the corresponding section name is not in S , and we remove all Z characters associated to syntax grammar non-terminals “Chapter”, where the corresponding chapter name is not in C . Additionally, we remove all Z characters associated to any token “default”. Note that all informal text blocks are associated to some formal text; they are also removed suitably where necessary.

4.5. Example: Some Features for the Family of LAN Message Services

We illustrate our approach by presenting some features for our family of LAN message services from Sect. 2.6 above. Figure 1(b) shows some of the feature definitions, and Figure 1(c) contains some queries for family members that are specified using features. The complete requirements module base in [14] contains more feature definitions; also not all requirements modules referenced here are shown in Figure 1(a).

Note that the feature `deskPhoneXY_hardware` means that the system uses the hardware of the office desk phones of brand XY instead of computer terminals. These phones only have a small text display with two lines. The associated software platform is restricted to the language Pascal.

4.6. Graphical Illustration of Configuring a System Using Features

Figure 3 illustrates how we configure a system using features. We take our LAN message service family, again. The figure contains scaled-down versions of the requirements module hierarchy and of the dependency relation. The four sub-figures show the base system, two features, and the resulting complete system.

The base system of our family is in Figure 3(a). The black boxes mark those sections which are part of the “plain” family member that has no features.

The changes by the feature `lunch_alarm` are in Figure 3(b). An automated alarm clock informs everybody when it is time for the lunch break. By default, the alarm is a short text message. A bold frame marks those sections and chapters which the feature adds on top of the base system. A dashed frame means that this addition is not essential and could be overruled by another feature. (Note that this particular non-essential section addition does not change anything, because the section is a default in the base system anyway. This is due to our wish to express our intention behind the feature `lunch_alarm` better.)

The changes by the feature `deskPhoneXY_hardware` are in Figure 3(c). A bold cross marks those sections and chapters which the feature explicitly removes.

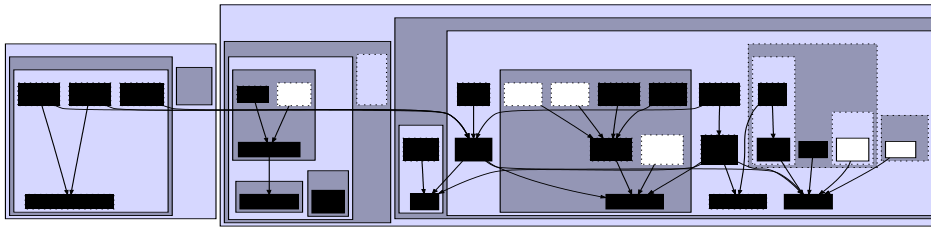
The final configuration of the “Lunch Phone” system is in Figure 3(d). It consists of the above two features. Compare also the formal descriptions of the features and of the “Lunch Phone” system in Figure 1.

5. Discussion

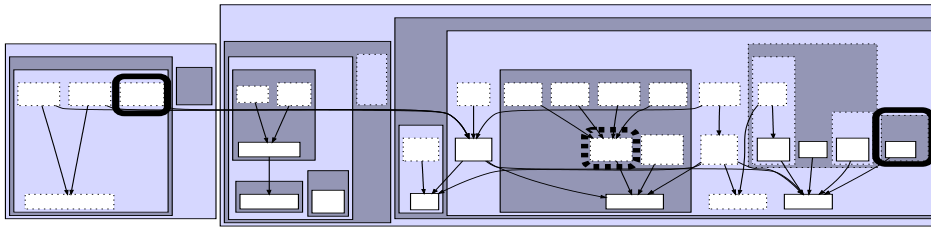
We presented an approach to configure members from a family of requirements using features. A family of requirements is a set of requirements specifications for which it pays off to study the common requirements first; we have several versions of requirements in the sense of configuration management. Our goal is to avoid feature interaction problems by a software engineering approach. Naive feature orientation does not scale due to complexity problems. But we can structure a family of requirements into requirements modules to make it easier to maintain. We then can configure family members from these requirements modules. In this, we must distinguish the notions of a requirements module and of a feature to avoid feature interaction problems. A feature is a set of changes, not a requirements module. A family member is defined by three parts: the requirements module base, the configuration rule base, and a query. We see a feature as a configuration rule. A specific selection of feature names then constitutes a query. We demonstrated our ideas by adding suitable constructs for families and for features to the formalism Z, and by then specifying a family of LAN message services and a set of features for it.

Our approach is applicable to other formalisms, too. Not by chance, this paper does not show any actual Z formulas from our specification in [14]. Only the formal means for structuring and configuring are relevant here. However, the underlying formalism must support a constraint-oriented specification style. This allows to split the requirements into small, independent properties.

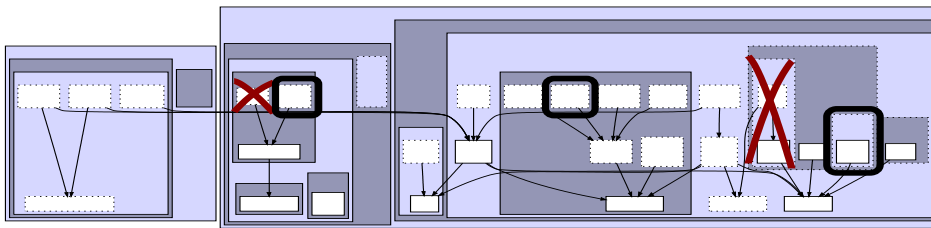
We already have experience with one other formalism, CSP-OZ [18,19]. We report two insights from this earlier work. First, we did not yet distinguish between a requirements module and a feature. This had the disadvantages discussed above. Second, plain CSP-OZ [20,21] provides two independent constructs for grouping requirements; this



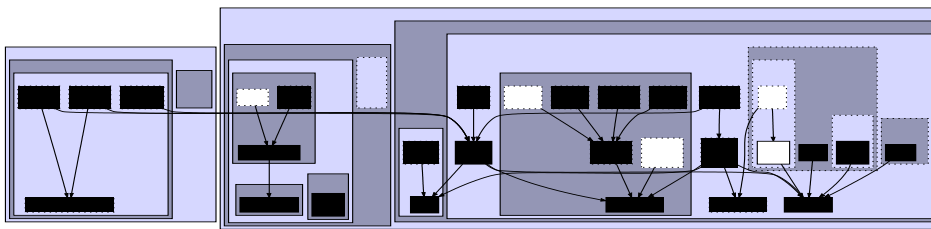
(a) The base system.



(b) The feature lunch_alarm.



(c) The feature deskphone_hardware.



(d) The complete "Lunch Phone" system.






legend:  add, essential  add, changeable  remove  selected  not selected

Figure 3. Configuring a system using features in the LAN message service family.

complicated our extension. Plain CSP-OZ has the class construct from object orientation. But CSP-OZ also has the section construct from ISO Z. The latter was adopted together with the detailed definition of the syntax and semantics of Z which only the ISO standard provides. Consequently, our extension ended up having two similar, but nevertheless different means for grouping requirements, too.

We based our extension on the section construct of CSP-OZ and not on the class construct, because the latter cannot group together certain parts of a specification. The class construct comes from Object-Z [11]. Object-Z, and thus CSP-OZ, allows all the different kinds of paragraphs of Z to be used, outside of any class. A paragraph can be, for example, one type definition, one axiomatic description, or one schema definition.

The class construct did not interwork too well with the section construct. There were many sections that contained exactly one class. This turned out to be syntactic clutter. We removed it by defining a shorthand notation in our extension of CSP-OZ. The shorthand allows to use a class in the places where a section can occur, too. A syntactic transformation then implicitly adds a suitable section heading for such a class [18,22]. This shorthand fixed the problem at the surface, but conceptually it is not entirely satisfactory. Our current formalism Z_F does not have this problem anymore. Z_F does not build on Object-Z, only on ISO Z. It is an interesting research question how our requirements configuration means can be integrated seamlessly with object-oriented requirements analysis.

Configuring requirements family members needs tool support; some tools already exist, more are planned. We have written a \LaTeX style for typesetting specifications in Z_F . We have implemented and used a configurator and a type checker for a predecessor of Z_F ; we now plan the same for the current Z_F , following the precise specification here (and in more detail in [14]). We can exploit the experience with the checker and generator tool [22] that we already wrote for our extension of CSP-OZ discussed above. We already use the free tool CADiZ [23] for general type checking. CADiZ is for plain (ISO-)Z, but we have written a simple transformator from Z_{CI} to plain Z.

This is ongoing work. Besides having more tool support, we want to gather more experience with specifying and configuring families of requirements in Z_F . We already collected considerable experience with a large case study in our extension of CSP-OZ. We specified the requirements for a telephone switching system [18,19]. The case study comprises about 40 pages of commented formal specification, with about 50 sections in nine features/modules, including the base system. The structure of the specification avoided the feature interactions between, e. g., call forwarding and terminating call screening, and our type checker tool pointed us to a contradiction between features. An interesting question for further research is how we can associate code fragments to requirements written in a constraint-oriented style, such that we can also configure all or a part of the implementation automatically from a code base, using features.

References

- [1] 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.
- [2] STEPHEN GILMORE AND MARK D. RYAN, editors. “Language Constructs for Describing Features”. Springer (2001).
- [3] SHMUEL KATZ. A superimposition control construct for distributed systems. *ACM Trans. Prog. Lang. Syst.* **15**(2), 337–356 (April 1993).

- [4] DANIEL AMYOT AND LUIGI LOGRIFFO, editors. “Feature Interactions in Telecommunications and Software Systems VII”. IOS Press, Amsterdam (June 2003).
- [5] MUFFY CALDER AND EVAN MAGILL, editors. “Feature Interactions in Telecommunications and Software Systems VI”. IOS Press, Amsterdam (May 2000).
- [6] DAVID LORGE PARNAS. On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15**(12), 1053–1058 (1972). Reprinted in [24].
- [7] 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 [24].
- [8] 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 [24].
- [9] “Information Technology – Z Formal specification notation – Syntax, type system and semantics”. ISO/IEC 13568:2002(E) (July 2002).
- [10] JOHN MICHAEL SPIVEY. “The Z notation: a reference manual”. Series in Computer Science. Prentice-Hall, New York, 2nd edition (1995).
- [11] GRAEME SMITH. “The Object-Z Specification Language”. Kluwer Academic Publishers (2000).
- [12] “ISO/IEC 10646-1:1993 Information Technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane” (1993).
- [13] FRANK MITTELBACH, MICHEL GOOSSENS, JOHANNES BRAAMS, DAVID CARLISLE, AND CHRIS ROWLEY. “The L^AT_EX Companion”. Addison-Wesley, 2nd edition (April 2004).
- [14] JAN BREDEREKE. “Maintaining Families of Rigorous Requirements for Embedded Software Systems”. Habilitation thesis, University of Bremen, Germany (2005). *To appear*.
- [15] REIDAR CONRADI AND BERNHARD WESTFECHTEL. Version models for software configuration management. *ACM Comput. Surv.* **30**(2), 232–282 (June 1998).
- [16] JAN BREDEREKE. On feature orientation and on requirements encapsulation using families of requirements. In MARK D. RYAN, JOHN-JULES CH. MEYER, AND HANS-DIETER EHRICH, editors, “Objects, Agents, and Features”, volume 2975 of “LNCS”, pages 26–44. Springer (2004).
- [17] HUGO VELTHUIJSEN. Issues of non-monotonicity in feature-interaction detection. In KONG ENG CHENG AND TADASHI OHTA, editors, “Feature Interactions in Telecommunications III”, pages 31–42. IOS Press, Amsterdam (1995).
- [18] 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).
- [19] JAN BREDEREKE. Families of formal requirements in telephone switching. In Calder and Magill [5], pages 257–273.
- [20] CLEMENS FISCHER. Combination and implementation of processes and data: from CSP-OZ to Java. PhD thesis, report of the Comp. Sc. dept. 2/2000, University of Oldenburg, Oldenburg, Germany (April 2000).
- [21] 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).
- [22] JAN BREDEREKE. “genFamMem 2.0 Manual – a Specification Generator and Type Checker for Families of Formal Requirements”. University of Bremen (October 2000). www.tzi.de/~brederek/genFamMem.
- [23] IAN TOYN ET AL.. “CADiZ reference manual”. University of York, Heslington, York, England (2002).
- [24] DANIEL M. HOFFMAN AND DAVID M. WEISS, editors. “Software Fundamentals – Collected Papers by David L. Parnas”. Addison-Wesley (March 2001).