Universität Bremen · FB 3

genFamMem 2.0 Manual

– a Specification Generator and Type Checker for Families of Formal Requirements

Jan Bredereke

University of Bremen, FB 3
P.O. box 330 440
D-28359 Bremen
Germany

E-Mail: brederek@tzi.de
WWW: http://www.tzi.de/~brederek

Last modified: October 30, 2000
Document Revision: 1.1

Abstract

We define the syntax and the semantics of a feature construct for the formal specification language CSP-OZ. Our extension allows to specify a family of formal requirements in one document. The intended application area is to avoid or to detect feature interaction problems in telephone switching systems. Our extension converts some of these problems into type errors. CSP-OZ has been defined recently, it is a combination of CSP and Object-Z. Our extension itself is of a more general nature, since – while on our way – we define it for the well-known formal specification languages Z and CSP, too.

We then describe the tool genFamMem which implements our extension.

Contents

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. 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.

Feature interactions already arise in the requirements documents. When these documents are a *complete* description of the behaviours (and of other interesting properties), then even *all* feature interaction problems are at least inherently present in the requirements documents [1]. 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[1], 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.

While the above paper presents the methodological aspects of the approach, it describes the actual syntax and semantics of our language constructs only informally. In this manual, we define precisely the language constructs which we use for describing features and for describing family members constructed from features. In addition, we apply our extension to two more specification languages.

We have chosen the formal specification language CSP-OZ [2, 3, 4] 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) [5, 6] and of Object-Z [7, 8]. Object-Z provides us with the notion of inheritance, which is a means to express incremental extensions of a specification. 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 [9, 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.

In Section 2, we present an overview of plain CSP-OZ as it was defined by Fischer [2, 3, 4]. Fischer defines CSP-OZ in two steps as an extension of Z to the language $\mathrm{CSP_Z}$ and then as an extension of $\mathrm{CSP_Z}$ to CSP-OZ. Accordingly, we define our feature construct as an extension to Z in Section 3, and then lift it to $\mathrm{CSP_Z}$ in Section 4 and to CSP-OZ in Section 5. In the latter section, we also define one more operator dealing with inheritance, and we define two handy shorthand notations that require the notion of classes. Section 6, finally, presents the actual genFamMem tool which supports the three extended languages, and Section 7 contains a summary.

2 Overview of Plain CSP-OZ

Since we cannot present the entire definition of CSP-OZ here, we introduce to the relevant language aspects by a brief, didactic example. Additionally, we discuss how we use CSP-OZ for specifying requirements, since this motivates one of the extensions presented further below. Finally, we sketch how the syntax and the semantics are defined, in order to introduce the "hooks" for our extension of CSP-OZ.

2.1 Introductory Example

CSP specifications are defined over events. The specification of a (very much) simplified telephone system in Figure 1 has five such communication events, offHook, onHook, dial, connect, and disconnect. The "`main`" line in the figure describes a process that repeatedly does these five events in a sequential order. A process describes a (black-box) system in a certain state which is prepared to engage in one (or more or no) events, thereby transforming into another process. The system will refuse other events. The specification of a process may be composed of other processes. For example, different parallel composition operators are available which allow for different degrees of synchronization on the communication events.

An Object-Z class consists of the specification of a state space and of operations on this state space. In class SimplePhone in Figure 1, dialledNum is declared as a variable ranging over the natural numbers, with the additional constraint that it always must be between 0 and 3. In the Object-Z view, this class has four operations, enable_connect, effect_connect, effect_dial, and enable_dial.

In CSP-OZ, the operations of the Object-Z part are interpreted as CSP events and impose additional constraints on the CSP behaviour. The operation enable_connect demands that the value of the variable dialledNum must be in the set $\{1, 2, 3\}$ for the event connect to take place. This event has an output parameter "$n!$", and the operation effect_connect demands that this parameter must be set to the value of the variable dialledNum. The event dial has an input parameter "$n?$", and the operation effect_dial sets the variable

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

SimplePhone

chan offHook   chan dial : [$n?$: telNums]   chan onHook   [Input events.]

chan connect : [$n!$: telNums]   chan disconnect   [Output events.]

`main` = offHook → dial → connect → onHook → disconnect → `main`

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

Init

$\mathsf{dialledNum} = 0$

enable_dial

$n? : \mathsf{telNums}$

true

effect_dial

$n? : \mathsf{telNums}$

$\mathsf{dialledNum}' = n?$

enable_connect

$n! : \mathsf{telNums}$

$\mathsf{dialledNum} \in \{1, 2, 3\}$

effect_connect

$n! : \mathsf{telNums}$

$n! = \mathsf{dialledNum}$

Figure 1: A CSP-OZ example

dialledNum to its value in the after-state, denoted by the prime ("′"). The enabling condition for the event dial, specified by the operation enable_dial, does not impose any restriction, since it is "true". As an extension to Object-Z, CSP-OZ allows to mark events as either controlled by the system or as monitored by the system only. Monitored events have "true" as the enabling condition.[1] The variable dialledNum is not part of the semantics of this specification, it is merely an auxiliary construct to restrict the range of output values associated with the connect event. The semantics of CSP-OZ is defined entirely in terms of (parameterized) events.

As a general rule of our approach, we use a constraint-oriented specification style. CSP-OZ offers (multiple) inheritance of classes, and we use it for specifying incremental constraints. In Figure 2, the class BlackList introduces a black list of telephone numbers by whom the user never wants to be called. It defines a variable blackList that can hold

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

```
BlackList
chan addBList : [ n? : telNums ]                                   [Input event.]

  blackList : ℙ({0, 1, 2, 3})       [Variable that keeps the set of undesired callers.]

  Init                                  effect_addBList
  blackList = {}                        n? : telNums
                                        ───────────
                                        blackList = blackList ∪ {n}
```

```
BlackListPhone
inherit SimplePhone BlackList                 [Combine these classes by conjunction.]

  enable_connect
  n! : telNums
  ───────────
  dialledNum ∉ blackList                   [Add a further constraint to this operation.]
```

Figure 2: Inheritance in CSP-OZ, used for incremental constraints

any subset of telephone numbers, and an event addBList that allows the user to add a number to the black list. For brevity, we omitted further details. The class BlackListPhone composes the class SimplePhone from Figure 1 with the class BlackList by inheriting both, and by adding a further, connecting constraint.

The inheritance operator of CSP-OZ merges all variable definitions (which must be distinct), and it composes operations present in more than one class by a logical "and" of the respective predicates. For example, the operation enable_connect appears both in the class SimplePhone and in the class BlackListPhone. Thus the event connect may happen only when both restrictions on the variable dialledNum are met. If CSP parts were present, they would be composed by parallel composition, which is the CSP version of conjunction.

2.2 Using CSP-OZ for Specifying Requirements

When we use classes for specifying requirements in CSP-OZ, we specify one class in the end that inherits all other classes, like the class BlackListPhone in Figure 2. The interface of this class is the interface of the specified system, and the behaviour observable at this interface is the behaviour observable of the specified system. This usage style corresponds to the black-box view of requirements specification, where we don't restrict the internal structure of the specified system.

To be more precisely, we don't use only one class to collect all properties, as in the above didactic example, but *two* such classes. One class collects all the requirements of the system, and the other class collects all the requirements of the environment of the system. The idea behind this approach is that a requirements document is used as the base of a contract between a customer and a developer, and that the system to construct can provide some sensible functionality only if it can rely on some assumptions about its environment. It is the customer's duty to assure that the stated requirements of the environment are met, and then it is the developer's duty to provide a system that meets its stated requirements in this environment.

In addition to the two collecting classes, we define a final class that composes the system class and the environment class by parallel composition like this:

SpecificationCSys

inherit ... [Collect all system requirements.]

SpecificationCEnv

inherit ... [Collect all environment requirements.]

SpecificationC

inherit-interface SpecificationCEnv

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

We then also have a broader interpretation of the meaning of the specification. On the one hand, the last class specifies the behaviour of a world composed out of the system and the environment. On the other hand, the top-level parallel operator separates two distinct requirements specifications. One specifies the requirements of the system, and the other the requirements of the environment.

2.3 Syntax and Semantics Definition for CSP-OZ

The language CSP-OZ is defined in two stages. $\mathrm{CSP_Z}$ is a formally defined dialect of standard CSP which adds a Z notation for types, expressions, and parameters. CSP-OZ then adds classes and inheritance to $\mathrm{CSP_Z}$. We now sketch how the syntax and the semantics of these languages are defined, in order to prepare the grounds for our extensions.

Syntax and Semantics of Z

Currently, the International Standardization Organization (ISO) develops a standard for Z, and has produced a final committee draft in August 1999 [9]. $\mathrm{CSP_Z}$ is already based on the ISO draft standard for Z.

Traditionally Z uses *paragraphs* for structuring. One of the novelties introduced by the standard is to put *specifications* and *sections* on top of paragraphs. We will make use of this additional structure for our extension. It is not yet so well known, and we describe it here. A specification consists of sections, which in turn consist of paragraphs.

$$\begin{array}{rcl}
\mathit{Specification} & ::= & \{\mathit{Section}\} \mid \{\mathit{Paragraph}\} \\
\mathit{Section} & ::= & \mathit{SectionKeyword}\ ,\mathrm{NAME}\ ,\mathit{ParentsList}\ ,\mathrm{END}\ , \\
& & \{\mathit{Paragraph}\} \\
\mathit{SectionKeyword} & ::= & \mathrm{section} \\
\mathit{ParentsList} & ::= & \mathrm{parents}\ ,[\ \mathrm{NAME}\ ,\{\ ,\text{-tok}\ ,\mathrm{NAME}\ \}\ ]
\end{array}$$

A paragraph can be, for example, one type definition or one schema definition. The *SectionName* clause defines the name of a section, and it also refers to the parents of the section; usually this is just the preceding section. A specification may consist of paragraphs instead of sections, too. In this case, there is a single, implicit, anonymous section.

The standard defines the syntax and the semantics of Z in nine phases, the first ones being mark-up, lexing, and parsing, and the last one being the semantic relation. Since the mark-up of a specification is defined in a separate phase, different mark-ups are possible. The standard defines two mark-ups, LaTeX markup and email mark-up. Usually a section contains informal, explanatory text between its paragraphs, too. When determining the formal meaning, the informal text is eliminated in the mark-up phase. For example, the LaTeX markup defines that each formal Z paragraph appears between a pair of `\begin{xxx}` and `\end{xxx}` commands. Text not appearing between such commands is informal accompanying text. We will need to handle such informal text in families of requirements specifications, too.

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.

$$\mathit{Theory} \quad == \quad \mathit{Name} \mathrel{\twoheadrightarrow\!\!\!\!\!\!\shortmid\;\;} \mathbb{P}\,\mathit{Model}$$

A model associates Z names with values from the semantic universe.

Syntax and Semantics of $\mathrm{CSP_Z}$

Fischer [2] defines $\mathrm{CSP_Z}$ by adding two new kinds of paragraphs to Z: channel declarations and processes.

$$\mathit{Paragraph}_C \quad ::= \quad \mathit{Paragraph}_Z \mid \mathit{Channel} \mid \mathit{Process}$$

Thus every Z specification is also a $\mathrm{CSP_Z}$ specification.

The semantics definition of $\mathrm{CSP_Z}$ is basically the same as the semantics definition of Z, except that it extends the semantic function for expressions $[\![\cdot]\!]^{\mathcal{E}}$ by a semantic function for process expressions.

$$[\![\cdot]\!]^{\mathcal{O}} \quad : \quad \mathit{ProcExpr} \rightarrow\!\!\!\!\!\!\shortmid\;\; (\mathit{Model} \rightarrow \mathit{LTS})$$

It provides an operational semantics for process expressions in terms of labelled transition systems (LTS). There exists also an alternative, provably consistent denotational semantics $[\![\cdot]\!]^{\mathcal{D}}$ which defines the traces, failures, and divergences for every process.

Syntax and Semantics of CSP-OZ

CSP-OZ adds two further kinds of paragraphs for two types of classes to the list of $\mathrm{CSP_Z}$ paragraphs.

$$Paragraph_O \quad ::= \quad Paragraph_C \mid Class_O \mid Class_C$$

For both types of classes, the syntax of a class may comprise a declaration of its inherited classes.

$$\begin{array}{rcl} InheritedClasses & ::= & [\ \text{inherit}\ ]\ ,\ InheritedClassesNames\ ,\text{END}\ \mid \\ & & \text{inherit-interface}\ ,\ InheritedClassesNames\ , \\ & & \text{END} \\ InheritedClassesNames & ::= & InheritedClassesName\ , \\ & & \{\ \text{,-tok}\ ,\ InheritedClassesName\ \} \\ InheritedClassesName & ::= & Word\ ,\ [\ \text{[-tok}\ ,\ ActualParam\ ,\ \text{]-tok}\ ]\ , \\ & & [\ \text{(-tok}\ ,\ Expr\ ,\ \text{)-tok}\ ]\ ,\ [\ Renaming\ ] \end{array}$$

The keyword "inherit" is optional for compatibility with Object-Z, but its use is encouraged in CSP-OZ. The keyword "inherit-interface" is used if only the interface of a class is inherited. Multiple inheritance is supported. The inheritance operator of CSP-OZ has optional parameters, but we don't use them for reasons explained in Section 5.4 below.[2]

Smith [7] defines the semantics of Object-Z expressions by a translation to Z. Fischer essentially follows the same approach by translating CSP-OZ to $\mathrm{CSP_Z}$, for which he re-uses Smith's translation. Additionally, Fischer needs to extend $\mathrm{CSP_Z}$ to deal with object references, and he defines appropriate new translations for CSP-OZ specific aspects.

The meaning of (multiple) inheritance in CSP-OZ is roughly the logical conjunction of all inherited predicates, and the parallel composition of all inherited CSP parts. Interestingly, parallel composition and logical conjunction turn out to be essentially the same in this context. Both add more constraints.

3 $\mathrm{Z_F}$: Z with a Feature Construct

We define the language $\mathrm{Z_F}$ in this section by adding a feature construct to the language Z. The next sections will then merge the extensions of CSP-OZ step by step into $\mathrm{Z_F}$ in order to define $\mathrm{CSP\text{-}OZ_F}$.

[2]Fischer's original definition of the inheritance operators allowed one name of an inherited class per inheritantance operator, only. This leads to visual clutter in the case of multiple inheritance because of repeated inheritance operators. Therefore we added the possibility of multiple inherited classes above, separated by commas, and terminated by an END token.

3.1 Syntax and Informal Semantics

One idea behind Z_F is to maintain many variants and versions of a requirements specification in one document. In particular, when a specifier creates a new variant, he should have as much information as possible, in a concise form, about all the other variants. We hope that this will reduce the chance of mistakes that lead to feature interaction problems. It also enables a formal analysis of the compatibility of the variants, as discussed in [1]. All the variants form a family of requirements specifications. Our feature construct aims at extracting common parts of these specifications, in order to avoid duplication of information and thus to avoid the resulting maintenance problems about keeping the information consistent.

Basic Families

The language Z_F is like the language Z, except that we change the top-level structure. We remove specifications from the syntax, and we add *families*, *feature chapters* and *family member chapters* instead. The start symbol of the syntax grammar is now *Family*.

$$
\begin{array}{rcl}
\mathit{Family} & ::= & \{\ \mathit{FeatureChapter} \mid \mathit{FamilymemberChapter}\ \} \\
\mathit{FeatureChapter} & ::= & \mathrm{feature}\ ,\mathrm{NAME}\ ,\mathit{FeatureChapterBody}\ , \\
 & & [\ \mathrm{feature\_end}\ ,\mathrm{NAME}\ ] \\
\mathit{FeatureChapterBody} & ::= & \{\ \mathit{Section}\ \} \\
\mathit{FamilymemberChapter} & ::= & \mathrm{familymember}\ ,\mathrm{NAME}\ ,\mathit{FeaturesList}\ , \\
 & & \mathrm{END}\ ,[\ \mathit{FeatureChapterBody}\ ]\ , \\
 & & [\ \mathrm{familymember\_end}\ ,\mathrm{NAME}\ ] \\
\mathit{FeaturesList} & ::= & \mathrm{features}\ ,[\ \mathrm{NAME}\ ,\{\ \text{,-tok}\ ,\mathrm{NAME}\ \}\ ]
\end{array}
$$

Feature chapters are chapters of Z sections. We will need to refer to the named Z sections inside of a (named) feature chapter. Pure family member chapters are named chapters that contain nothing but a features list. A family consists of feature chapters and of family member chapters.

Additionally, a family document usually comprises chapters of entirely informal accompanying text, like an introduction, the bibliography, and indices. When determining the semantics of the family, the latter kind of chapters is eliminated in the mark-up phase, as for Z documents.

A family member chapter optionally may contain a *FeatureChapterBody*, too, as a shorthand. The meaning of this is defined by a syntactic transformation that removes the *FeatureChapterBody*, inserts it in a new feature chapter immediately before the family member chapter, gives the feature chapter a fresh name otherwise not used, and adds this name to the *FeaturesList* of the family member chapter.

Informally, each family member chapter defines one specification in plain Z. We can extract one family member from a Z_F document by collecting all its features, and by concatenating all the sections of these features, resulting in a specification in plain Z.

In contrast, Z does not provide a defined way to extract one variant from a document. Furthermore, having incompatible variants in the same document is awkward. Z demands that it must be possible to order all sections in a linear way. Incompatible variants of the same item can be specified only by renaming them, as in *BirthdayBook*, $BirthdayBook_2$, $BirthdayBook_3$, ...

For our extension, we still demand that section names must be unique in the family document (as well as feature names and family member names must be unique in their respective name space), but other Z items such as schemas need to be unique only for the extracted specification in plain Z, not for the family document.

Handling Non-Monotonicity

Adding more sections to a Z specification, or adding more feature chapters to a Z_F family allows to add more constraints to the requirements. But this is not sufficient. 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. In particular, the behaviour of some component usually is not only extended by more possibilities, or alternatively restricted to less possibilities, but both these modifications happen at the same time. A feature typically is *non-monotonous.*

This demands that our formalism allows us to change the behaviour of a feature. (We consider a base system as just another feature.) Allowing such changes of course means allowing the possibility of feature interaction problems. Therefore, it is important that the changes happen in a controlled way. 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 a constant or by substituting an 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. Therefore, we choose that a feature may modify another feature at the level of Z sections.

Furthermore, we provide another language construct to beat feature interaction problems. 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. We therefore distinguish between the *essential behaviour* and the *changeable behaviour* of a feature. 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.

We add two more kinds of sections to the Z sections. The syntax is the same as for Z sections, except for the new keywords default_section and collect_section. (We also add an optional section_end construct to all these kinds of sections; we discuss its use further below.)

$$\begin{aligned} \mathit{SectionKeyword}_{ZF} \quad &::= \quad \mathit{SectionKeyword}_{Z} \mid \\ &\qquad \text{default\_section} \mid \text{collect\_section} \\ \mathit{Section}_{ZF} \quad &::= \quad \mathit{Section}_{Z} \text{ , [section\_end , NAME]} \end{aligned}$$

Default sections can be referenced by a remove clause. A remove clause is a new construct that may appear in the Z feature chapters.

$$\begin{aligned} \mathit{FeatureChapterBody}_{ZF} \quad &::= \quad \{ \; \mathit{Section}_{ZF} \mid \mathit{Remove} \; \} \\ \mathit{Remove} \quad &::= \quad \text{remove , [NAME , \{ ,-tok , NAME \}] , END} \end{aligned}$$

There is no difference in meaning between a default section and another kind of section. The difference is whether the remove clause produces a type error. A remove clause may legally name a default section only. If it names another kind of section, this is a type error.

The remove clause changes the way in which a plain Z specification is constructed for a family member. 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 identical to a normal section except that its list of parents sections adapts itself to losses of default sections due to remove clauses, as described in Section 3.3 below. A collect section should be used only to collect and combine properties defined in other sections, as discussed in Section 2.2 above. The composition operators used must be tolerant to missing definitions. We provide such a composition operator for CSP-OZ_F below. Similar operators can be defined for Z_F and for CSP_{ZF}, but this is still for further study.

3.2 Type Constraints

In Z, the sections of a specification may be presented in any order, but there must exist a permutation such that each section is well-typed. Similarly, in Z_F the chapters of a family document may be presented in any order, i.e., the feature chapters and the family member chapters, but there must exist a permutation such that each chapter is well-typed. This means that for such a permutation, each name in a features list must be the name of a feature chapter, and that each name in a remove clause must be the name of a default section.

There are restrictions on parents sections, and on remove clauses.

1. Neither normal nor default sections may have a collect section as a parent.

2. If the parent of a normal, i.e., essential, section is from the same feature or family member, then it must not be a default section.

3. The parents lists of collect sections must contain sections only that are either from the same feature or that are collect sections.

4. A remove clause may name default sections only.

5. For each family member m, we construct the set $\mathit{SectionNames}(m)$ 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 $\mathit{SectionNames}(m)$ only.

   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.

6. We also construct a reduced set $\mathit{RSectionNames}(m)$ from $\mathit{SectionNames}(m)$ by removing those default sections that are named in a remove clause of this family member.

   For any normal or default section in $\mathit{RSectionNames}(m)$, its set of parents sections must be a subset of $\mathit{RSectionNames}(m)$.

3.3 Semantics

The set *Features* contains all feature chapters' names. The set *Sections* contains all mappings from feature chapter names to sets of (any kind of) section names.

$$\mathit{Sections} \quad == \quad \mathit{Features} \twoheadrightarrow\!\!\!\!\!\!\mapsto \mathbb{P}\,\mathit{Name}$$

The meaning of a feature chapter can be determined by applying the function *Sections* to the feature chapter's name.

The meaning of a $\mathrm{Z_F}$ family is a mapping from family member names to meanings of plain Z specifications.

$$[\![\cdot]\!]^{\mathcal{F}} \quad : \quad \mathit{Family}_Z \rightarrow\!\!\!\!\!\!\!+ \; (\mathit{Name} \twoheadrightarrow\!\!\!\!\!\!\mapsto \mathbb{P}\,\mathit{Theory})$$

That is, each (pure) family member chapter is essentially a complete Z specification, except that all of its actual sections have been moved into feature chapters.

If a family member chapter is not pure, i.e., if it contains a feature chapter body, it is transformed syntactically into a feature chapter and a pure family member chapter, as described above. Such a feature chapter body might be convenient to specify the composition of all items from the different features in a suitable way.

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.

$$\mathit{Remove} \quad : \quad \mathit{Features} \twoheadrightarrow\!\!\!\!\!\!\mapsto \mathbb{P}\,\mathit{Name}$$

The features list of a family member chapter determines the features that are used to construct its Z 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.

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

The sections of a family member chapter's Z specification are all the sections $[\![fe_1 \ldots fe_n]\!]^{\mathcal{FL}}$.

The Z standard defines several semantic transformations. We add two such transformations. After the type check, we remove all section names in $(\mathit{Remove}\ fe_1 \cup \ldots \cup \mathit{Remove}\ fe_n)$ from the sets of parents sections of the collect sections. After that, all kinds of sections are transformed into normal sections. The meaning of the Z specification is then constructed in the usual way from the transformed sections $[\![fe_1 \ldots fe_n]\!]^{\mathcal{FL}}$.

3.4 Handling of Informal Text

The definition of the semantics of a family member chapter suggests the use of a tool that extracts a specification in plain Z for a specific family member. Such a tool does this by deleting all family members, features, and sections not selected. We propose that it does the same with the accompanying informal text that is associated with these family members, features, and sections.

We define the association of informal text to the formal text through the position in the document. The informal text between the first Z character of a family member and the last Z character of it is associated with this family member. Informal text between the last Z character of a family member and the first Z character of the next family member or feature is associated with the preceding family member. Features and sections are handled similarly.

The optional constructs "familymember_end", "feature_end", and "section_end" allow to delimit the association of informal text to the formal text. It is never extended beyond the last Z character of such a construct. While the start of a document is never associated to any formal text, these constructs allows to have the same for the end of a document, too. This can be used, e.g., to prevent a bibliography chapter near the end of the document to become part of the last family member chapter, and as a consequence to be removed when some other family member is selected.

The three ending constructs must each be followed by the name of the corresponding family member, feature, or section, as defined in the grammar above. This is supposed to aid the reader.

The specification in plain Z still should contain the family member chapter and feature chapter headings, in order to preserve the informal document structure. Even though they have a formal meaning in our extended $\mathrm{Z_F}$ language, they are informal text in plain Z.

4 Extending Z_F to CSP_{ZF}

Our extension of Z to Z_F is completely orthogonal to Fischer's extension of Z to CSP_Z. We modified the structure of a specification document at the level of sections and higher, while Fischer added added two new kinds of paragraphs and a semantic function for process expressions. Thus we can merge the extensions from Z to CSP_Z into Z_F right away. We call the resulting language CSP_{ZF}.

5 Extending CSP_{ZF} to CSP-OZ_F

Fischer's extension of CSP_Z to CSP-OZ again is completely orthogonal to our extension of Z to Z_F. We call the language resulting from a merge of the two extensions CSP-OZ_F^- .

On top of CSP-OZ_F^-, we add three more extensions that require the concepts of classes and inheritance introduced by the extension to CSP-OZ. The resulting language is CSP-OZ_F.

Finally, in Section 5.4 we recommend to not use one language construct of CSP-OZ_F when feature interaction problems are a concern. We don't restrict it formally, though, in order to have its full power available for other applications.

5.1 Default Properties Operator

The step from CSP_{ZF} to CSP-OZ_F^- brought (multiple) inheritance of classes, which is a new means of composing parts of a specification. We have already made the distinction between normal sections and default sections. Only the latter may be removed from a composition. Therefore, we also distinguish between the normal inheritance of CSP-OZ, and a special inheritance operator that is ignored in case the inherited class has been removed. We extend the syntax of CSP-OZ_F^- accordingly.

$$\begin{array}{lcl} \mathit{InheritedClasses}_{OF} & ::= & \mathit{InheritedClasses}_{O} \mid \\ & & \text{default\_properties}\ , \mathit{InheritedClassesName} \end{array}$$

The syntax is the same as for the "inherit" operator, except for the keyword.

We add two type constraints. The "default_properties" operator may appear in a collect section only. The class name in the operator must be a valid class name in the same feature, even though the referenced class may be defined in a default section that is removed in some family members.

The semantics of the "default_properties" operator are exactly the same as the semantics of the "inherit" operator, except that there is an additional semantic transformation. If the class name is not a valid class name for a family member in the sections of set *RSectionName* defined above, then the default properties operator is eliminated when determining the semantics of the family member. Otherwise, the operator is replaced by an "inherit" operator.

5.2 Shorthand: Classes as Sections

Classes are larger units than mere Z 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.

$$\begin{array}{rcl} \mathit{Section}_{OF} & ::= & \mathit{Section}_{ZF} \mid \\ & & [\ \mathit{SectionKeyword}_{ZF}\ ,\ \mathrm{NAME}\ ]\ ,\ \mathit{Class}\ , \\ & & \{\ \mathit{Paragraph}_O\ \}\ ,\ [\ \mathrm{section\_end}\ ,\ \mathrm{NAME}\ ] \\ \mathit{Class} & ::= & \mathit{Class}_C \mid \mathit{Class}_O \end{array}$$

The name of this implicit section is the name of the class. (Please note that family members, features, sections, and classes each have separate name spaces.) The parents sections names of the section are the names of the classes inherited by this class. Any following class will be part of the section. You may use "section_end" to prevent this.

The class is transformed into a collect section instead of a section exactly if it contains a "default_properties" operator. The class is transformed into a default section exactly if there exists a "default_properties" operator somewhere in the same feature that references this name.

Note: default sections are always identified correctly if collecting classes are used, as recommended in Section 2.2. Collecting classes are identified correctly only if they reference at least one default class. In practice, this should always be the case, since it is necessary to ensure extensibility.

The above grammar rules also allow an "intermediate" abbreviation where the class is preceded by a (normal, default, or collect) section keyword and the class's name, but not by a parents list, which is otherwise mandatory. This is useful when we would like to have some section heading in larger print, but when we don't want to repeat the same list of names as the list of parents and as the list of inherited classes. It also allows to make default or collect sections explicit in the case where the recommendations of Section 2.2 are not followed. The missing parents list is constructed from the names of all inherited classes that appear somewhere in the section, and that are not defined in this section.

5.3 Shorthand: Pure Family Member Chapter Filled with Collecting Classes

In Section 2.2, we showed how we use CSP-OZ to specify requirements. We collect all requirements of the system and of the environment into two classes, using inheritance, and then we compose these two classes by a parallel composition.

In order to make this style of requirements specification more convenient, we define the meaning of a pure family member chapter (i.e., one without a feature chapter body) in CSP-OZ$_{\mathrm{F}}$ differently from $\mathrm{Z_F}$ and $\mathrm{CSP_{ZF}}$. By a syntactic transformation, we add a feature chapter body with one collect section containing three classes, as described below, and then we interpret this chapter in the usual way as an additional feature chapter and a pure family member chapter (now without further expansion). The parents sections of the collect section are determined by the mechanism for implicit parents described above. The first of the classes has the same name as the family member chapter, but with the three letters “Sys” appended. The second has the same name, but with the three letters “Env” appended instead. The third has the name of the family member chapter with nothing appended. The first class contains “inherit” operator(s). The names of the inherited classes are the names of all the features from the features list of the family member chapter, with “Sys” appended to each of them. The second class contains the same, but with the text “Env” instead. The third class contains an “inherit-interface” operator with the name of the second class[3], and it contains a CSP process *main* that consists of a synchronized parallel composition of the first two classes.

Note that is the specifier's duty to actually provide classes with all the names generated by appending text. If they do not exist, this is a type error, as usual. The entire generation step may be regarded as a convenience for the specifier. It avoids listing all composed features in three places, when we use our specific specification style. If someone uses CSP-OZ$_{\mathrm{F}}$ with a different specification style, he should simply avoid pure family member chapters.

5.4 Using the Inheritance Operators

Our approach to detecting feature interaction problems assumes that non-monotonicity is handled exclusively by remove clauses, and that inheritance essentially is logical conjunction. This assumption is wrong in the general case, since CSP-OZ (and also CSP-OZ$_{\mathrm{F}}$) allows the renaming of items when applying the inheritance operator. Items that can be renamed are channels, local definitions, operations, CSP processes, and even variable names defined in schemas, channels, and operations. Renaming effectively removes the old definition of this item, at least under the old name. It therefore makes room for any new, maybe completely different, definition. Therefore, the powerful optional renaming part of the inheritance operators should not be used, if feature interaction problems are a concern. This might be checked by a type checker and flagged as a warning.

6 The **genFamMem** Tool

6.1 Overview

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

- extracts specifications in plain CSP-OZ from a family document,

[3]The inherit-interface operator exists for CSP-OZ only, and not for Object-Z. Actually, its very idea came from our application presented here, and was accepted by Fischer for CSP-OZ.

- auto-fills the final collecting classes,
- 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,
  - 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 parser for our extension of CSP-OZ which is written using the lex/yacc parser generator [12], (or, rather, their flex/bison variants), and C code. GenFamMem is a re-development of a previous, script-based prototype. The parser takes a family document and the name of a family member as its input for a first pass, and it calculates which sections should be included into the generated specification. After parsing the structure of the family, the parser 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. After that, we type-check the resulting requirements document in plain CSP-OZ using von Garrel's CSP-OZ type checker cspozTC [13].

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 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 [14, 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. 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.

The tool's parser 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

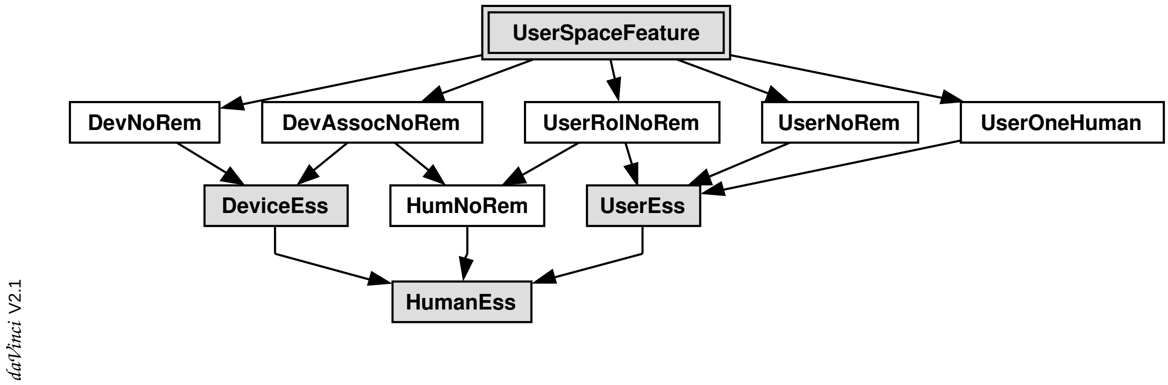

Figure 3: The uses-hierarchy of the sections of feature UserSpace from the case study.

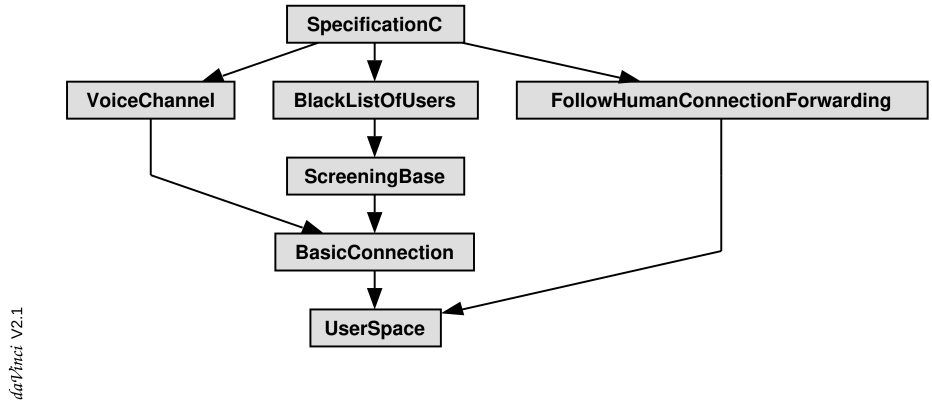

Figure 4: The uses-hierarchy of family member SpecificationC from the case study, abstracted to features.

them as sequences of characters. The standard defines two mark-ups, LaTeX markup and email mark-up. Our tool currently supports LaTeX mark-up only. But since the mark-up phase is contained in a separated program module, other mark-ups can be added easily.

Due to the modular structure of our family extension, every Z_F specification is also a CSP_{ZF} specification, and every CSP_{ZF} specification is also a CSP-OZ_F specification (except for the shorthand of pure family member chapters). Therefore, the generator can be applied to CSP_{ZF} and Z_F family documents as well, then generating CSP_Z and Z specifications, respectively. 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 [16].

6.2 Invocation

The tool genFamMem can be invoked as follows:

```
Usage:
genFamMem <options> familymember infile
genFamMem <options> --uhier-all infile
genFamMem <options> --uhier-familymember familymember infile
genFamMem <options> --uhier-feature feature infile
Options are:
--uhier-all, -l:
print uses-hierarchy of all sections, only
--uhier-familymember familymember, -a familymember:
same, for one family member
--uhier-feature, -e:
same, for one feature
--ascii, -x:
when --uhier-*, use ASCII output format
--daVinci, -X:
when --uhier-*, use daVinci output format (default)
--only-features, -f:
when --uhier-*, abstract sections to features
--language-z, -z:
input language is Z
--language-cspz, -c:
input language is CSP_Z
--language-cspoz, -o:
input language is CSP-OZ (default)
--help, -?, -h:
print this help info
--version, -V:
print version information
--debug, -d:
print additional debug information
--bison-debug, -D:
let the bison parser print its debug information
--markup, -M:
do mark-up only and print it (useful for debugging)
Output appears on stdout.
An option argument is specified as in "--uhier-feature=foo",
in "--uhier-feature foo", or in "-efoo".
```

7 Summary

In a recent paper[1], we introduced the idea of families of formal requirements. In this paper, we define precisely the language constructs which we use for describing features and for describing family members constructed from features. We have chosen the formal specification language CSP-OZ [2, 3, 4] as a base for adding our feature constructs.

The main part of this manual presents the definition of the syntax and the semantics of our language extension. Interestingly, most of our extension can be applied already to the language Z [9, 10, 11], which is not only a starting point for the definition of CSP-OZ, but which is a formal specification language with a rather large user community itself. While we follow the step-wise definition of CSP-OZ, we lift our extension from Z to $\mathrm{CSP_Z}$. $\mathrm{CSP_Z}$ is a variant of CSP (Communicating Sequential Processes) [5, 6], which again is a formal specification language with quite some tradition and user community. Finally, we lift our extension to CSP-OZ and add a few more constructs that require the object-based concepts of inheritance and classes.

We use the language CSP-OZ – in its extended version – for our case studies [17], since it provides all the expressive power and convenience we need there. Our extension itself is of a more general nature, and our work also contributes to the communities of Z and CSP users, since they can now use the idea of families, too. The use of the extended languages even is not restricted to specifying requirements.

Even though the tool support for CSP-OZ is still restricted to a type setting style and a type checker, our extension proper needs little tool support, i.e., a generator program, which already exists. This generator can be applied to the extended versions of Z and CSP, too, for which much broader tool support exists.

The goal of our extension is to avoid feature interaction problems in requirements documents, or at least to help detect them, by

- allowing to maintain all variants of a requirements document together, 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 class inheritance and of parents sections sets, 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.

Therefore, our extension converts at least some of the feature interaction problems into type errors.

References

[1] Jan Bredereke. Families of formal requirements in telephone switching. In Muffy Calder and Evan Magill, editors, “Feature Interactions in Telecommunications and Software Systems VI”, pages 257–273, Amsterdam (May 2000). IOS Press.

[2] 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).

[3] 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).

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

[5] C. A. R. Hoare. “Communicating Sequential Processes”. Prentice-Hall (1985).

[6] A. W. Roscoe. “The Theory and Practice of Concurrency”. Prentice-Hall (1997).

[7] Graeme Smith. “The Object-Z Specification Language”. Kluwer Academic Publishers (2000).

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

[9] ISO Panel JTC1/SC22/WG19, Final Committee Draft, CD 13568.2. “Z Notation” (August 1999).

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

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

[12] Levine, Mason, and Brown. “Lex & Yacc”. O’Reilly & Associates Inc (1992).

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

[14] Michael Fröhlich. "Inkrementelles Graphlayout im Visualisierungssystem daVinci". PhD thesis, University of Bremen, BISS Monographs No. 6, Shaker Verlag, Aachen, Germany (August 1998).

[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. "genFamMem 2.0 Home Page". University of Bremen (2000). URL http://www.tzi.de/~brederek/genFamMem/.

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