A Tool for Generating Specifications from a Family of Formal Requirements

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 are concerned with the maintenance of formal requirements documents in the application area of telephone switching. We propose a specification methodology that avoids some of the so-called feature interaction problems from the beginning, and that converts some more into type errors. We maintain all the variants and versions of such a system together as one family of formal specifications. For this, we define a formal feature combination mechanism. We present a tool which checks for feature interaction problems, which extracts a desired family member from the family document, and which generates documentation on the structure of the family. We also report on a quite large case study.

Keywords: tools; formal requirements; maintenance; feature interaction problems; telephone switching; Object-Z.

1 Introduction

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 [1–3]. 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. We will come back to this example later.

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

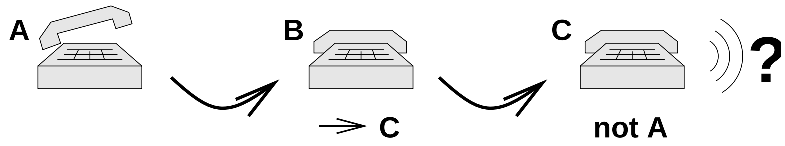

Fig. 1. Feature interaction between Terminating Call Screening and Call Forwarding.

problems are at least inherently present in the requirements documents. Therefore, they should be tackled already there.

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 a recent paper[4], we introduced the idea of *families of formal requirements*. Our approach proposes 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. 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. 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, such that they can be resolved.

A family of requirements needs a means for incremental specification. One standard means is refinement. In refinement approaches, a base system is specified first, which is then extended step by step by specifying further parts. The formalisms are constructed such that the interesting properties of previous steps are preserved by subsequent steps. For example, some approaches allow to add further possible behaviour to a behaviour description, while all existing behaviour remains possible. Other approaches allow to impose incremental, explicit constraints on an (implicit) behaviour description, which preserves all previous constraints due to the nature of the logical “and”. While these formal mechanisms are useful for many applications, they are not yet sufficient in our application area of telephone switching. In our experience with telephone switching requirements, most *new features change the behaviour of the base system or of other features.* “Change” here means that something really becomes different than it was before. For example, the Call Forwarding feature stops to connect to the dialled number and thus restricts the base system behaviour, and it starts connecting to the forwarded-to number, extending the base system behaviour. A telephony feature typically is *non-monotonous*.

Various feature operators have been proposed to allow non-monotonous changes. They allow to express the desired changes. If their definition and use reflect the current practice of modelling telephone switching features, feature interaction problems occur in such a formal setting in the same way as in an informal setting. A lot of research work has been done on a formal analysis of the changes [1–3, 5]. Our experience is that this indeed works to some extent, but that the exponential growth of complexity not only beats the human specifiers, but also beats the automated tools, when it comes to real-sized systems with hundreds of features. In particular, feature operators allowing arbitrary syntactic modifications of individual source lines lead to unmanageable feature interaction problems, since the complete system must be analyzed for the consequences.

We therefore attempt to reduce the complexity by using modularization in the information hiding sense for the requirements documents. We support it by a feature construct which allows for non-monotonous changes but which does not allow for arbitrary changes. In our experience, planning ahead for a careful modularization and for future potential modifications reduces the number and complexity of the dependences between features.

In Section 2, we briefly introduce to the formal specification language we use, and we present a suitable, new feature construct for it. Section 3 describes our tool support for managing features. Section 4 demonstrates the application of the approach and the tool in a case study from telephone switching, and Section 5 discusses how we avoided or detected feature interaction problems. Section 6 compares our approach to related work, and Section 7 summarizes this paper.

2 A Specification Methodology Supporting Families of Formal Requirements

In this section, we give a very brief overview of the formal specification language we use, concentrating on the means to structure a specification document. Another prerequisite is a suitable specification style, which we describe. We then add in a first step the concept of families to this language, and in a second step the concept of non-monotonous, but well-structured, changes.

2.1 Base Language CSP-OZ

We have chosen the formal specification language CSP-OZ [6, 7] as a base for adding our feature constructs. This language is suitable for telephone switching, since it supports communication events explicitly, and it is is suitable for specifying requirements, since it allows to specify the communication events that are observable externally, without prescribing any internal structure. CSP-OZ is a combination of the process algebra CSP (Communicating Sequential Processes) [8] and of Object-Z [9]. CSP-OZ allows to specify communication and control aspects in the CSP part, and data aspects separately in the Object-Z part. Such a separation has proven useful in the communications area. The Object-Z data part is in turn based on the language Z [10, 11], which is a formalization of set theory. Object-Z provides the notions of abstract data types and state invariants which are not present in CSP. Numerous algebraic laws have been derived for the refinement of behaviour in CSP and for the refinement of data in Z, providing a sound theoretical base to the inheritance operators we use.

The inheritance operator of CSP-OZ, in its basic form, allows to specify refinement. Optionally it can be supplemented by a renaming operator. If the latter is used, it effectively allows to change the behaviour of an inherited class completely. CSP-OZ inheritance thus offers either pure refinement or powerful, but uncontrolled, changes.

The Z language was defined for many years by Spivey's reference manual [11]. Currently, the International Standardization Organization (ISO) finalizes a standard for Z [10]. Besides providing a more detailed, well-structured definition of the syntax and semantics of Z, the standard also provides a few extensions to Spivey's Z. One of these extensions concerns the structuring of a specification document, and it supports our work considerably. We therefore already base our work on the forthcoming Z standard. Fortunately, the definition of CSP-OZ is also already based on the new Z standard.

A Z (and thus also a CSP-OZ) specification consists of paragraphs of formal text. A paragraph can be, for example, one type definition or one schema definition. The new standard allows to put a *specification* and *sections* on top of paragraphs. A specification consists of sections, which in turn consist of paragraphs. The meaning of a Z specification is the set of named theories established by its sections. Each section adds one named theory to the set established by its predecessors. A named theory associates a section name with a set of models.

As a consequence, each section has a self-contained formal meaning. Any initial sequence of sections can be taken as the requirements for a variant of the specified system. Since each further section can only add more constraints on the system (and new declarations), this allows for a constraint-oriented, incremental style of specification.

But there is no means for non-monotonous extensions. Z sections have a *parents* construct such that a section can be an extension of specifically named sections. The only way to get a modified version of a system is to copy, rename, and edit the changed sections, and also to copy and rename all sections that have these sections as a parent. If a section needs to be modified that is at the leaves of the hierarchy of parent sections, this can amount to a large part of the document being duplicated.

2.2 Specification Style

For requirements, we prefer a constraint-oriented specification style. Adding one (small) constraint after the other helps us to concentrate on one aspect of the system at the time, without accidentally being overly restrictive on other aspects.

When we perform non-monotonous extensions, it is important that the changes happen in a controlled way. Therefore, we choose that a feature may modify another feature at the level of Z sections. Leaving out an entire feature and introducing a new feature with a slight modification is too coarse-grained. If the feature is complex, this results in a considerable duplication of code. Allowing a feature to modify the specification text of another feature in an arbitrary way, e.g., by changing an integer value or by substituting an arithmetic operator, is too fine-grained. It is hard to understand the consequences of such changes, in particular if many of them are applied at the same time. This easily leads to feature interaction problems.

We distinguish between the *essential behaviour* and the *changeable behaviour* of a feature. Some parts of a feature are essential for its nature. Other parts are provided only in order to make the requirements specification complete. For example, some of the behaviour restrictions might only be made to make the behaviour predictable for the user. If the changeable behaviour is modified, this can never be an undesired feature interaction. When a specifier needs to modify a feature that was written some time ago by another person, he will find it difficult to find out which part of the behaviour is essential. Therefore, we require the original specifier of a feature to document which behaviour is essential and which behaviour is changeable, by using an appropriate language construct.

Furthermore, we distinguish between the requirements of the system and the requirements of its environment. This is important for any software development contract. It is the developer's duty to implement the former, and it is the customer's duty to ensure the latter, in order to make the system actually work. Formally, we collect both parts into one CSP-OZ class each, using inheritance. We then compose them by parallel composition into a description of the aspects of the world which are relevant to the contract.

2.3 Extension of CSP-OZ: Families with Monotonous Increments Only

In order to support families, we change the top-level structure of the language. We remove specifications from the syntax, and we add *families*, *feature chapters*, and *family member chapters* instead.

A family consists of feature chapters and of family member chapters. Feature chapters are (named) chapters of (named) sections. Family member chapters are named chapters that contain nothing but a features list. Informally, each family member chapter defines one specification in plain CSP-OZ. We can extract one family member from a family document by collecting all its features, and by concatenating all the sections of these features, resulting in a specification in plain CSP-OZ.

A detailed definition of the syntax and semantics of the language extension can be found in the manual [12].

The inheritance operator of standard CSP-OZ can be supplemented by a renaming operator, effectively allowing to change the behaviour of an inherited class completely. We therefore ban the use of the renaming option in our extension of CSP-OZ.

Semantics. The set *Features* contains all feature chapters' names. The set *Sections* contains all mappings from feature chapter names to sets of section names: $\mathit{Sections} == \mathit{Features} \ffun \mathbb{P}\,\mathit{Name}$. The meaning of a feature chapter can be determined by applying the function *Sections* to the feature chapter's name. The meaning of a family is a mapping from family member names to meanings of plain CSP-OZ specifications.

$$[\![\cdot]\!]^{\mathcal{F}} : \mathit{Family} \pfun (\mathit{Name} \ffun \mathbb{P}\,\mathit{Theory})$$

That is, each (pure) family member chapter is essentially a complete CSP-OZ specification, except that all of its actual sections have been moved into feature chapters, which can be shared with other family members.

Shorthand Notations. Optionally, a family member chapter may contain a feature chapter body. This is transformed syntactically into a separate feature chapter and a "pure" family member chapter. Such a feature chapter body can be convenient to specify the composition of all items from the different features in a suitable way.

CSP-OZ classes are larger units than mere (Z language) paragraphs. Therefore we found it sometimes convenient to use classes for structuring instead of sections. This is true in particular when each section contains only one class. Therefore, we allow a class in the places where a section can occur, too. When determining the semantics, we transform such a class into a section heading followed by this class, and followed by whatever other formal paragraphs may follow. The name of this implicit section is the name of the class. (Please note that family members, features, and sections each have separate name spaces.) The parents sections names of the section are the names of the classes inherited by this class.

2.4 Extension of CSP-OZ: Families with Non-Monotonous Increments

Sections are our unit of increment for monotonous increments, and they are also our unit of modification for non-monotonic changes. We distinguish between the essential and the changeable behaviour of a feature, and we specify this information explicitly by using different kinds of section. The type rules on the new constructs defined below will allow us to exploit the information for formal checks.

But one more kind of section is not sufficient. Both the essential and the changeable behaviour constraints need to be composed. The composition operators need to be grouped into sections too, and these operators specify neither essential nor changeable behaviour. They require a third kind of section.

We therefore add two more kinds of section. We put the essential properties of a feature into the normal kind of section. The two new kinds of section have the same syntax as the normal kind, except for the new keywords default_section and collect_section. *Default sections* are different insofar that they can be referenced by a *remove clause*, and they serve to express changeable properties. A remove clause is a another new construct that may appear in feature chapters. When the family member comprises a feature that contains a remove clause, the named default sections are disregarded when determining the set of sections. A *collect section* is special in that it adapts its list of parent sections automatically to losses of default sections due to remove clauses. A collect section should be used only to collect and combine properties defined in other sections.

We define six *type constraints* on these different kinds of sections which ensure their intended use, and which flag errors that may constitute feature interaction problems. A basic rule is that only changeable properties can be removed: 1) A remove clause may name default sections only. 2) The parent of a normal, i.e., essential, section must not be a default section, except if it is from a different feature or family member chapter.

The above exception is necessary to allow one feature to use a specific, non-essential version of another feature as a base. Note: a type checker may issue a warning if there is more than one remove clause for one default section in the features of a family member.

By further rules, parents clauses and remove clauses are restricted to sensible section names with respect to any specific family member: 3) For each family member m, we construct the set $SectionNames(m)$ of names of all its feature's normal sections, default sections, and collect sections. For each family member m, all remove clauses of its features must name sections from the set $SectionNames(m)$ only. 4) We also construct a reduced set $RSectionNames(m)$ from $SectionNames(m)$ by removing those default sections that are named in a remove clause of this family member. For any normal or default section in $RSectionNames(m)$, its set of parents sections must be a subset of $RSectionNames(m)$.

Finally, the varying set of defined items in collect sections cannot be referenced in property-defining sections: 5) Neither normal nor default sections may have a collect section as a parent. 6) The parents lists of collect sections must contain sections only that are either from the same feature or that are collect sections.

We also introduce a special kind of inheritance operator, called "default_properties", that is ignored in case the inherited class has been removed. We extend the syntax accordingly, and we add two more type constraints for the new inheritance operator: 7) The default properties operator may appear in a collect section only. 8) The class name in the operator must be a valid class name in the entire family document; but for any specific family member, the default properties operators in its sections may reference class names from outside of these sections.

A detailed definition of the syntax and semantics of this part of the language extension can be found, again, in the manual [12].

Semantics. The meaning of a remove clause is the set of its names of default sections. The function *Remove* maps feature chapter names to the sets of names of default sections which are named in a remove clause in this feature chapter: $Remove : Features \twoheadrightarrow\!\!\!\!\!\!+ \ \mathbb{P}\, Name$.

The features list of a family member chapter determines the features that are used to construct its CSP-OZ specification. The features list is a subset of the set *Features*. Its meaning is the union of all the sets of section names of the features listed, minus the set of default sections that are removed by the features listed.

$$\begin{aligned}[\![fe_1 \dots fe_n]\!]^{\mathcal{FL}} = {} & (Sections(fe_1) \cup \dots \cup Sections(fe_n)) \setminus \\ & (Remove(fe_1) \cup \dots \cup Remove(fe_n))\end{aligned}$$

The sections of a family member chapter's specification are all the sections $[\![fe_1 \dots fe_n]\!]^{\mathcal{FL}}$. The only modification is that we remove all section names in $(Remove\ fe_1 \cup \dots \cup Remove\ fe_n)$ from the sets of parents sections. The meaning of the specification is then constructed in the usual way from these sections.

Shorthand Notations. When classes are used for structuring instead of sections, a class is transformed into a collect section instead of a normal section exactly if it contains a "default_properties" operator. A class is transformed into a default section exactly if there exists a "default_properties" operator somewhere in the same feature that references its name.

3 The Tool

Our approach is supported by a tool, called *genFamMem 2.0*, which

- extracts specifications in plain CSP-OZ from a family document,
- detects feature interactions by
  - performing the additional type checks for families like the constraints on normal, default, and collect sections, on "remove", and on the "default_properties"/ "inherit" operators,

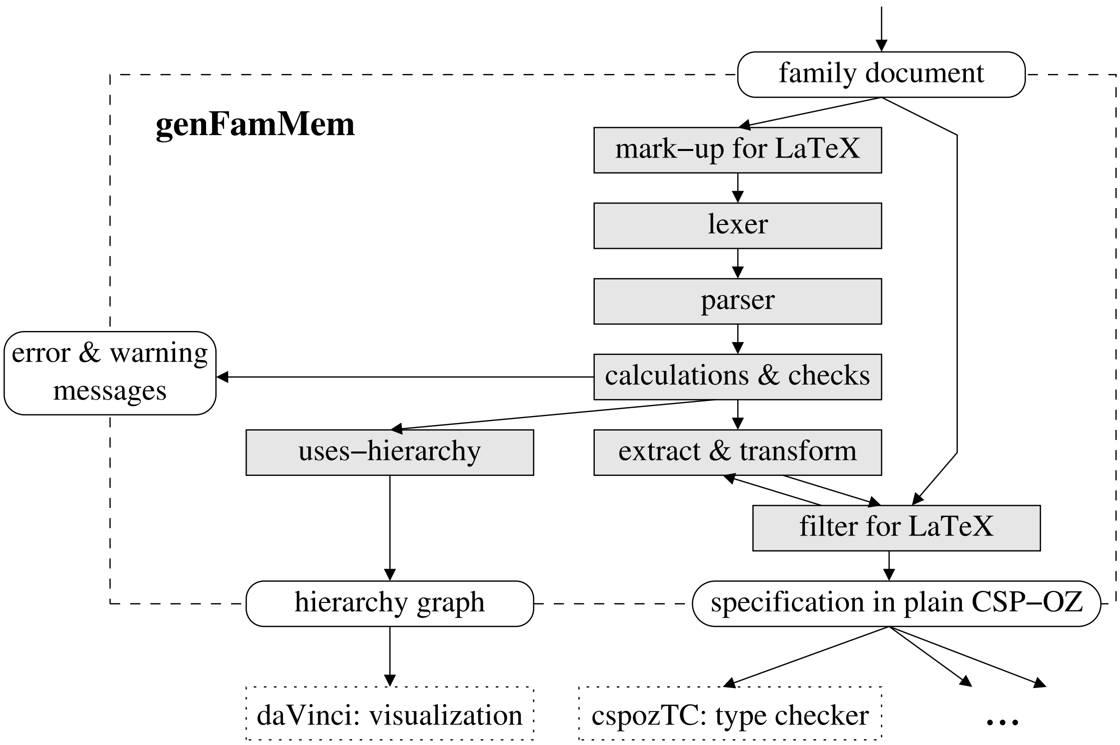

Fig. 2. Structure of the genFamMem tool.

- issuing some heuristic warnings, for example when two different remove clauses remove the same class, and

– helps avoiding feature interactions by generating documentation on the structure of the family.

The tool contains a lexer/parser for our extension of CSP-OZ which is written using the lex/yacc parser generator [13], and C code. The tool comprises about 8500 lines of commented source code. Figure 2 shows its module structure. The tool has a modular structure that reflects the well-structured definition of the Z semantics in the forthcoming standard. The standard defines the syntax and the semantics in nine phases, the first ones being mark-up, lexing, and parsing, and the last one being the semantic relation. The separate mark-up phase solves the problem that Z provides many more special symbols than normal keyboards can provide, by encoding them as sequences of characters. The standard defines two mark-ups, LaTeX markup and email mark-up.

The mark-up module takes a family document for a first pass and translates LaTeX control sequences into CSP-OZ characters like "$\mathbb{F}$", "$\pfun$" etc. It could be replaced by mark-up modules for other mark-ups. The next steps are the lexical and the syntactic analysis. The semantic calculation module takes the information from the syntactic analysis, and the name of a family member, and it calculates which sections should be included into the generated specification. It also performs the type checks described above, and flags any type errors in the family document structure, indicating feature interaction problems. When no type errors occurred, the tool extracts the actual specification document in a second pass, while adjusting the parents clauses, turning the default properties clauses into appropriately reduced inherit clauses, and generating the

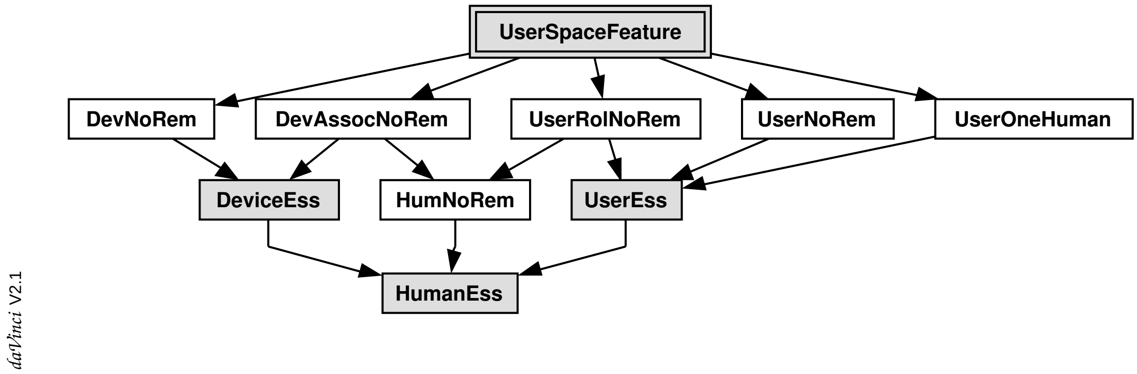

Fig. 3. The uses-hierarchy of the sections of feature UserSpace from the case study [16, 17].

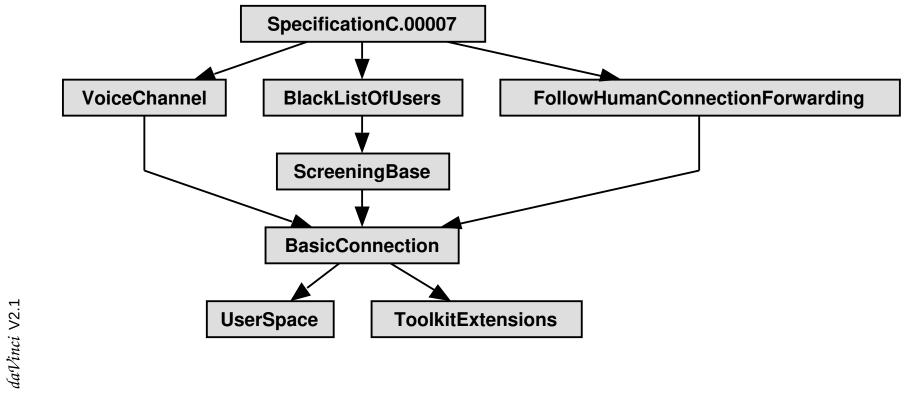

Fig. 4. The uses-hierarchy of family member SpecificationC from the case study, abstracted to features.

final family member collect section (discussed below in Section 4.4). After that, we type-check the resulting requirements document in plain CSP-OZ using von Garrel's CSP-OZ type checker cspozTC [14].

Besides generating and checking family members, the tool genFamMem also has another mode of operation, in which it generates documentation on the structure of the family. It extracts the parents relations of all sections and the inheritance relations of all classes, it extracts the association of sections to features, and then it constructs and outputs a *uses-hierarchy graph*. The uses-hierarchy graph describes which section needs which sections as prerequisites. This graph then is visualized graphically using the tool daVinci [15]. We have the choice to see the uses-hierarchy of all sections of the family (which is large!), of one family member only, or of one feature only. Furthermore, we have the choice to abstract sections to features, such that we see the uses-hierarchy of all features of the family, or of all features of one family member.

For example, Figure 3 presents the uses-hierarchy graph of the sections of feature UserSpace from our case study [16, 17]. Plain grey boxes denote essential sections, white boxes denote changeable sections, and double border boxes denote collect sections. Since the uses-relation is transitive, redundant arrows are not shown, such as the one from DevAssocNoRem to HumanEss. Figure 4 shows the uses-hierarchy of family member SpecificationC from our case study, abstracted to features.

Our extensions can be defined not only for CSP-OZ, but also for plain Z and for CSP_Z. The latter is a dialect of CSP that uses Z for data definitions, but does not offer object-based constructs like classes and inheritance. Our tool already supports Z and CSP_Z, through switches which optionally disable the more advanced language constructs. We have not conducted experiments with families of Z or CSP_Z specifications yet, but plan to do so.

The tool is available without charge through its Web home page [18].

4 Case Study

We specified the requirements for a telephone switching system in a case study [16, 17], using our extension of CSP-OZ. The case study currently comprises about 40 pages of commented formal specification, with about 50 sections in nine features, including the base system. We demonstrate the new structuring means using excerpts from two simple features and from the base switching system.

The family of requirements from the case study can be used to generate requirements specifications with different combinations of features. Such a requirements specification can then be the starting point for a software development process delivering an implementation for this family member.

4.1 The Base Telephone Switching System

The base concepts of telephony are at the bottom of the class inheritance hierarchy. The class HumanEss introduces the set of humans, modelled as the variable humans. Classes are written in CSP-OZ as a box, with the class name in the heading. We write the state space of the class in a contained, unnamed box. The (essential) class UserEss introduces the set of user roles users of humans and also the association of humans to user roles userRoles:

```
section HumanEss    [ Humans ]

    parents

...
  __HumanEss________________
 |  ...
 |   ______________________
 |  | humans : 𝔽 humanIds
 |  |______________________
 |__________________________
```

```
section UserEss    [ User Roles ]

    parents HumanEss

...
  __UserEss____________________________
 |  inherit HumanEss
 |  ...
 |   _________________________________
 |  | users : 𝔽 userIds
 |  | userRoles : humanIds → 𝔽 userIds
 |  |_________________________________
 |_____________________________________
```

The class DeviceEss introduces the set of terminal devices devices and also the association of humans to devices possessedDevices. The section and class DeviceEss is very similar to UserEss, we do not reproduce it here.

In order to get a plain old telephone system, we specify next that the elements of these sets are all in a one-to-one relation, and that these sets are static. For example, the association between humans and devices should never change:

`default_section` **DevAssocNoRem** [**Telephone Device Associations neither Added nor Removed**]

DevAssocNoRem

`inherit` HumanEss, DeviceEss, ...

$$\forall\, h : \mathsf{humans} \bullet \\ \quad \mathsf{possessedDevices}(h) = \mathsf{initialPossessedDevices}(h)$$

The specification describes furthermore how a connection is set up and torn down. Due to space limitations, we leave out this rather large description [16, 17] here.

4.2 A Black List Feature

A black list feature protects its subscriber from calls of users that she has put on a black list. We specify the black list feature in two steps. First, we specify a feature ScreeningBase that only serves as a “hook” which allows to block connections according to some criteria. The variable connectionScreen is defined in section ConScreenEss, it describes the pairs of users which shall never be connected. We specify this screening as a kind of missing resource for a connection. A default section ConNoScreen of this feature then states that the screening condition is never satisfied.

In a second step, we specify one specific kind of black list feature BlackListOfUsers, including means to add users to the list and to delete them again. Due to the size of this definition [16, 17], we omit it here. But we present how we couple the feature blackListUsr to the general screening above:

`feature` **BlackListOfUsers**

...

`remove` ConNoScreen

`default_section` **BlackListUsrOnly** [**User Black List is the only Screening Feature**]

BlackListUsrOnly

`inherit` ..., ConScreenEss

$$\forall\, u, v : \mathsf{users} \bullet \\ \quad ((\mathsf{connectionScreen}(u, v) = \mathsf{connectionScreenBlock}) \\ \quad \Leftrightarrow ((u, v) \in \mathsf{blackListUsr}))$$

`collect_section` **BlackListOfUsersFeature** [**Collection of all the Requirements**]

parents ..., BlackListUsrOnly, ...

```
__ BlackListOfUsersFeatureSys ________________________
| inherit ...
| default_properties ..., BlackListUsrOnly, ...
|______________________________________________________
```

```
__ BlackListOfUsersFeatureEnv ________________________
| ...
|______________________________________________________
```

4.3 A Call Forwarding Feature

The feature FollowHumanConnectionForwarding allows a human to move to another telephone device and register himself there, in order to have all of his calls forwarded there. We specify this feature mainly by lifting the restriction of a static association between humans and devices introduced above.

`feature` **FollowHumanConnectionForwarding**

...
`remove` DevAssocNoRem

In order to facilitate the implementation of these requirements, the feature also demands that a registration may be performed only when no connection to the human exists. Again, we omit the details [16, 17] due to space limitations.

4.4 Composition of a Complete System

The above two features, and some more, can be combined into the specification of the requirements of a complete system:

`familymember` **SpecificationC**

`features` ..., BlackListOfUsers, FollowHumanConnectionForwarding

The collect sections of the listed features are composed in a single collect section of the family member. The structure of this collect section always follows a fixed pattern: one class inherits all requirements on the system, a second class inherits all requirements on the environment, and in a third, final class the first two classes are composed by a parallel composition operator:

$main = \mathsf{SpecificationCEnv} \parallel \mathsf{SpecificationCSys}$

Since this final collect section has such a fixed structure, it can be omitted, the semantics definition of the language then derives it from the features list. The entire family member chapter then consists of the features list only. Accordingly, our genFamMem tool fills in this collect section when generating a family member in plain CSP-OZ.

Similarly, we do not need to specify the parents sections of a section under certain circumstances, they are then derived from the inheritance operators. We omit the formal details [12] of these shorthand notations here.

Performing the Generation. We use the tool genFamMem to extract the above family member from the family document. The command

```
genFamMem SpecificationC case-study.tex > SpecificationC.tex
```

extracts SpecificationC and performs the type checks, and the command

```
genFamMem --uhier-familymember SpecificationC --only-features \
          case-study.tex > SpecificationC.daVinci
```

extracts the uses-hierarchy graph shown a few pages above in Figure 4.

5 Avoiding and Automatically Detecting Feature Interactions

The goal of our approach is to either 1) avoid feature interaction problems in requirements documents from the beginning, or 2) to help detect them, such that they can be resolved, by

- allowing to maintain all variants of a requirements document together, by
- using a specification style that encapsulates variant choices, and by
- allowing/urging the original specifier to document as much of the information as possible that another specifier of an incremental feature will need.
  The information provided formally by a specification written in CSP-OZ with our extension comprises
  - the *dependences* among the features, through the hierarchy of parents sections sets and of class inheritance, and
  - what is *the core of a feature*, by the distinction between default sections, which are changeable, and normal sections, which are essential.

  This information can help the maintenance specifier to avoid problems, and it can be used for automated type checks, as performed by our generator tool. The detected feature interactions then can be resolved by the specifier.

Therefore, thanks to the additional explicit information in the document, our approach converts some of the feature interaction problems into *type errors*.

5.1 Example of Feature Interaction Avoidance

In our case study, we explicitly introduce the notions of telephone devices, humans, and user roles, even though we then state a static association between these for the Plain Old Telephone Service. Nevertheless in the definition of the screening feature presented above, we consequently must specify that user roles are screened, not devices. Furthermore, we encapsulated this variant choice by specifying separately a screening base feature and a user screening feature. We also specified another variant, a device screening feature not presented in this paper, which also builds on the screening base feature.

Similarly, our case study comprises two kinds of call forwarding features. One registers a human with another device, it is presented above. The other feature transfers a user role to another human. An example use case would be a help desk employee who transfers his help desk user role to a colleague while having a coffee break. His boss can still reach him in his role as an employee, even though he is not disturbed by customers.

We can now come back to the standard example of a feature interaction between Terminating Call Screening (TCS) and Call Forwarding (CF) from the introduction. Due to our separation of notions, for each combination of forwarding/screening features it is clear whether the phone should ring. A subscriber of BlackListOfUsers will be protected against a connection from a screened user from all devices, at the device where she is registered currently. A subscriber of BlackListOfDevices will not expect to be protected from humans that are able to use different devices. We just don't have the ambiguous notion of "caller" anymore.

5.2 Example of Feature Interaction Detection and Resolution

Our case study does not have any feature interaction problems between Terminating Call Screening and Call Forwarding, and therefore we cannot demonstrate any feature interaction detection using this standard example. Nevertheless, our type rules defined above point us to some remaining problems.

For example, both screening features "remove" the same changeable section ConNoScreen of the screening base feature (compare Section 4.2, first paragraph). Each feature then states in one new, essential section when no connection must be possible. The type rules flag this double removal as a warning. A manual analysis shows immediately that the two replacing, simple sections are not contradictory. But a further look reveals that both features also have a changeable section each that states that its respective feature is the only screening feature, such that otherwise no screening happens. This of course is a contradiction in the general case.

After we have detected a feature interaction problem, we need to resolve it. As in most cases, the harder part was to detect the problem before a customer notices it, and the relatively easy part is to find a specific remedy. We specify this remedy as just another feature which we select whenever both of the above features are included. It contains a section that states that screening happens exactly when either of the two screening features demands it. Furthermore, it removes both conflicting sections. We are explicitly allowed to do this since both sections are marked as changeable.

5.3 Conventional Further Consistency Checks

After we have generated one specific requirements document, all the conventional methods and tools can be applied which check the consistency of formal requirements. We used the CSP-OZ type checker cspozTC [14] to find conventional type errors ("bugs"), but they are of no interest here.

6 Related Work

Related work investigates "feature" constructs for different specification and programming languages, in particular in the context of the recently terminated FIREworks project [19]. The aim of this project is to define a feature construct for several languages, and then to evaluate it in case studies. The approach bases on the superimposition idea of Katz [20], where we specify a base system first, and where we then specify textual increments by a suitable operator. For example Plath and Ryan [21] do this, they define a feature construct for CSP. They use a state-oriented specification style, and their feature operator allows to add or replace transitions at states. They don't define type rules for their operator, instead they detect feature interactions by model-checking the combined systems. 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. We try to reduce these problems by encapsulating one self-contained property each in one section, and by either including or excluding it entirely.

Also related is the foreground/background approach of Hall [22]. He separates the description of a feature into its foreground behaviour, which is "essential" behaviour in our terms, and its background behaviour, which is "changeable" by another feature's foreground behaviour without causing a warning by his analysis tools. His goal is to perform feature interaction detection with *less* results; i.e., his tools do not report "spurious" interactions. His approach does not provide multiple layers of features, as ours does, even though he mentions a generalization to an "n-ground" approach. We use a hierarchical, constraint-oriented structuring of features, and we are in particular concerned about avoiding feature interactions. Hall attempts to cope with current telephony system architectures, while we investigate a more suitable architecture.

7 Summary

We are concerned with the maintenance of formal requirements documents. In the application area of telephone switching, so-called feature interaction problems have become a serious obstacle to evolve these systems according to market needs.

We propose a specification methodology that either 1) avoids feature interaction problems in requirements documents from the beginning, or 2) helps to detect them, such that they can be resolved. The original specifier of a feature must document additional information on what is the core of the feature, and what are the dependences on other features. Keeping all the variants and versions of a system as a single family enables us to exploit this information.

For specifying families, we present in this paper (and in full detail in the manual [12]) the syntax and semantics of a formal feature combination mechanism. It is an extension to the specification language CSP-OZ. In our formalism, some feature interactions appear as type errors.

In this paper, we also present a tool which checks for feature interaction problems, which extracts and transforms a desired family member from the family document, and which generates documentation on the structure of the family. The tool is available freely.

We finally report on a quite large case study. We show how a standard example of feature interactions does not occur due to the modular structure we use, and we show how our type checker detects a further feature interaction, which we then resolve.

Several issues are worth further research. More experience with our methodology is important; we plan to extend our case study further. Our approach should be applicable with other formalisms than CSP-OZ as well, as long as the base formalism used allows for a constraint-oriented specification style and provides a means for incremental refinement. Our tool even already supports the languages $\mathrm{CSP_Z}$ and Z. Finally, the relationship between families of requirements and families of programs needs to be investigated, with respect to a reuse-oriented implementation process.

References

1. L. G. Bouma, Nancy Griffeth, and Kristofer Kimbler. Special issue: Feature interactions in telecommunications systems. *Comp. Networks* **32**(4) (April 2000).
2. Muffy Calder and Evan Magill, editors. "Feature Interactions in Telecommunications and Software Systems VI". IOS Press, Amsterdam (May 2000).
3. Kristofer Kimbler and L. G. Bouma, editors. "Feature Interactions in Telecommunications and Software Systems V". IOS Press, Amsterdam (September 1998).
4. Jan Bredereke. Families of formal requirements in telephone switching. In Calder and Magill [2], pages 257–273.
5. Jan Bredereke. "Communication Systems Design With Estelle – On Style, Efficiency, and Analysis". PhD thesis, University of Kaiserslautern, Shaker Verlag, Aachen, Germany (August 1997).
6. Clemens Fischer. Combination and implementation of processes and data: from CSP-OZ to Java. PhD thesis, report of the Comp. Sce. dept. 2/2000, University of Oldenburg, Oldenburg, Germany (April 2000).
7. 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).
8. A. W. Roscoe. "The Theory and Practice of Concurrency". Prentice-Hall (1997).
9. Graeme Smith. "The Object-Z Specification Language". Kluwer Academic Publishers (2000).
10. ISO Panel JTC1/SC22/WG19, Final Committee Draft, CD 13568.2. "Z Notation" (August 1999).
11. John Michael Spivey. "The Z notation: a reference manual". Series in Computer Science. Prentice-Hall, New York, 2nd edition (1995).
12. Jan Bredereke. "genFamMem 2.0 Manual – a Specification Generator and Type Checker for Families of Formal Requirements". University of Bremen (October 2000). URL http://www.tzi.de/~brederek/genFamMem/.
13. Levine, Mason, and Brown. "Lex & Yacc". O'Reilly & Associates Inc (1992).
14. Jens von Garrel. Parsing, Typechecking und Transformation von CSP-OZ nach Jass. Masters thesis, University of Oldenburg, Germany, Dept. of Comp. Sci. (July 1999).
15. Mattias Werner. "daVinci V2.1.x Online Documentation". University of Bremen (June 1998). URL: http://www.tzi.de/~davinci/doc_V2.1/.
16. Jan Bredereke. Modular, changeable requirements for telephone switching in CSP-OZ – revision 2.0. Tech. Rep. IBS-00-1, University of Oldenburg, Oldenburg, Germany (2001). *To appear.*
17. Jan Bredereke. Modular, changeable requirements for telephone switching in CSP-OZ. Tech. Rep. IBS-99-1, University of Oldenburg, Oldenburg, Germany (October 1999).
18. Jan Bredereke. "genFamMem 2.0 Home Page". University of Bremen (2000). URL http://www.tzi.de/~brederek/genFamMem/.
19. 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.
20. Shmuel Katz. A superimposition control construct for distributed systems. *ACM Trans. Prog. Lang. Syst.* **15**(2), 337–356 (April 1993).
21. Malte Plath and Mark Ryan. Defining features for CSP: Reflections on the feature interaction contest. In Gilmore and Ryan [19], pages 55–69.
22. Robert J. Hall. Feature combination and interaction detection via foreground/background models. *Comp. Networks* **32**(4), 449–469 (April 2000).