Interner Bericht FB 10 Abteilung Semantik

Modular, Changeable Requirements for Telephone Switching in CSP-OZ†

IBS-99-1

Jan Bredereke

University of Oldenburg, FB 10
P.O. box 2503
D-26111 Oldenburg
Germany

E-Mail: jan.bredereke@topmail.de
WWW: http://www.poboxes.com/jan.bredereke

Date: October 11, 1999
Document Id: IBS-99-1
Revision: 1.0

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

Abstract

Requirements documents for software need not only be written, but they also need to be maintained afterwards. In telephone switching, there arise particular problems due to the strong mutual dependences of telephone features and due to the current rapid change in this area. We attempt to avoid or at least reduce feature interaction problems during the extension or change of a requirements document through a suitable requirements document structure. We perceive all variants and revisions as a single *requirements family*, documented together. Our approach to requirements specification grew out of the Functional Documentation approach, also known as "Parnas tables". We now apply and extend this approach using the formal description technique CSP-OZ, taking advantage of its built-in support for inheritance and parallel composition. We structure the requirements in a modular way suitable to our application area; and we present a way to compose the partial specifications. A preliminary case study demonstrates our approach and shows that CSP-OZ can indeed be used for it. More work is required. Besides an extension of the case study, several aspects of the incremental specification formalism still need to be worked out.

Contents

Chapter 1

Introduction

Telephone switching systems must undergo substantial changes currently. Reasons comprise the opening of the telecommunications markets in the world, cheaper broadband transmission hardware, faster switch processing hardware, the advent of wireless mobility, and the probable advent of mass internet telephony. The providers need to incorporate an ever increasing number of new services and features into the switches, in shorter and shorter intervals, in order to remain competitive [CCG+98]. Such switches already comprise hundreds of features [BDC+89], and they rank among the largest software systems in the world. In recent years, so-called *feature interaction problems* have become a serious obstacle to adding more features to these systems [IEE92, BoVe94, ChOh95, DBL97, KiBo98].

A simple example of a feature interaction is the introduction and joint subscription to Originating Call Screening (OCS) and Abbreviated Dialling (ABD). The OCS feature allows to enter directory numbers into a list, e.g., by a parent. The feature aborts a call if a number is dialled, e.g., by an adolescent, which is found on this list. The ABD feature translates a list of short, i.e. one-digit, numbers into full directory numbers. If both features are executed in the wrong order, i.e., OCS before ABD, the OCS feature can be circumvented by the adolescent. He registers the desired number as an abbreviated number, and then dials the abbreviated number, reaching a destination which he should not reach. A resolution to this problem is to execute the OCS feature only after all number translating features have been executed.

One counter-measure against feature interaction problems was to devise the Intelligent Network (IN) standard [DuVi92, ITU97b], which introduces an open interface into existing systems. So-called Detection Points in the finite state automata of the Basic Call State Model allow extended processing at predefined events, thus allowing to specify and implement new features separately from the base system. Nevertheless, it turned out that feature interaction problems still occur in the IN. One of the reasons is that the extended processing may have global effects: it can modify arbitrary data, and it can terminate by jumping back to an arbitrary Detection Point.

Maintenance of software is often considered as the last step in a Software Engineering process. Nevertheless, this task usually constitutes the largest part of work and costs during the lifetime of a successful software product. A major difficulty in maintenance of large systems is to identify all parts that need to be changed, without the need to gain an understanding of all details of the entire system. In this report, we are concerned with the maintenance of software requirements for telephone switching systems.

In previous work [Bre99c, Bre98b] (see also [Bre99a, Bre99b, Bre98a]), we have investigated how we should structure software requirements with regard to future modifications using the Functional Documentation approach [PaMa95, vSPM93, vS92] to requirements specification, also known as "Parnas tables". We presented a way to compose partial requirements specifications written in Functional Documentation.

While this previous work presents an earlier version of our method, we were only able to sketch the details of the composition operators and their semantics. In this report, we take the formalism CSP-OZ [Fis97b, Fis97a] and use it for our method. CSP-OZ is a formal specification language suited for both specifying communication system behaviour and data, since it is a combination of CSP (Communicating Sequential Processes) [Hoa85, Ros97] and Object-Z [DRS95, Smi92] (which in turn is an extension to Z [WoDa96, Spi95]). It can be used to specify pure software requirements, without introducing design decisions already.

We chose CSP-OZ for several reasons. The semantic model of CSP is based on synchronous communication events between processes; this has lead to a well worked-out refinement calculus for CSP. The language SDL (Specification and Description Language) [FæOl94, ITU94], which is often used in telecommunications, is based on asynchronous communication and thus cannot offer a comparable refinement calculus. Similary, there is a well established notion of (operation and) data refinement in Z [WoDa96, Spi95] and its object-based derivative Object-Z. Accordingly, CSP-OZ already provides a worked-out formalism for (multiple) inheritance, both for the CSP behavioural part and for the Object-Z data part. In the CSP-OZ semantics, both inheritance and parallel composition are essentially a kind of logical conjunction, and are integrated seamlessly. Furthermore, there is tool support available for CSP, in particular there is the model checker FDR (Failures-Divergence Refinement) [Ros94]. There will be a translation tool for CSP-OZ to FDR soon [FiWe99, Wir99], and there exists alread a type checking tool [vG99]. CSP-OZ has been developed recently at the University of Oldenburg.

In the appendix of this report, we present a case study in CSP-OZ that is the translation of a part of the case study of [Bre98b], which we specified in Functional Documentation. Using the formalism of CSP-OZ as a sound base, we now work out further details of the different composition operators that we need.

Unfortunately, unexpected time restrictions prevented us from translating the entire case study, and from treating the composition operators at the depth we would have liked. For example, the case study comprises only two very basic features and none of the optional features that go beyond the Plain Old Telephone Service (POTS). Nevertheless, the principle should become clear anyway, and the specification is self-contained and can be used for further experiments, for example for verification experiments. This already shows an advantage of our approach: the requirements specification is quite scalable due to its modularisation. We can not only add more features, we can also remove any number of features.

In Chapter 2, we sketch the application domain. In Chapter 3, we provide some details about CSP-OZ that are relevant for this report, and in Chapter 4 we present how we specify requirements in general using CSP-OZ. Chapter 5 is the core part where we discuss how in particular we apply our method for modular requirements specification using the composition operators of CSP-OZ. Finally, Chapter 6 sketches related work, and Chapter 7 summarizes our results and indicates further research issues.

Chapter 2

Telephone Switching Requirements

We now introduce to the domain of telephone switching requirements, but only as far as necessary for an impression of the complexity and of the specific problems involved. In Section 2.1, we report on one imporant way of specifying telephone switching services, and give some evidence for the complexity involved. In Section 2.2, we specifically introduce to what is known as feature interaction problems. Again, we cannot cover the entire subject, but we motivate the relavance of our approach.

2.1 The Intelligent Network Standard

In order to open up proprietary telephone switching systems, and in order to allow to enhance them by value-added services quickly, possibly by third-party providers, the ITU-T has conceived a set of standards for the *Intelligent Network* (IN) in 1993 [DuVi92, ITU97b]. Since then, these standards have been evolved further by two *Capability Sets* [ITU93, ITU97a], which fill the general framework with specific descriptions of services and features. A third Capability Set is currently under design. An important point in the IN system architecture is the separation between basic call processing and the processing of value-added services.

For the purposes of this standard, basic call processing is described by a *Basic Call State Model* (BCSM), which consists of two finite state automata; one for the originating side of a call and one for the terminating side. Figures 2.1 and 2.2 show these automata in the Capability Set 2 version. We will not discuss their actual behaviour here, but we will highlight some of there structure, as far as it is relevant for this report. The large boxes are *Points in Control* (PICs), i.e., states of the automata. The small square boxes are the *Detection Points* (DPs), at which the execution of the basic call processing may be suspended and control may be transferred to some extended processing. There are elaborate mechanisms to arm and to trigger these DPs. When the extended processing has completed, control will return to the PIC following the same DP or to a specified different PIC.

One specifies the extended processing by a combination of *Service-Independent Building Blocks* (SIBs). Each of them has an entry point and one or more exit points, and one composes them by *Global Service Logic*. Global Service Logic can be represented as arrows between SIB exit and entry points. This may include the special *Basic Call Process* SIB, whose exit and entry points are called Points of Invocation and Points of Return, respectively. The Basic Call Process is mapped onto the Basic Call State Model presented above, and the Points of Invocation and Points of Return are mapped onto the Detection Points. The standard

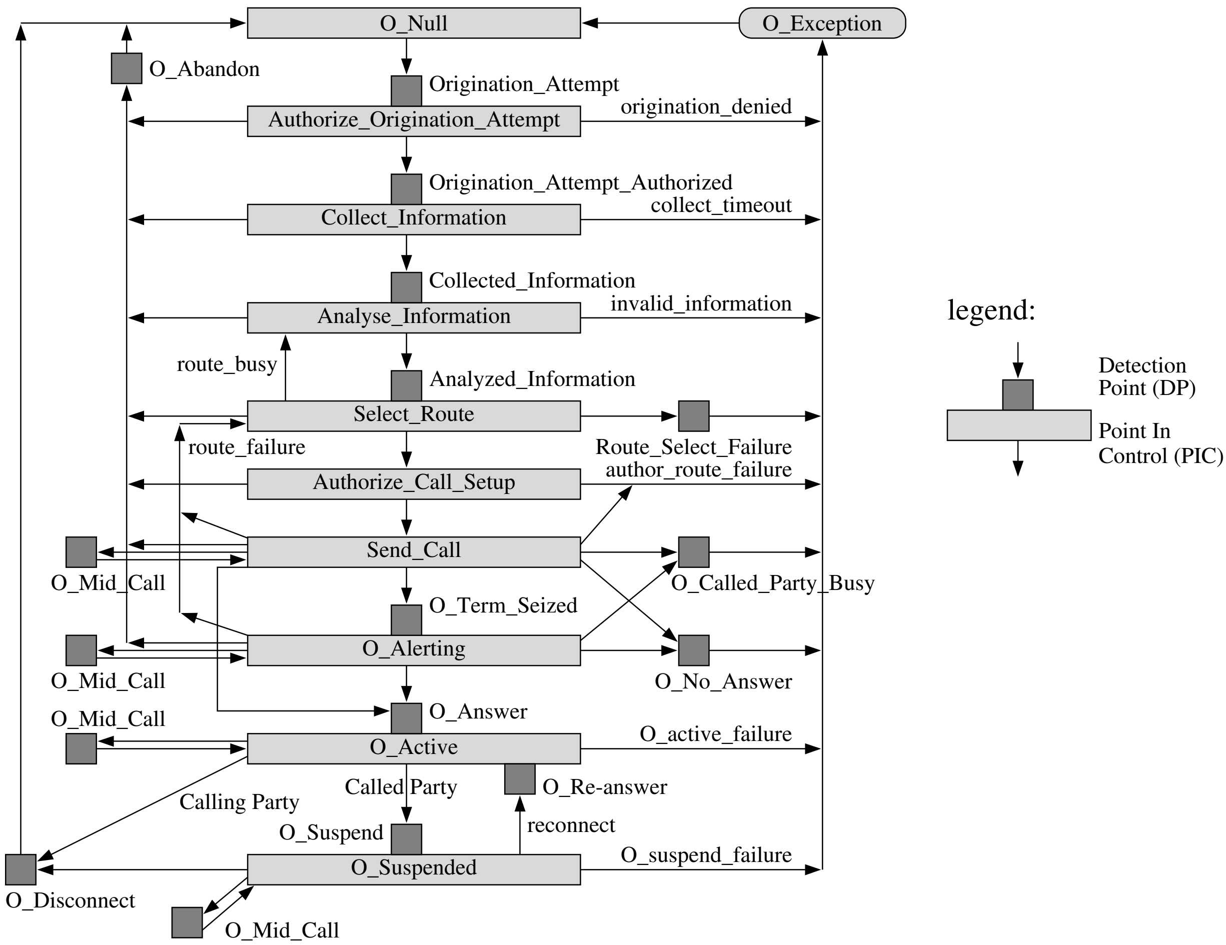

Figure 2.1: The Basic Call State Model of ITU-T standard Q.1224, originating side automaton.

specifies the behaviour of SIBs by SDL [ITU94] diagrams. An example for a SIB is the *Charge* SIB, which determines the special charging characteristics (e.g., special rate, reverse charge, split charging) that apply to calls related to IN-provided service features. Another example is the *Compare* SIB, which provides the capability to compare an identifier against a specified reference value.

Different SIBs provide functionality of quite different complexity, as the Charge and the Compare SIBs already show. Some of the SIBs defined in the standards can even be called vague, as the *Algorithm* SIB. It applies a not yet specified mathematical algorithm to input data and produces an output data result. The BCSM automata shown in Figures 2.1 and 2.2 provide 29 DPs, which are mapped onto 15 Points of Invocation, to which SIBs may be attached. When we add several features, some of the Points of Invocation need to be used for more than one feature, and undesired feature interactions become likely. The Intelligent Network Capability Set 2 already standardizes 63 switching features, and additionally supports 16 management features and 38 service creation features.

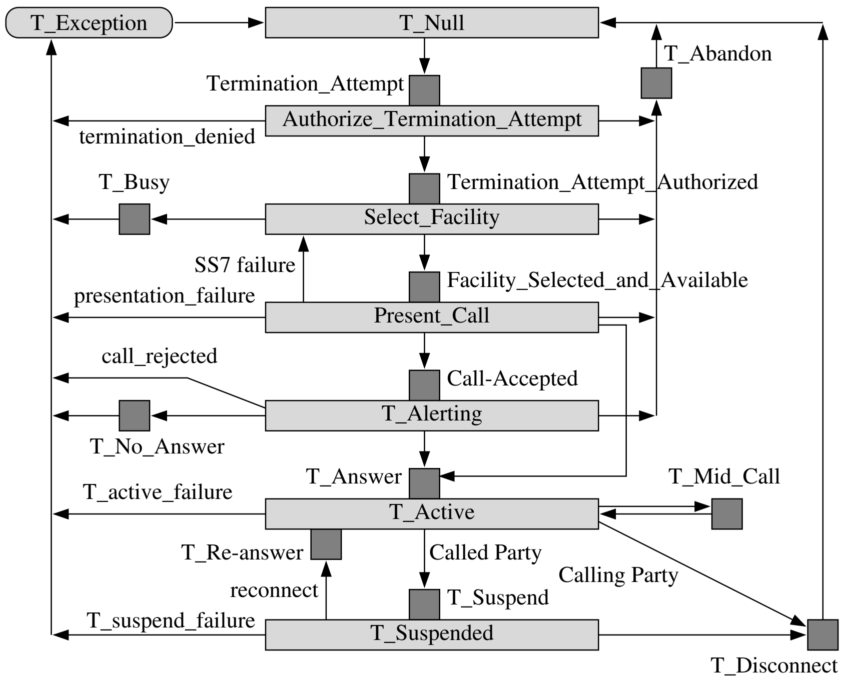

Figure 2.2: The Basic Call State Model of ITU-T standard Q.1224, terminating side automaton.

2.2 Feature Interaction Problems

Digital telephone switching systems rank among the largest software systems in the world, and due to their success they have a long history of modifications and enhancements. Hundreds of features have been developed already [BDC+89]. Therefore, the probability is high that augmenting a switching system by one more feature will influence another feature, potentially in an undesired way. This phenomenon is called a *feature interaction*. Due to the number of possible combinations of features, even experienced developers can detect and resolve feature interactions reliably no more.

Even though many people use the term "feature interaction *problem*", we prefer the term in the plural, "problem*s*". A common definition is: "a feature interaction occurs when the behaviour of one feature is altered by the use of another" (in particular in an undesired way). (Compare, e.g., [CaVe93, KiVe95, AhGr95, MTJ93].) When we look at this definition, a "feature" can be, more or less, about every change to the system imaginable. This can range from the introduction of Call Completion to Busy Subscriber over Number Portability to just increasing the number of trunks that the switch can handle. Accordingly, several different classes of causes for feature interactions have been identified. Different feature interaction categorizations have been proposed, e.g., in [CGL+94, ChOh95, CaCo96]. Among others, Cameron et al. [CGL+94] present a categorization by *causes* of interactions, and their top-level categories are: violation of feature assumptions, limitations on network support, and intrinsic problems in distributed systems. In our view, the feature interaction problems are a collection of separate problems from different domains. One of the domains is the *structuring of the software requirements with respect to maintenance*. It is the topic of this report.

We present one example for a feature interaction. It can occur between a Credit Card Call

(CCC) feature and a Voice Mail feature. The example is taken from the feature interaction benchmark by Cameron et. al. [CGL+94]. Credit Card Call includes an authorization phase where the card's number and PIN must be entered. For convenience, this feature often allows placing another call without re-entering, by just pressing the "#" button at the end of the first call. On the other hand, to access voice mail messages from phones other than her own, a subscriber of some Voice Mail service can call her own phone number (maybe using the CCC feature) and then press the "#" button followed by some identification to indicate that she is the subscriber. These features cannot work together properly since the telephone system has no way to determine whether the caller presses the "#" button during a voice mail announcement because she wants to terminate the call without leaving as message, or whether she is the subscriber of the voice mail box and wants to authenticate herself.[1]

The feature interaction benchmark of Cameron et. al. comprises 22 representative examples of feature interactions, which are taken from practical experience in developing features, and which are intended to cover a wide variety of causes and symptoms, such that proposed detection and resolution approaches for feature interactions can be evaluated by using them. Not yet included are mobile and Internet phone services, which present additional challenges.

[1]This feature interaction is resolved in the calling card feature to which the author once subscribed: the "#" button must be pressed at least 2 seconds to take effect. Thus, we have two distinguishable signals.

Chapter 3

CSP-OZ

We do not introduce to CSP-OZ in this report and assume a working knowledge of it. Introductions to CSP-OZ are for example [Fis97b, Fis97a]. In this chapter, we introduce only to the particular composition constructs of CSP-OZ that we use, as a base for our particular selection decisions on composition constructs. Furthermore, we present an overview of the tool support available, in particular the tools that we used.

3.1 Built-In Syntactic Composition Means

CSP-OZ offers several syntactic means of composition. We now sketch those that are interesting with respect to the composition of features in our approach. For more details on CSP-OZ see [Fis97b, Fis97a].

Parallel composition of CSP processes. We can choose among different flavours of synchronization between two processes: *interleaved* processes ($P \mid\mid\mid Q$) can take part in their respective events independently; if both processes can take part, only one will do it. *Parallel* processes (P $\|$ Q) must both take part in any event that happens. It is also possible to synchronize the processes on specific events and to interleave any other events ($P \underset{A}{\|} Q$ and $P\, {}_A\|_B\, Q$).

Hiding for a CSP process. ($P \setminus A$) This removes the specified events from the alphabet of the process, and replaces these events in the behaviour by the internal event.

Logical conjunction of Object-Z operations. This is a special case of Z schema conjunction. The predicate parts of the operations are combined by a logical "and" ($S \wedge T$). The declaration part is not used.

Inheritance of Object-Z classes. All declarations of the inherited classes are merged into those of the inheriting class. Common identifiers must have identical types. Predicates on the state are combined by logical "and" ($S \wedge T$). The predicate parts of common operations are combined by conjunction, too; furthermore their parameters must be the same, or one operation must only extend the list of parameters of the other. Multiple inheritance is available.

CSP-OZ class with both a CSP and an Object-Z part. Both parts have a semantics in terms of events. They are combined by parallel composition (P $\|$ Q).

Inheritance of CSP-OZ classes. The Object-Z parts of the classes are combined as with inheritance of Object-Z classes, and the CSP parts are combined by parallel composition (P $\parallel$ Q). The definitions of the possible set of events, i.e., the channel definitions, are merged. Multiple inheritance is available.

It is possible to replace specific operations of the inherited class in the inheriting class. We do not use this language feature since it contradicts our constraint-oriented specification style and would subvert our analysis of change consequences (see below).

Inheritance of the channels of CSP-OZ classes. This does use neither the Object-Z parts nor the CSP parts. Only the definitions of the possible set of events, i.e., the channel definitions, are merged, as within normal inheritance. This may be handy to provide the necessary channel definitions for a following parallel composition of CSP-OZ classes.

All operators that are available in CSP for processes may be applied to CSP-OZ classes, too. Of particular interest are the parallel composition operators and the hiding operator:

Parallel composition of CSP-OZ classes. These parallel composition operators work like CSP parallel composition, i.e., we have the respective synchronization or interleaving of events as above, but the behaviour of the components that are composed is specified in CSP-OZ instead of CSP. Only events may be shared. If a class comprises predicates on the state space, these restrictions apply to the state space of the respective class only.

Hiding for a CSP-OZ class. This removes the specified events from the interface of the CSP-OZ class, and it replaces these events in the behaviour by the internal event.

One interesting result from the development of CSP-OZ is that inheriting two CSP-OZ classes is equivalent to their parallel composition (P $\parallel$ Q), provided the variables are disjoint [Fis97a].

The main means of composition that we use is (multiple) inheritance of CSP-OZ classes. It joins the alphabets of the inherited classes suitably, and the logical conjunction (or parallel composition) of the inherited restrictions matches our constraint-oriented specification style.

3.2 Tool Support

There is some tool support available for CSP-OZ:

- a LaTeX typesetting style
- the type checker cspozTC
- the syntax-directed, graphical editor Moby/CSP-OZ

We used the LaTeX style `csp-oz.sty` to typeset this document and the specification in its appendix; and we used the type checker cspozTC [vG99] to ensure the syntactic and type correctness of the specification in the appendix.[1] Since we prefer textual editing over

[1] The class ${}^{P}\mathit{SimpleSpecification}$ on page 51 could not be checked since the language construct `inheritchannels` was devised after the type checker was completed. When this class is left out, no errors are found.

graphical editing personally, we did not try out the editor Moby/CSP-OZ. More tool support will be available until the end of 1999:

- the model checker FDR with a translation from CSP-OZ to the language of FDR

This translation [FiWe99, Wir99] will enable us to validate arbitrary safety properties of CSP-OZ specifications, through the commercial model checking tool FDR [Ros94] which is available for CSP. The validation will be restricted, of course, by the size of the complete transition graph which FDR constructs. Also, a few constructs of Object-Z might be difficult to translate into executable code automatically.

Chapter 4

Specifying Requirements Using CSP-OZ

In this chapter, we discuss some general aspects of using CSP-OZ for specifying requirements, which are not directly related to incremental and modular specification.

4.1 Specifying Requirements Only

A requirements document shall describe the behaviour of the system without prescribing any design decisions about how this behaviour can be achieved. In particular, no internal structure shall be prescribed. Therefore, the entire description must be in terms of what is observable by the environment of the system to construct.

In the Functional Documentation approach, we observe variables in the environment, and we then can construct events out of value changes of variables. This leaves the possibility to specify continuous value changes too, through piecewise continuous functions. Such continuous behaviour does not appear in our application domain. Therefore, considering events is sufficient. This is the approach of CSP-OZ; the semantics of a CSP-OZ specification are the events that can or cannot be observed.

We view software development as a contract between the developer and the customer. The requirements specification prescribes how the relevant part of the world should behave when the system has been installed. Since the developer can influence some parts of the world only, the requirements document must also state who is responsible to provide which part of the behaviour. In particular it must state on which behaviour of the environment the system developer can rely.[1] Some events are under the control of the system only, and some events are under the control of the environment only. We call the former controlled events, and the latter monitored events.[2]

The semantics of a CSP-OZ specification consists in statements about which of the globally visible events may happen when. Variables have a semantics only insofar as they may restrict when operations and thus events can occur, and classes serve only as a means to structure the document and to restrict the scope of identifiers. CSP-OZ does not prescribe any particular

[1]For example, our simple specification in the appendix makes the assumption that telephone devices are neither added nor removed during the lifetime of the system. (This could be changed later through a suitable additional feature.)

[2]Events under the control of both are rare in requirements documents, and they do not occur in our work.

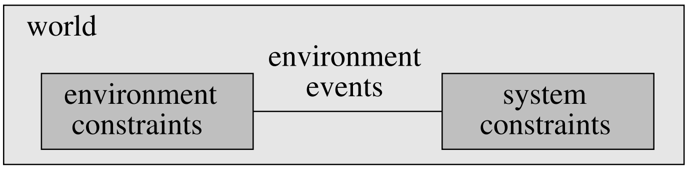

Figure 4.1: Composition of the world from system and environment.

$^{P}SimpleSpecification$

inheritchannels $^{P}SimpleSpecificationEnv$

$main \;=\; {}^{P}SimpleSpecificationEnv \parallel {}^{P}SimpleSpecificationSys$

Figure 4.2: Specification of the composition of the world.

implementation structure. In order to specify the above distinction between the system and the environment, we add a *special interpretation* of the topmost class structure to the semantics of CSP-OZ: it shall be a separation of the obligations of the developer and the customer. We specify the entire relevant part of the world as a single CSP-OZ class. This class contains nothing but the definition of the possible events and the parallel composition of one class for the environment and one class for the system (Figure 4.1). Figure 4.2 shows an example from the appendix.

The environment shares all the events in the alphabet of the system; we ensure this by taking the events of the environment as the events of the world through the `inheritchannels` construct.[3] This prevents us from specifying additional system events (that result from design decisions).

4.2 Distinguishing Monitored and Controlled Events

It is necessary to distinguish between events under the control of the system and events under the control of the environment in requirements documents. For this, Object-Z does not provide suffient means, and we need the new language constructs that CSP-OZ introduces.

In Object-Z, operations may or may not block. But this information is buried in the relation between pre- and post-states of that operation, and it is not explicit. CSP-OZ allows to specify separately the enabling condition and the effects of an operation *op* through the `enable_`*op* and `effect_`*op* constructs. Thus we can ensure syntactically that monitored events cannot be rejected by the system by specifying an `enable_`*op* operation for the system with the condition *true*. This assumes that this operation is not mentioned in the CSP part of the system.

If we specify behaviour in CSP, either for the system or for the environment, we have to use the usual CSP operators. In general, they can make it a little more difficult to see whether there are restrictions on a certain event or not. In our case study, this information is plain to see. But for more advanced specifications, it would be interesting to investigate further stylistic or syntactic rules.

[3]Actually, this construct was added to CSP-OZ on our request.

In any case, it would be useful to check the consistency of the CSP-OZ specification. For this reason, we explicitly specify whether an event is monitored or controlled by the system by prefixing all monitored events by "$^m\_$", and by prefixing all controlled events by "$^c\_$". Besides enhanced clarity for the reader, this also enables a consistency check through automated tools. For example, model checking can reveal whether the system process can always take part in monitored events, and whether the environment process can always take part in controlled events. This can be done by the tool FDR for the CSP part (provided the state space is not too big). For the Object-Z part, a simple enhancement of the type checker cspozTC could check for the proper uses of the $\mathtt{enable}\_op$ and $\mathtt{effect}\_op$ constructs.

4.3 Proper Event Control and Concurrency

Functional Documentation and CSP-OZ are different in that the former is a *synchronous language* and the latter is not. This means that the change of a monitored variable in Functional Documentation may demand the immediate change of the value of a controlled variable.

To be more precise, the "ideal behaviour" may demand an immediate change. In Functional Documentation, a requirements specification consists of three separate parts:

1. the monitoring precision
2. the ideal behaviour
3. the control tolerance

The first part describes after which time the system may notice a value change of physical monitored variables, and after which time it eventually must notice.[4] The third part describes how fast the system must physically implement the changes of the "ideal" controlled variables.[5] The second part, the ideal behaviour, comprises the complexities of the behaviour relation between monitored and controlled variables. (It may state real-time constraints, too.) A requirements specification consisting of the ideal behaviour only would be impossible to implement, since all physical effects take time, and no system can react immediately. But a requirements specification consisting of the above three parts can be implemented, provided the precision and tolerance constraints are reasonable.

In CSP-OZ we cannot specify real-time constraints. And for systems where timing is not crucial, leaving out the timing details can be a good abstraction. Nevertheless, we still want to specify the ordering of events. The process $P = a \rightarrow b \rightarrow \mathtt{Skip}$ states that event a must happen first, and that event b then must happen at an unspecified time afterwards. But variable values may change at the exactly same time, too (in particular when this has a common cause). There are two ways to model such changes in CSP-OZ. First, we can specify one event that denotes two value changes at the same time. In some cases, this is not appropriate, in particular when we extend a specification incrementally by a new variable, and when we don't want to modify the change events for an existing variable just because the new variable sometimes should change its value at the same time. The second way is to model concurrent changes by an arbitrary interleaving of two or more events. The justification is that physical events can be observed with finite precision only, such that events happening

[4]For analogue values, the first part also states the value measurement precision.

[5]For analogue values, it also states the admissible value tolerance.

at about the same time cannot be distinguished from events at exactly the same time. This is the *interleaving hypothesis.*

When we use CSP-OZ, we usually describe the monitoring precision mentioned above outside of CSP-OZ. A CSP-OZ event then is not a physical event itself but the event that the system notices that a physical event has happened. Similarly, CSP-OZ specifications often leave out the control tolerance part, too; their controlled events are no physical events but events internal to the system that will cause the real physical events after a short delay. In particular, this approach allows to add a synchronicity hypothesis: after one monitored event, no further monitored event will happen before all resulting controlled events have happened. this allows us to specify a system that monitors an event ${}^{m}a$ and always reacts with two controlled events ${}^{c}b$ and ${}^{c}c$ in the following way: $S = {}^{m}a \rightarrow {}^{c}b \rightarrow {}^{c}c \rightarrow S$. Since ${}^{m}a$ is a monitored event, the system must always be prepared to take part in it. Due to the synchronicity hypothesis, this is the case even though the CSP-OZ process S is not, after taking part in ${}^{m}a$ and before ${}^{c}c$ has occured.

Unfortunately, this synchronicity hypothesis does not work when we not only specify the system in CSP-OZ, but also its environment, as discussed in Chapter 4.1. For example, consider an environment process that starts with a monitored event ${}^{m}a$ and then reacts to any of the controlled events with a further event ${}^{m}a$: $E = {}^{m}a \rightarrow ({}^{c}b \rightarrow E \, \Box \, {}^{c}c \rightarrow E)$. Since the environment shall accept any controlled events, we have the external choice operator here. But when we compose the two processes S and E by parallel composition, $W = S \parallel E$, the resulting description of the world will deadlock after two events. Therefore, this naive composition approach does not work. This is because we have assumed the synchronicity hypothesis for both parts, and in a contradictive way. Each side assumes that it must be prepared to react to one event only and is then allowed to insist on an arbitrary number of events before it "releases control" again.

Therefore, when we write a specification consisting of both a system description and an environment description, we need something that breaks this cycle of dependence. The missing parts are the relations that describe the monitoring precision and the control tolerance. The control tolerance decouples the "ideal" system reaction from its physical realisation. For example, if two controlled events occur "closely together", then this relation determines when and how the environment will notice. One way could be that the events are buffered in a FIFO style and then are consumed by the environment process later. Another way could be that the environment does not notice all events that are "too close", such that the buffer relation will present both events to the environment, but it will drop silently the second after the first has been noticed. Analogous arguments apply to the monitoring precision, which can be regarded as the "output buffer" of the environment process.

For small specifications, it is not mandatory to specify these buffers as explicit sub-processes. Alternatively, we can specify just two monolithic processes, one describing the system and the control tolerance, and one describing the environment and the monitoring precision. The first process must be prepared to *always* take part in monitored events, and the second to *always* take part in controlled events. This may mean that these processes will buffer pending incoming events internally when more incoming events happen.

It is easy to specify two simple processes that do nothing than to take part in all required events, and then we can prove by a suitable refinement relation that our two complex processes have these properties, too. This can be done, for example, with a model checking tool like FDR.

For large specifications, we can reduce the effort of understanding them if we are able to factor out the buffer part from the "ideal" behaviour into separate sub-processes. An explicit separation into an "ideal" system process and a buffer is to be prefered, of course, when we also separate mode class definitions from the definitions of controlled events, as discussed in Section 5.3 below, in order to impose incremental constraints separately.

We have developed ideas recently on how we can structure a CSP(-OZ) process such that it consists of separate parts for the mode classes, the controlled events, and the buffers. These ideas include how we can specify the behaviour of the mode classes by CSP processes.

Unfortunately, we were not yet able to incorporate them in the case study yet. The case study separates the mode classes and the controlled events, only. And it uses the Object-Z part to specify the mode classes. Instead of an explict buffer, it employs a different interpretation of the specification.

Our interpretation specifies requirements as a relation between monitored events, as before, and *controlled variables*. The values of the controlled variables are represented by whether the controlled events are enabled, and with which parameters. For example, a boolean variable is represented by the possibility or impossibility of the system to take part in an event. An integer variable is represented by the parameter value that the system allows for an event.

Therefore when we test a behaviour equivalence between a requirements specification and some implementation, the requirements specification must allow the controlled events that correspond to the observed values for controlled variables of the implementation, in order for the implementation to be correct.

As a side remark we note that the SCR variant [HJL96, HJKL+98] of Functional Documentation, too, uses the approach of monitored events and controlled variables.

4.4 Specification Parameters

This specification in the appendix is loose in that it allows several values for some constants, and that its behaviour is non-deterministic. When the system is implemented, concrete values and decisions will have to be supplied. But we distinguish two different cases here: either the developer is allowed to make some arbitrary choices that suit his design needs, or the customer reserves the right to choose. For example, the initial set of users will be supplied by the customer. The developer should try not to restrict this set at all. If he cannot find any (sensible) implementation without further conditions, he may restrict this set if the customer agrees. For example, they may negotiate a specific syntax for user IDs.

We indicate this distinction between constants by prefixing them with superscripts: "$^{=}\_$" denotes a developer's choice ("normal constant"), and "$^{p}\_$" denotes a customer's choice ("specification parameter").

4.5 Linking Formal Requirements and Reality

We use formal CSP-OZ specifications as a means to specify software requirements precisely. This requires a precise description of what the CSP-OZ events denote in reality; such designations necessarily must be in natural language. In our approach, we provide them by a CSP-OZ comment at the point of definition of the event.

The Functional Documentation approach uses a tabular notation to provide this information: a table contains lines which each list a variable name, whether it is a monitored or a

$^{P}ConChannels$

inherit $^{P}UserFun$

...

chan $^{m}connectReq : [\, u?, v? : {}^{m}userIds \,]$	Monitored event: user u requests a connection to user v.
chan $^{c}connectInd : [\, u : {}^{m}userIds \,]$	Controlled event: user u is offered a connection.
chan $^{m}connectRsp : [\, u? : {}^{m}userIds \,]$	Monitored event: user u accepts an offered connection.
chan $^{c}connectCnf : [\, u : {}^{m}userIds \,]$	Controlled event: the previous connection request of user u has been accepted.

...

Figure 4.3: Example of a tabular channel definition specification text layout.

controlled variable, the designation in natural language, and the variable's data type. Such a description is compact and well readable.

Therefore we attempt to provide the same information in a similar layout. The standard form of definition for a channel in CSP-OZ provides the name and possible parameters of a channel, and each definition can be written on one line each. We add the information of whether it is a monitored or controlled event by prefixing all monitored events by "^{m}_", and by prefixing all controlled events by "^{c}_".

Furthermore, we designed a LaTeX style `rcomment.sty` that allows to provide a CSP-OZ comment to the right of a definition at a fixed column position. This renders the overall effect of a tabular notation. We also use this notation for the definition of CSP-OZ variables. Figure 4.3 presents an excerpt from the channel definitions in the appendix on page 38.

Chapter 5

Composing Modular Requirements Using CSP-OZ

This chapter is the core part of this report, it presents our approach, and how we apply it with CSP-OZ.

5.1 Modularizing Telephone Switching Requirements

In previous work [Bre99c, Bre98b] (see also [Bre99a, Bre99b, Bre98a]), we have investigated how we should structure software requirements with regard to future modifications using the Functional Documentation approach [PaMa95, vSPM93, vS92] to requirements specification, also known as "Parnas tables".

One aspect of the Functional Documentation approach that facilitates maintenance is that it takes great care to avoid duplication of information in the documents, by factoring out common parts. Such duplication would make it hard to change the documents consistently. Furthermore, the general structure of the requirements documents allows to extract *partial views* of the system by selecting the currently relevant controlled variables only, and those parts of the specification that are part of their definition; as detailed in [Bre99c].

Telephone switching systems are special, in that they have *very few events* at the user interface, compared to the functionality they provide. A typical telephone handset has twelve buttons, a hook switch, and can generate a few signalling tones like the dial tone or the busy tone. Despite this syntactically scarce interface, current systems allow for a host of *features* to which a user can subscribe (e.g., call forwarding on busy or on other conditions, call completion to busy subscriber, distinctive ringing, several kinds of black lists, credit card call, conference call, virtual private network, and many more). Therefore, we cannot partition such kinds of systems by controlled events sufficiently. The situation can be improved when we distinguish between syntactic and semantic events (we discussed this in [Bre98a] for syntactic and semantic variables), and map a larger number of semantic events to the few syntactic ones. Nevertheless, current call processing employs one pair of finite state automata which are involved in all call related processing. Their state can be considered as two auxiliary variables, closely coupled, that are involved in all processing. This bottleneck still prevents a separation of events. This leads to substantial maintenance problems.

In the feature interaction example between the Originating Call Screening and the Abbreviated Dialling feature in Section 1, both features work on exactly the same variable (the dialled number) in about the same stage of call processing. Other number translation features such as 800-FreePhone and Credit Card Call will complicate the situation even further. All these features influence the directory number used for the call processing step of routing, and no partitioning of features according to variables is possible.

Therefore, we resort to a *constraint-oriented specification style*. Adding another constraint by a kind of logical conjunction does not invalidate existing constraints, due to the law $P_1 \wedge P_2 \Rightarrow P_1$. This allows us to structure a requirements specification into many independent partial specifications, even though they constrain the same few events. Obviously, all constraints should be chosen in the least restrictive way possible, in order to leave sufficient room for an implementation. We discuss the question of un-implementable specifications below in Section 5.6.

5.2 Families of Requirements

In the formal refinement approach to the specification of software, one adds more details to the description step by step. We distinguish two categories of refinement application. *Internal refinement* adds implementation details and does not change the visible behaviour of the system (except for a reduction of the internal non-determinism). The *external refinement* of a requirements specification renders a requirements specification, again, by adding more desired constraints, and it results in a more restricted visible behaviour of the system. In both cases, we usually have a linear sequence of refinement steps. A revision of a step potentially invalidates all subsequent steps.

For internal refinement, Parnas's idea of *program families* offers relief. Parnas [Par76] defines program families as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members. He proposes to keep and document the results of the intermediate (internal) refinement steps of the first program and to use them as starting points for further programs. Thus, the family of programs becomes the set of leaves of a refinement tree. Furthermore, he proposes that design decisions that might be changed easily are postponed by using the concept of information hiding modules. An information hiding module is the precise description of the behaviour of a part of a program, without revealing how this behaviour is achieved. Thus, the module encapsulates a design decision as a secret, and the rest of the system cannot be affected by changing this decision later. When we make different design decisions inside a module for different programs, we get different branches in the refinement tree.

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

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

As with families of programs, it is crucial to identify early what are the common parts of the family, on which most other parts will depend. Accordingly, we have to do an *SCV analysis* (scope, commonality, and variability) of the family, as proposed by Weiss and others [CHW98, CuWe98, GJKW97] for families of programs. We do not already specify any conceivable telephone switching system, but we determine which requirements aspects will be common to all conceivable systems, which aspects will be common to only some versions, and which aspects are highly special to the particular system we want to build next. Furthermore, we find out (as far as possible) which requirements aspects are likely to change together.

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

With the program family approach, it is important to not think in terms of processing steps [Par72]. Instead, one must find out which aspects about data and processing can be encapsulated and hidden, and replaced by a simpler interface specification. Only in the last step, one may compose some highly abstracted interfaces to a complete system. Accordingly, in the requirements family approach, one must not think in terms of processing steps, too. Such thinking seduces to specify global constraints that refer to details of the system that might change. For example, one should not assume a specific order of detailed processing, or a particular representation of data. Instead, first one describes those aspects that one expects to never change with high confidence.

Please note that we put together any specific, complete requirements specification by two different operations. On the one hand, we compose requirements on different aspects of the system, i.e., from different branches of the tree, by a kind of logical conjunction. On the other hand, we augment the basic requirements by some requirements variants. This is external refinement. But this external refinement is nothing but a restriction of the possible behaviour of the system, and therefore it is formally equivalent to a restriction of the behaviour by logical conjunction. Accordingly, both kinds of composition are a kind of logical conjunction.

5.3 Incremental Restrictions on Controlled Events

The Functional Documentation approach allows to specify a relation between monitored and controlled variables. Experience has shown that the expressions which define the values of the controlled variables often contain common sub-expressions. In particular, one often can identify one or more *mode classes* of the system. A mode class is a finite set of modes, and the system is always in exactly one mode of each mode class. Identifying these mode classes and using them for the specification of the controlled variables removes redundancy from the requirements document. Less redundancy in turn reduces the likelihood that later changes to the document will not catch all text locations concerned.

Having explicit mode classes is obviously crucial for our particular goals. In the Object-Z part, we represent a mode class by a *free type* listing the modes, and by a variable of this type. In order to make this explicit, we prefix each mode with "$^{M}\_$", and each variable representing a mode class with "$^{C}\_$". In a second step, we specify the controlled events using modes and further conditions through `enable`$\_op$ operations.

This approach facilitates composing incremental restrictions. On the one hand, we can add one restriction to the mode transitions of a mode class, and have this restriction imposed on all controlled events that use this mode class in their definition. On the other hand, we

Table 5.1: The sections of our requirements documents.

1.	introduction
2.	**feature A**
⋮	⋮
7.	**feature F**
8.	**Composition of the Complete Specification**
9.	indices / bibliography

Table 5.2: The structure of a feature section. (Parts not applicable may be omitted.)

$i.1$	**fundamental relevant events** of the feature
$i.2$	**fundamental mode classes and auxiliary functions** of the feature
$i.3$	(fundam. relevant events / mode classes / auxiliary functions of the environment)
$i.4$	**fundamental requirements for the environment** of the feature
$i.5$	**fundamental requirements for the system**, i.e., the feature itself
$i.6$	**changeable relevant events** of the feature
$i.7$	**changeable mode classes and auxiliary functions** of the feature
$i.8$	(chang. relevant events / mode classes / auxiliary functions of the environment)
$i.9$	**changeable requirements for the environment** of the feature
$i.10$	**changeable requirements for the system** of the feature
$i.11$	**collection of all the requirements of this feature**, optionally including a graphical representation of the property dependences

have separate properties that specify the definitions of the mode classes and of the controlled events. When we modify the behaviour of one controlled event only, for example when we add a further parameter, we can replace the definition of the event without also touching the underlying mode class definition.

In CSP-OZ, variables are not part of the semantics of a (requirements) specification, they contribute only indirectly by influencing events. Consequently, our approach does not impose any design decisions, exactly as with the mode classes in Functional Documentation.

When we specify mode classes and controlled variables incrementally, this usually means that *partial* specifications contain non-determinism. Please note that a complete specification will contain much less or no non-determinism.

5.4 Syntactic Structure of One Family Member

We structure the requirements specification of a single family member into the sections shown in Table 5.1. The main part is the sequence of sections describing one feature each. Each of these sections has the substructure shown in Table 5.2. First, there are subsections on the fundamental aspects of the feature, then there are subsections on the changeable parts, and a final subsection collects all requirements together. The separation of fundamental

$^{P}DevNoRem$ – **Telephone Devices neither Added nor Removed**

$^{P}DevNoRem$

`inherit` $^{P}DeviceFun$

$^{m}devices = {^{p}initialDevices}$	Relation: telephone devices are neither added nor removed during the lifetime of the system.

Figure 5.1: Example of a specification of a property as a CSP-OZ class.

and changeable aspects is discussed in Section 5.6. For both parts, we first specify the events, mode classes, and auxiliary functions, and then we specify the actual requirements for the environment and for the system. The separation of events, mode classes, and auxiliary functions from the actual requirements was discussed in Section 5.3.

There may be subsections on the events, mode classes, and auxiliary functions of the environment too, as shown in Table 5.2. These are not part of our approach, but artifacts of a current syntactic restriction in CSP-OZ that will be lifted in the future. This is discussed in Section A.3.3 of the appendix on page 45.

Each of our subsections contains a sequence of one or more CSP-OZ classes, where each CSP-OZ class specifies one *property*, which is the smallest unit of our incremental composition. Figure 5.1 presents an example for such a property from the appendix on page 35. This property contains one restriction on the state space, where the declaration of the state space is inherited from another property.

Inheritance of CSP-OZ classes is the syntactic means by which we represent the external refinement of properties. Each CSP-OZ class may start with an `inherit` section, as in Figure 5.1. As discussed in Section 3.1 on page 7, this merges the declarations of all inherited classes and it conjoins their constraints on the behaviour.

A final subsection collects all requirements of a feature together. We do this in two classes, one for the environment and one for the system requirements. Figure 5.2 presents an example from the appendix on page 50. To facilitate a fast overview, we note as a CSP-OZ comment at each property identifier whether it is a fundamental or a changeable property. Furthermore, this subsection may optionally be supplemented by a directed acyclical graph that displays the inheritance relation among the properties of this features.

The last main section of the document the collects all features together. Again, we do this in two classes for the environment and the system, as shown in Figure 5.3 from the appendix on page 51. Then these two classes are immediately composed to a complete system, as discussed in Section 4.1, with the special interpretation that we oblige the implementer to provide the system part and the customer to provide the environment part. (Compare Figure 4.1 on page 11.)

$^{P}BasicConnectionFeatureEnv$

inherit

$^{P}BasicConnectionFeatureEnvModes$	[(see above)]
$^{P}ConEnvFun$	[fundamental]
$^{P}ConEnvRandom$	[changeable]

$^{P}BasicConnectionFeatureSys$

inherit

$^{P}BasicConnectionFeatureSysModes$	[(see above)]
$^{P}ConFun$	[fundamental]
$^{P}ConnectRelIndReasonUNotAvail$	[changeable]

Figure 5.2: An example of collecting all requirements of a feature together.

$^{P}SimpleSpecificationSys$

inherit

$^{P}UserSpaceFeatureSys$

$^{P}BasicConnectionFeatureSys$

$^{P}SimpleSpecificationEnv$

inherit

$^{P}UserSpaceFeatureEnv$

$^{P}BasicConnectionFeatureEnv$

Figure 5.3: An example for collecting all features together.

5.5 Extending the Requirements Family by Further Members

5.5.1 Refining Existing Features.

The first member of a requirements family is just specified by a requirements document structured as presented above. When we conceive another member of the requirements family, we add further feature requirements sections to the requirements document, and we explicitly state which features each family member comprises. For each member, there is a *list of its features*. Syntactically, this is the list of inherited classes as presented in the previous section, and shown in the example in Figure 5.3.

We interpret each of the CSP-OZ classes that collect an entire system as a separate requirements specification for the respective configuration. The implementer needs to consider only those feature sections of the document that are listed in the respective configuration. A suitable extraction tool may eliminate the other document sections automatically.

Table 5.3 shows an example of the general overall structure of a document that covers several members of a requirements family. The specification α is composed from features A–F, the specification β is composed of features A–C,E,F, only, and the specification γ is composed of features A–C,F and additionally features G and H. Each feature's section appears before its first use.

Table 5.3: Example sections of a requirements document in the general case.

1.	introduction
2.	**feature A**
⋮	⋮
7.	**feature F**
8.	**Composition of Complete Specification** α
9.	**Composition of Complete Specification** β
10.	**feature G**
11.	**feature H**
12.	**Composition of Complete Specification** γ
13.	indices / bibliography

5.5.2 Changing Existing Features.

As long as features are composed by a kind of logical "and", no feature interactions among invariance properties are possible, since all invariance properties are preserved when adding more restrictions. This is due to the law $P_1 \wedge P_2 \Rightarrow P_1$. In practical work, this kind of composition is often not sufficient for the incremental extension of a system. Either, an extension feature needs to make an outright functional change to some pre-exisiting feature. Or, the writer of the pre-existing feature at least needed to add some requirements of the form "if none of the previous conditions applies, then nothing will happen". Otherwise, the requirements specification would leave room for a lot of weird, unexpected behaviour that would not satisfy customers and users. Generally, such requirements remove a lot of non-determinism.

Therefore, when we add a new feature requirement to a system requirements specification, we usually have to remove at least one previously specified property. In some approaches, this is done implicitly, for example by a precedence operator that gives any new feature a higher precedence than the previously added features.[1]

Many approaches provide a way to add features to a base system in some way. In particular operational approaches usually allow more or less arbitrary changes to the base system. An example is the Detection Point concept in the Intelligent Network standard. This renders sufficient expressive power to introduce any desired feature. But we argue that this approach also makes it very hard to understand the consequences of changes. We argue that *the system to modify should have an explicit interface* where it can be modified.

Each family member has a number of features with a number of properties each. When conceiving a new family member and re-using parts of existing features, we do not modify any feature specification syntactically. If necessary, we set up a new feature of which the properties may inherit some of the existing properties.

More often, a new feature F_2 will rely on all the properties of some already existing feature F_1, with the exception of a few default properties, which replaces it by new, more complex properties. We can specify this in CSP-OZ by writing a feature F_1' with classes that inherit all the properties that the classes of F_1 inherit, too, except the few to be left out. Then, we

[1] Such precedences are used, in particular, in operational approaches that specify features by chains of state transition steps.

can build feature F_2 on top of F'_1. In order to facilitate the analysis of feature interaction problems, we should record some meta-information that feature F'_1 really is feature F_1 with a few modifications. This is important when another feature F_3 similarly modifies F_1 to F''_1, and we want to investigate the differences between F'_1 and F''_1.

An alternative solution, that avoids copying information, is to extend our composition operator which composes a feature from properties beyond the capabilities of the CSP-OZ `inherit` operator. We then need a language facility that assembles a list of properties and composes them semantically like the `inherit` operator, but which can be referenced by a derived feature's classes such that they can copy and modify this list before composing the semantics, again.

The standard CSP-OZ `inherit` operator does not support this for a good reason: this kind of property combination would not be compositional anymore. The semantics of the result depends on the structure of the composed items, not only on their partial semantics. Therefore, our imagined operator should be called differently.

These kinds of operations take place on a meta-level beyond the semantics of a single system specification. They are concerned with the generation of single members from the family of requirements, and they include the application of feature interaction detection criteria on this meta-level.

We have worked this out in more detail for Functional Documentation as a specification language [Bre99c]. Unfortunately, we were not yet able to adapt the approach to CSP-OZ. In particular, the "replacement operation" below for properties / CSP-OZ classes still requires work. Therefore, we now just describe the approach for Functional Documentation in the remainder of this section.

If we inherit at least one property of a feature, this feature must be included explicitly into the list of features of the new family member. When we do not want to include all properties of a feature, one of the new properties must state which existing properties are *replaced* by it.

The meaning of a property that replaces other properties is exactly the same as if it would inherit these properties, except that the definitions, relations, etc. of the named properties are *not* included. But the contents of the properties that these properties inherit *are* included: we use the same tree of properties, only the root is removed. A property must not both be replaced and inherited by properties of the same family member (except when the inheriting property itself has already been replaced). A simple syntactic check can reveal the well-formedness of a family member with this respect.

If a property is replaced by more than one new property, this is legal, but it is a hint to a possible interaction, as discussed in Section 5.6 below. Such checks concern the syntactic structure of the specification and can be performed automatically by suitable tools.

The explicit interface for change thus is the set of properties a family member has. This is sufficiently fine-grained to avoid interactions like those caused by the global-jump style of the Intelligent Network's Detection Points, and it is sufficiently coarse-grained to allow a specifier to grasp the consequences of his modifications. Of course, the success depends on that all specifiers put only one aspect into each property. This, in turn, depends on a careful SCV analysis of possible commonalities and variabilities before anything is specified. The case study in [Bre99c] presents some examples.

5.5.3 Incremental Development Tasks.

We stress the distinction between the environment of a system to construct, and the system itself. This allows to use the requirements document as the base of a legal contract between a developer and his customer. Mutual responsibilities are stated clearly.

In the case of maintenance and incremental changes to the original system, there is not only one contract, but an entire *sequence of contracts*. For each of these contracts, the parties involved need to state what has to be done or provided by whom. Additionally, both parties need documentation describing the entire system. At the very least, this facilitates greatly the design of the next contract, since the documentation, modulo some changes, already contains a description of the environment of the next development task to perform.

Therefore, both the environment's and the system's specification of a development task are kept, such that the system's specification can be *relabelled as environment's specification* for the next task.

5.6 Detection of Interaction Problems During Composition

As just mentioned, some properties are not essential to a feature, but need to be specified nevertheless for "technical reasons". For example, we would like to specify that a human and a telephone device are different entities with different properties, but that, for the moment, we assume that the association between humans and telephones is static and bijective.

It happens frequently that we have to remove some properties of a feature in order to define a new member of the requirements family which has more functionality. Such changes to a feature might cause an adverse feature interaction. But one can determine this only with a thorough knowledge of the feature. Therefore, we require already the original specifier of a feature to state which of the properties are *fundamental* to the feature, and which are *changeable*. This is done by putting the properties into the appropriate ones of the four sections in Table 5.2.

The inheritance relation among properties renders us a dependence graph. It is acyclic through the syntax definition of CSP-OZ. This directed acyclic graph of the inheritance relation is related to the idea of the *uses-hierarchy* in software development [Par79]: the "uses" relationship between programs denotes "requires the presence of a correct version of". We can reduce software development problems when we make this relationship acyclic, rendering a hierarchy. A carefully designed hierarchy is vital to achieve a broad family of systems.

Furthermore, we allow environment properties to inherit only other environment properties, while system properties may inherit both environment and other system properties. As an additional guideline, fundamental properties should inherit fundamental properties only, while changeable properties may inherit both fundamental and other changable properties.

Technically, it is allowed in our approach that two properties replace the same property by our syntactic replacement. The property to remove is removed, and both new properties are conjoined into the global meaning of the family member. But such a situation should be looked out for and flagged by any supporting tools, since there is some probability that there exists a feature interaction of another kind: the new properties may be contradictory. This means that the resulting set of allowed behaviours is empty.

Our choice of CSP-OZ facilitates the detection of contradictions since it allows to specify invariance properties only. An invariance property states that something undesired does not happen, but it does not specify any required progress. Therefore, if there exists any initial state that satisfies all specified properties, then there also must be some system behaviours that satisfy these properties. (The simplest are those where just nothing happens.) In this case, we can prove the absence of an empty set of behaviours by determining some valid initial state. In general, the composition of progress properties (and in particular of real-time constraints) is still an area of research. Concerning our approach, restricted progress property checks are feasible by model checking tools: we can prove by exhaustive state space exploration that no deadlocks occur, i.e., that always at least some progress is possible. Such checks depend on a limited size of the state space, in order to avoid state space explosion problems.

Another problem that may arise from our constraint-oriented specification style is that the conjunction of two conditions on the effect of an operation may yield false, or at least yield some condition that is not satisfyable under one of the enabling conditions. According to the semantics of CSP-OZ, this leads to the *divergence* of the behaviour. As long as the effect predicates are sufficiently simple, automated theorem proving could be used to detect such problems. In the current specification, such problems cannot arise, since there is only one operation for each monitored event, and thus no conjunction, and since the effect predicates of controlled events are true anyway.

Since we use the graph of the inheritance relation for detecting feature interaction problems, it is important that the specifiers state explicitly only the minimal dependences among the properties. It is worth investing time for analysis here in order to later save time for the detection and resolution of possible feature interactions. In our experience, a little thought here can reduce the complexity of possible interactions considerably. Furthermore, such analysis often led us to split one large property into several smaller ones that depended much less on each other that it first appeared. An example is our attempt to specify a voice connection in our first case study [Bre98b] (using Functional Documentation, but otherwise the same approach). In a first attempt, we specified one rather large monolithic property. Then we factored out the core modes of it, achieving a rather small mode transition table for it that used only few variables and thus few other properties. Furthermore, we separated the notion of a voice channel from the notion of a connection, which simplified the definition of a connection, and which opened the option to specify later, e.g., a video channel reusing our definition of a connection.

Sometimes, several new features are developed concurrently, and are integrated only later, after they have been completed. Such an integration step consists basically in just appending the requirements specifications of the new features to the main requirements document, and checking for possible feature interactions, as discussed above.

Chapter 6

Related Work

Related work investigates "feature" constructs for different specification and programming languages, in particular in the context of the FIREworks project [SaTü98]. The aim of this project is to define a feature construct for several languages, and then to evaluate it in case studies. The notion of *feature-oriented programming* shall emphasize the importance of features in the system structure. The approach bases on the superimposition idea of Katz [Kat93], where we specify a base system first, and where we then specify textual increments by a suitable operator, for example as done by Plath and Ryan [PlRy98]. One problem of the superimposition idea is that, even though the textual increments have a defined interface, the base system does not. Therefore, a superimposition can invalidate arbitrary assumptions about the base system and thus cause feature interaction problems. This effect is known sufficiently from the Intelligent Network, of which the Detection Points together have about the same potential for modification.

A variant of the Functional Documentation approach, called CoRE [Mil98, FBWK92], allows to partition a specification into "classes" where each class groups together portions that are logically related and likely to change together. The CoRE method requires that all restrictions on a controlled variable are specified within a single class. Therefore, it does not allow our approach of incremental restrictions on the controlled variables. We need these incremental restrictions in our specific application domain due to its small number of controlled variables. Further differences are that the CoRE method does not distinguish between fundamental and changeable properties of a feature, and is generally less centered on the incremental extension and modification aspect that is our focus. Nevertheless, it is a closely related approach, and future research should investigate whether our ideas that go beyond the CoRE method can and should be merged with it.

Chapter 7

Summary and Future Work

Requirements documents for software need not only be written, but they also need to be maintained afterwards. In telephone switching, there arise particular problems due to the strong mutual dependences of telephone features and due to the current rapid change in this area. We attempt to avoid or at least reduce feature interaction problems during the extension or change of a requirements document through a suitable requirements document structure.

In previous work, we conceived an approach for this, based on Parnas' Functional Documentation. In this report, we develop it further. In particular, we apply it with the formal description technique CSP-OZ, in order to take advantage of its worked-out refinement calculus. This renders the necessary formal precision for the composition operators without the need to define all their details from scratch. A preliminary case study in the appendix shows that we can indeed structure a requirements document in a suitably modular way, using CSP-OZ.

In our approach, we specify requirements only, without introducing any implementation decisions yet, such that the requirements document can be used as a base for a contract between a developer and a customer. We are concerned with many requirements documents for all the different telephone switching systems that result from choosing optional features, and from adding further features. A new insight in this work is that such a set of requirements documents should be treated as a *requirements family.* We transfer the idea of program families, which exists in software design, to the area of requirements engineering. In particular, we apply the idea of information-hiding modules to requirements documents. Each module encapsulates choices of requirements variants that are likely to change together, and it hides the choice of a particular requirements variant set from readers of other modules.

Our main idea we use from Functional Documentation (which has proven successful during the long maintenance period of the A-7 flight computer requirements document [AFHB+92]) is that a requirements document should contain *no duplicated information*, in order that it cannot become inconsistent through modifications. Our requirements family idea extends the scope of this to a set of related requirements documents. Furthermore, we require the orginal specifier of a system (and any specifier of a modification of it) to *document explicitly* a number of informations that specifiers of later modifications will need, but which are difficult or impossible to extract later, and whose absence can cause feature interaction problems.

We specify a complete requirements document by many increments, each consisting in one requirements module. In order to add required properties incrementally, we employ a *constraint-oriented* specification style. We express the requirements modules by the CSP-

OZ language construct of the class, and we express the incremental extension by syntactical inheritance. We use a restricted version of inheritance, without CSP-OZ's usual option to replace inherited CSP-OZ operations; the effect is that the inheritance becomes a kind of conjunction.

We use information hiding modules recursively. This renders a directed acyclic graph for the uses-hierarchy among these modules. In particular, we avoid having a linear sequence of increments where their dependences are not explicit. Accordingly, we employ multiple inheritance to specify explicitly on which other modules each module must rely.

Besides the explicit uses-hierarchy, our specification approach comprises further measures that support avoiding and detecting interaction problems caused by extensions and changes: we distinguish explicitly between those properties of a feature that define it fundamentally, and those properties that may be changed later by other features without causing adverse feature interactions. The approach restricts changes to a feature to replacing or adding entire so-called properties; this facilitates the analysis of the changes.

These measures add to those already used in Parnas' Functional Documentation approach: separation of the requirements on the system to construct from the requirements on its environment; accordingly an explicit distinction between events that the system can only monitor from the events it controls; and the extraction of the definition of (multiply used) mode classes from the specification of the behaviour of controlled events.

Our general experience with CSP-OZ is that it is possible to formally specify telephone switching requirements in CSP-OZ, and that it is possible to structure them in the way we propose to reduce and avoid requirements maintenance problems. CSP-OZ is a formalism developed recently. Its improvements over Object-Z and CSP prove valuable for our approach, for example it allows an explicit distinction between monitored and controlled operations, this is neither possible in Object-Z nor in CSP. Conversely, our case study contributed to the development of CSP-OZ, whose author was eager to integrate ideas arising during the case study. Another experience is that tool support is important, the type checker cspozTC helped considerably in the case study. The worked-out formal semantics of CSP-OZ was – as expected – a substantial aid to our work on incremental extension.

Finally we experienced that the textual layout of a specification is one of the factors that determine its readability; therefore we investigated how we can arrange the text of a CSP-OZ requirements specification in a compact and table-like way, similar to Functional Documentation.

The work reported here is still preliminary. We were not yet able to investigate several questions. In particular, in this report we do not fully integrate the CSP part of CSP-OZ into our approach. This concerns the specification of mode classes in CSP, and also the separation of the definition of controlled events from the definition of the associated control tolerance. We have ideas about how to do this, but we did not yet have sufficient time to rework the case study accordingly.

Even though the approach already provides the basic information for the syntactic detection of remaining feature interactions, we still have to define precisely the syntactic criteria. Similarly, an automated validation of these criteria is obviously possible, but it needs to be implemented still. Furthermore, we have sketched ideas about meta-operators for manipulating the list of properties of a feature, but we were not yet able to work them out for CSP-OZ. Generally, the current case study needs considerable extension to provide evidence of practical usefulness, and we would like to perform further case studies, too.

Some more general areas of interesting research are how to perform a constraint-oriented composition of "unbounded progress" properties, and how to specify real time properties in CSP-OZ.

Appendix A

Case Study

A.1 Introduction

A.1.1 Notation

A requirements specification comprises many different kinds of identifiers. It is helpful for the reader if he can determine the particular kind of identifier immediately, for example, whether it denotes a monitored or a controlled event. It is customary for Functional Decoration to decorate all identifiers according to their kind, using non-alphanumeric characters like "!,%,$". Since these decorations have no mnemonic value, and thanks to the advances in typesetting, we prepend a superscript character to each identifier instead. These characters are chosen to remind of the kind of identifier. The entire set of characters used is presented in Table A.1.

A.1.2 Naming Conventions

We choose the identifiers in the specification such that they help to guess and to remember their meaning. When we combine several English words into one identifier, we start each component by a capital letter. The first character is in lowercase, except for properties, where it is in uppercase. When names threaten to become too long, we abbreviate some of the components. Table A.2 lists abbreviations used.

A.1.3 Extensions to the Mathematical Toolkit

Since the Z builtin **if-then-else** statement is a predicate, no expression, we need another one that is a conditional expression.

$$[X]$$

$$\mathit{ifthen} : \mathbb{B} \times X \times X \rightarrow X$$

$$\forall\, a, b : X \bullet \mathit{ifthen}(\text{true}, a, b) = a \wedge \mathit{ifthen}(\text{false}, a, b) = b$$

Example: $\mathit{result} = 42 + \mathit{ifthen}((i = 5), 3, 7)$. If $i = 5$, this renders $\mathit{result} = 45$, otherwise $\mathit{result} = 47$. Please note that the second and third parameters must be defined properly only in the case they are used. If a parameter is undefined, the Z semantics assumes that it may have any value.

We need some boolean operators that return expressions, instead of being a predicate.

Table A.1: Letters to indicate kind of identifier.

letter	**kind of identifier**
m	monitored event or variable
c	controlled event or variable
e	internal event
t	type
$=$	constant
p	specification parameter (customer-chosen constant)
M	mode
C	mode class
f	auxiliary function
P	property

Table A.2: Abbreviations in identifiers.

abbrev.	**expanded form**
Assoc	association
Avail	available
Cnf	confirm
Con	connection related
D	design phase
Desir	desirable
Dev	device
Env	environment related
Fun	fundamental
Hum	human
Id	identifier
Ind	indication
NoRem	not removed
O	orgininating side related
Orig	orgininating side related
Rel	release
Res	resources
Rol	role
Sys	system related
T	terminating side related
Term	terminating side related
U	user
Req	request
Rsp	response

$$\begin{array}{|l}
_ \mathit{and} _ : (\mathbb{B} \times \mathbb{B}) \rightarrow \mathbb{B} \\
_ \mathit{or} _ : (\mathbb{B} \times \mathbb{B}) \rightarrow \mathbb{B} \\
_ \mathit{imp} _ : (\mathbb{B} \times \mathbb{B}) \rightarrow \mathbb{B} \\
\mathit{not} : \mathbb{B} \rightarrow \mathbb{B} \\
\hline
\mathrm{true}\ \mathit{and}\ \mathrm{true} \\
\forall b : \mathbb{B} \bullet \neg (b\ \mathit{and}\ \mathrm{false}) \wedge \neg (\mathrm{false}\ \mathit{and}\ b) \\
\neg (\mathrm{false}\ \mathit{or}\ \mathrm{false}) \\
\forall b : \mathbb{B} \bullet (b\ \mathit{or}\ \mathrm{true}) \wedge (\mathrm{true}\ \mathit{or}\ b) \\
\mathrm{true}\ \mathit{imp}\ \mathrm{true} \\
\neg (\mathrm{true}\ \mathit{imp}\ \mathrm{false}) \\
\forall b : \mathbb{B} \bullet \mathrm{false}\ \mathit{imp}\ b \\
\neg (\mathit{not}(\mathrm{true})) \\
\mathit{not}(\mathrm{false})
\end{array}$$

We need some general generic operators that express equality, inequality, and set (non-)membership, and that return a boolean expression, instead of being a predicate.

$$\begin{array}{|l|}
\hline
[X] \\
\hline
_ \mathit{eq} _ : (X \times X) \rightarrow \mathbb{B} \\
_ \mathit{neq} _ : (X \times X) \rightarrow \mathbb{B} \\
_ \mathit{mem} _ : (X \times \mathbb{P}\, X) \rightarrow \mathbb{B} \\
_ \mathit{nmem} _ : (X \times \mathbb{P}\, X) \rightarrow \mathbb{B} \\
\hline
\forall a, b : X \bullet \textbf{if}\ a = b\ \textbf{then}\ a\ \mathit{eq}\ b\ \textbf{else}\ \mathit{not}(a\ \mathit{eq}\ b) \\
\forall a, b : X \bullet \textbf{if}\ a \neq b\ \textbf{then}\ a\ \mathit{neq}\ b\ \textbf{else}\ \mathit{not}(a\ \mathit{neq}\ b) \\
\forall a : X;\ b : \mathbb{P}\, X \bullet \textbf{if}\ a \in b\ \textbf{then}\ a\ \mathit{mem}\ b\ \textbf{else}\ \mathit{not}(a\ \mathit{mem}\ b) \\
\forall a : X;\ b : \mathbb{P}\, X \bullet \textbf{if}\ a \notin b\ \textbf{then}\ a\ \mathit{nmem}\ b\ \textbf{else}\ \mathit{not}(a\ \mathit{nmem}\ b) \\
\hline
\end{array}$$

For multiple choices, we need a generalized select expression, too.

$$\begin{array}{|l|}
\hline
[X] \\
\hline
\mathit{select} : (\mathrm{seq}(\mathbb{B} \times X)) \rightarrow X \\
\mathit{select}_d : (\mathrm{seq}(\mathbb{B} \times X) \times X) \rightarrow X \\
\hline
\forall s : \mathrm{seq}(\mathbb{B} \times X) \bullet \forall i : 1 \,..\, \# s \bullet \mathbf{first}(s(i)) \Rightarrow \mathit{select}(s) = \mathbf{second}(s(i)) \\
\forall s : \mathrm{seq}(\mathbb{B} \times X);\ x : X \bullet \mathit{ifthen}((\mathrm{dom}(\mathrm{ran}\, s))\ \mathit{eq}\ \{\mathrm{false}\}, \mathit{select}_d(s, x)\ \mathit{eq}\ x, \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{select}_d(s, x)\ \mathit{eq}\ \mathit{select}(s)) \\
\hline
\end{array}$$

Example: $\mathit{result} = 42 + \mathit{select}\langle((i < 5), 2), ((i = 5), 3), ((i > 5), 4)\rangle$ This renders *result* equal to 44, 45, or 46, depending on i. An alternative notation would be: $\mathit{result} = 42 + \mathit{select}\langle(i < 5) \mapsto 2, (i = 5) \mapsto 3, (i > 5) \mapsto 4\rangle$. select_d is a variant that provides a default value in case that none of the guards is true. Please note that if more than one guard is true, the results must agree.

A.2 User Space Feature

This feature provides the notions of *humans*, *users*, and *devices*, and it constructs a state space over these. There is not much behaviour specified yet; only the changeable behaviour part specifies a number of invariances over time.

A.2.1 Fundamental Requirements for the Environment

$^{P}HumanFun$ – **Humans**

This specification is generic in the sense that we leave open some parameters, and only give a range of values. We can instantiate a concrete specification by selecting a set of concrete parameter values; for example, by providing a name directory.

$[^{m}humanIdSpace]$ — Monitored: the basic type from which human IDs come.

$^{m}humanIds : \mathbb{F}\, ^{m}humanIdSpace$ — Monitored: the (finite) set of identities of humans that have telephone access.

$^{P}HumanFun$

$^{p}initialHumans : \mathbb{F}\, ^{m}humanIds$ — Specification parameter: the initial (finite) set of humans that have telephone access.

$^{m}humans : \mathbb{F}\, ^{m}humanIds$ — Monitored: the current (finite) set of humans that can use the telephone system.

Init

$^{m}humans = {}^{p}initialHumans$

$^{P}DeviceFun$ – **Telephone Devices**

$[^{m}deviceIdSpace]$ — Monitored: the basic type from which device IDs come.

$^{m}deviceIds : \mathbb{F}\, ^{m}deviceIdSpace$ — Monitored: the (finite) set of identities of telephone devices.

$^{P}DeviceFun$

inherit $^{P}HumanFun$

$^{p}initialDevices : \mathbb{F}\, ^{m}deviceIds$ — Specification parameter: the initial (finite) set of telephone devices.

$^{p}initialPossessedDevices : {}^{p}initialHumans \rightarrow \mathbb{F}\, ^{p}initialDevices$ — Specification parameter: the initial (finite) set of telephone devices that a human has under his physical control.

$^{m}devices : \mathbb{F}\, ^{m}deviceIds$ — Monitored: the current (finite) set of telephone devices that can be used.

$^{m}possessedDevices : {}^{m}humans \rightarrow \mathbb{F}\, ^{m}deviceIds$ — Monitored: the (finite) set of telephone devices that a human currently has under his physical control.

Init

$^{m}devices = {}^{p}initialDevices$

$^{m}possessedDevices = {}^{p}initialPossessedDevices$

$^{P}UserFun$ – User Roles

In the environment of the telephone system, we distinguish the two concepts of *humans* and *users*. Humans are natural humans, as understood by the term "individual". A user is a social role that can be played by humans, and it can be played by different humans at different times. When it is clearer, we sometimes say "user role" instead of "user".

For example, John Smith, sitting in his office near his phone, is a human, as well as Mary Taylor in the office next door. The service hotline of their company is a user (role). This user role can be played by John. During his lunchtime, the user role can be played by Mary. During peak times, both may play this user role. During nighttime, nobody plays this user role. A human can play several user roles at the same time. For example, John Smith plays also the user role "private person John Smith", and, if we differentiate further, "father of Jody Smith".

Formally, we describe the relation between a human and his or her user roles as a monitored variable. This monitored variable describes the set of current roles of the human.

$[{}^{m}userIdSpace]$ — [Monitored: The basic type from which user IDs come.]

$^{m}userIds : \mathbb{F}\, {}^{m}userIdSpace$ — [Monitored: The (finite) set of identities of user roles that have telephone access.]

$^{P}UserFun$

inherit $^{P}HumanFun$

$^{p}initialUsers : \mathbb{F}\, {}^{m}userIds$ — [Specification parameter: the initial (finite) set of identities of user roles that have telephone access.]

$^{p}initialUserRoles : {}^{p}initialHumans \rightarrow \mathbb{F}\, {}^{p}initialUsers$

[Specification parameter: The initial (finite) set of user roles that a user plays.]

$^{m}users : \mathbb{F}\, {}^{m}userIds$ — [Monitored: the current (finite) set of users (i.e., roles that humans can play) relevant to the telephone system.]

$^{m}userRoles : {}^{m}humanIds \rightarrow \mathbb{F}\, {}^{m}userIds$

[Monitored: the (finite) set of users (i.e., roles) that a human currently plays.]

Init

$^{m}users = {}^{p}initialUsers$

$^{m}userRoles = {}^{p}initialUserRoles$

A.2.2 Changeable Requirements for the Environment

${}^{P}HumNoRem$ – **Humans neither Added nor Removed**

This property makes the implementation of the system considerably easier since it relieves us from implementing a lot of administration features. For a realistic, long running system, this requirement on the environment should be dropped and replaced by some more elaborate scheme. For simplicity, we don't do this in our case study.

${}^{P}HumNoRem$

`inherit` ${}^{P}HumanFun$

$${}^{m}humans = {}^{p}initialHumans$$

[Relation: humans are neither added nor removed during the lifetime of the system.]

${}^{P}DevNoRem$ – **Telephone Devices neither Added nor Removed**

This is a similar simplification as the above one, with similar consequences.

${}^{P}DevNoRem$

`inherit` ${}^{P}DeviceFun$

$${}^{m}devices = {}^{p}initialDevices$$

[Relation: telephone devices are neither added nor removed during the lifetime of the system.]

${}^{P}DevAssocNoRem$ – **Telephone Device Associations neither Added nor Removed**

${}^{P}DevAssocNoRem$

`inherit`
${}^{P}DeviceFun$
${}^{P}HumNoRem$

$$\forall h : {}^{m}humans \bullet$$
$${}^{m}possessedDevices(h) = {}^{p}initialPossessedDevices(h)$$

[Relation: telephone device associations are neither added to nor removed from a human during the lifetime of the system.]

${}^{P}UserNoRem$ – **Users neither Added nor Removed**

$^{P}UserNoRem$

inherit $^{P}UserFun$

$^{m}users = {}^{p}initialUsers$

[Relation: users are neither added nor removed during the lifetime of the system.]

$^{P}UserRolNoRem$ – **User Roles neither Added nor Removed**

This property is one of the implicit assumptions that the Plain Old Telephone Service (POTS) makes, since it essentially equals humans and user roles. It is a rather restrictive assumption, again.

$^{P}UserRolNoRem$

inherit
    $^{P}UserFun$
    $^{P}HumNoRem$

$$\forall h : {}^{m}humans \bullet$$
$$\quad {}^{m}userRoles(h) = {}^{p}initialUserRoles(h)$$

[Relation: user roles are neither added to nor removed from a human during the lifetime of the system.]

$^{P}UserOneHuman$ – **One Human per User Role**

$^{P}UserOneHuman$

inherit $^{P}UserFun$

$$\forall h_u, h_v : {}^{m}humans;\ u : {}^{m}users \bullet$$
$$\quad (u \in {}^{m}userRoles(h_u) \land u \in {}^{m}userRoles(h_v)) \Rightarrow (h_u = h_v)$$

[Relation: currently, a user role may be played by at most one human at a time.]

A.2.3 Collection of all the Requirements of this Feature

The class $^{P}UserSpaceFeatureEnv$ collects all properties concerning the user space environment:

$^{P}UserSpaceFeatureEnv$

inherit

$^{P}HumanFun$	[fundamental]
$^{P}DeviceFun$	[fundamental]
$^{P}UserFun$	[fundamental]
$^{P}HumNoRem$	[changeable]
$^{P}DevNoRem$	[changeable]
$^{P}DevAssocNoRem$	[changeable]
$^{P}UserNoRem$	[changeable]
$^{P}UserRolNoRem$	[changeable]
$^{P}UserOneHuman$	[changeable]

The class $^{P}UserSpaceFeatureSys$ collects all properties concerning the system. There are none, and we specify it for the sake of being systematical, only:

$^{P}UserSpaceFeatureSys$

A.3 Basic Connection Feature

The basic connection feature provides the basic notion of a connection, and describes how a connection can be set up and torn down. No details are yet provided on the users' interface for this. Neither do we yet say what kind of connection is provided; there might be, e.g., voice channel or video channel features that build on this feature.

One of the biggest conceptual problems in current telephone systems is the notion of *call*, which has become fuzzy since long (compare, e.g., [Zav97]). Therefore, we only use the notion of *connection*.

Usually, telephone connections are described as the sum of two partial views: the view of the originating side and the view of the terminating side. When we specify the system requirements using such a structure, it is easy to find a distributed design that employs one processing component for each side. Unfortunately, this scheme becomes more complicated when we want to have more than one connection per user. Alternatively, we can describe each connection between two users by only one mode class. It turns out that the description becomes more concise and we feel that it is better understandable. Therefore, we follow this second approach.

The specification states nothing about users for which currently no telephones exist, except that they cannot be called by, or call, users which have telephones, while employing the services of our system. Everything else about users without telephones provided by us is irrelevant.

We investigate here how a *desirable* architectural structure for a telephone system looks like, therefore we are currently *not* interested in how to interface a new telephone system with the existing ones.

In [Bre98b], we discussed how to choose the interface line between the system and its environment. As a result, the monitored and controlled variables should be the users' decisions and perceptions, respectively. Thus we draw the interface line very close to the users; in fact, we draw it even a little inside the heads of the users. In particular, this implies that all network operator companies belong entirely to the system. A further consequence is that the shape of the physical users' interface becomes a question of the system design, not of the system requirements.

A.3.1 Fundamental Relevant Events

$^{P}ConChannels$ – **Connection Channels**

$^{t}connectRelReasons$ ::= $^{=}uNotAvail$ [Type: reasons why a connect release might happen.]

$^{P}ConChannels$

inherit $^{P}UserFun$

chan $^{m}connectReq : [\, u?, v? : {}^{m}userIds \,]$	Monitored event: user u requests a connection to user v.
chan $^{c}connectInd : [\, u : {}^{m}userIds \,]$	Controlled event: user u is offered a connection.
chan $^{m}connectRsp : [\, u? : {}^{m}userIds \,]$	Monitored event: user u accepts an offered connection.
chan $^{c}connectCnf : [\, u : {}^{m}userIds \,]$	Controlled event: the previous connection request of user u has been accepted.
chan $^{m}connectORelReq : [\, u? : {}^{m}userIds \,]$	Monitored event: user u requests to release the connection / all connections which he has initiated.
chan $^{m}connectTRelReq : [\, u? : {}^{m}userIds \,]$	Monitored event: user u requests to release the connection / all connections to him which he has not initiated.
chan $^{c}connectRelInd : [\, u : {}^{m}userIds;\ r : \mathbb{F}\, {}^{t}connectRelReasons \,]$	Controlled event: user u is indicated that a connection is released because of reasons r.

A.3.2 Fundamental Mode Classes and Auxiliary Functions

$^{P}ConFunUserAvailable$ – **User Available for Connection**

$^{P}ConFunUserAvailable$

inherit $^{P}UserFun$

$^{t}userAvailableType$ — [Mode classes: The modes of mode classes $^{C}userAvailable\ (u, v)$ below describe the availability of a resource: whether a user v is already busy, with respect to some user u, or whether u can engage v in a new connection.
When new features will be added to the system, the definition of these modes probably will be refined by introducing sub-modes.]

$::=\ ^{M}userAvail$ — [Mode $^{M}userAvail$: user v is engaged in no connection and therefore can accept a connection from user u.]

$|\ ^{M}userNotAvail$ — [Mode $^{M}userNotAvail$: user v is engaged in some connection and therefore cannot accept a connection from user u.]

$^{C}userAvailable : {}^{m}userIds \times {}^{m}userIds \rightarrow {}^{t}userAvailableType$

Init

true — [initial modes: any]

$^{P}ConFunConnectionDesignResources$ – **Design Resources Available for Connection**

$^{P}ConFunConnectionDesignResources$

inherit $^{P}ConFunUserAvailable$

$^{t}connectionDesignResourcesType$ — [Mode classes: The modes of mode classes $^{C}connectionDesignResources\ (u, v)$ below describe whether there are sufficient resources such as lines available to establish a connection and, e.g., a voice channel from user u to user v, or, if already existing, to maintain them.
These are the resources that will be introduced by design decisions only, and which are not part of the requirements specification.]

$::=\ ^{M}connectionDesignResourceAvail$

[there are sufficient lines, trunks, etc. to connect user u to user v.]

$\mid {}^{M}connectionDesignResourceNotAvail$ — [there are not sufficient lines, trunks, etc. to connect user u to user v.]

`local_channel` ${}^{e}connectionDesignResourcesChange$ — [Some spontaneous event internal to the implementation of the system.]

$${}^{C}connectionDesignResources : {}^{m}userIds \times {}^{m}userIds \to {}^{t}connectionDesignResourcesType$$

Init

true [initial modes: any]

com_${}^{e}connectionDesignResourcesChange$

$\Delta({}^{C}connectionDesignResources, {}^{C}userAvailable)$

true

${}^{P}ConFunConnectionResources$ – Resources Available for Connection

${}^{P}ConFunConnectionResources$

`inherit` ${}^{P}ConFunConnectionDesignResources$

${}^{t}connectionResourcesType$ — [Mode classes: The modes of mode classes ${}^{C}connectionResources\ (u, v)$ below describe whether there are sufficient of the diverse resources available to establish a connection from user u to user v, or, if already existing, to maintain them. These mode classes collect all the preconditions for a successful connection.]

$::= {}^{M}connectionResourceAvail$ — [sufficient resources for a connection from user u to user v are available.]

$\mid {}^{M}connectionResourceNotAvail$ — [sufficient resources for a connection from user u to user v are not available.]

$^{C}connectionResources : {}^{m}userIds \times {}^{m}userIds \rightarrow {}^{t}connectionResourcesType$

$\forall\, u : {}^{m}users;\ v : {}^{m}userIds \bullet$
$\quad {}^{C}connectionDesignResources(u, v) = {}^{M}connectionDesignResourceNotAvail$
$\quad \lor\ {}^{C}userAvailable(u, v) = {}^{M}userNotAvail$
$\Rightarrow {}^{C}connectionResources(u, v) = {}^{M}connectionResourceNotAvail$

[relations on mode classes ${}^{C}connectionResources\ (u, v)$: minimally, the user role to connect to must be currently available for a connection to succeed, and the resources provided by design decisions must be sufficent.]

Init

true [initial modes: any]

com_${}^{e}connectionDesignResourcesChange$

$\Delta({}^{C}connectionResources,$ [The new mode class may be changed by this operation.]
$\quad {}^{C}connectionDesignResources, {}^{C}userAvailable)$

true

${}^{P}ConFunConnectionPhase$ – **Connection Phase**

${}^{P}ConFunConnectionPhase$

inherit
$\quad {}^{P}ConChannels$
$\quad {}^{P}ConFunConnectionResources$

${}^{t}connectionPhaseType$ [Mode classes: The modes of mode classes ${}^{C}connectionPhase\ (u, v)$ below describe whether there is currently a connection between the two users u and v, and if so, whether it is currently in the setup phase, the conversation phase, or the teardown phase.]

$\quad ::=\ {}^{M}conIdle$ [there exists no connection between users u and v.]

$\quad \mid {}^{M}conSetup$ [user v is being alerted of a connection request by user u.]

$\quad \mid {}^{M}conEstablished$ [user u is engaged in a connection to user v and both can use it for communication; user u has initiated this connection.]

| $^{M}conBothTeardown$ — [a connection of user u to user v is currently torn down, and both users must still acknowledge this; user u has initiated this connection.]

| $^{M}conOrigTeardown$ — [a connection of user u to user v is currently torn down, but only user u must still acknowledge this; user u has initiated this connection.]

| $^{M}conTermTeardown$ — [a connection of user u to user v is currently torn down, but only user v must still acknowledge this; user u has initiated this connection.]

$$^{C}connectionPhase : {}^{m}userIds \times {}^{m}userIds \to {}^{t}connectionPhaseType$$

Init

true [initial modes: any]

effect_$^{m}connectReq$

$\Delta(^{C}connectionPhase,$
$\quad ^{C}connectionResources, {}^{C}connectionDesignResources, {}^{C}userAvailable)$

[Include the Δ list of `com_`$^{e}connectionDesignResourcesChange$.]

$u?, v? : {}^{m}userIds$

$$\begin{aligned}
&^{C}connectionPhase' = {}^{C}connectionPhase \oplus \{(u?, v?) \mapsto \\
&\quad ifthen(^{C}connectionPhase(u?, v?)\ eq\ {}^{M}conIdle, \\
&\qquad ifthen(^{C}connectionResources'(u?, v?)\ eq\ {}^{M}connectionResourceAvail, \\
&\qquad\quad {}^{M}conSetup, {}^{M}conOrigTeardown), \\
&\qquad {}^{C}connectionPhase(u?, v?))\}
\end{aligned}$$

$\mathtt{effect\_}{}^{m}connectRsp$

$$
\begin{array}{l}
\Delta({}^{C}connectionPhase, \\
\quad {}^{C}connectionResources, {}^{C}connectionDesignResources, {}^{C}userAvailable)
\end{array}
$$

[Include the Δ list of $\mathtt{com\_}{}^{e}connectionDesignResourcesChange$.]

$u? : {}^{m}userIds$

$$
\begin{array}{l}
\forall v : {}^{m}userIds \bullet \\
\quad ({}^{C}connectionPhase(v, u?) \neq {}^{M}conSetup) \Rightarrow \\
\quad\quad ({}^{C}connectionPhase'(v, u?) = {}^{C}connectionPhase(v, u?)) \\
\forall v : {}^{m}userIds \bullet \exists w : {}^{m}userIds \bullet \\
\quad ({}^{C}connectionPhase(v, u?) = {}^{M}conSetup) \Rightarrow \\
\quad\quad {}^{C}connectionPhase(w, u?) = {}^{M}conSetup \land \\
\quad\quad ({}^{C}connectionPhase'(w, u?) = \\
\quad\quad\quad ifthen((({}^{C}connectionResources(w, u?)\ eq\ {}^{M}connectionResourceAvail) \\
\quad\quad\quad\quad and\ ({}^{C}connectionResources'(w, u?)\ eq\ {}^{M}connectionResourceAvail)), \\
\quad\quad\quad {}^{M}conEstablished, {}^{M}conBothTeardown))
\end{array}
$$

$\mathtt{effect\_}{}^{m}connectORelReq$

$$
\begin{array}{l}
\Delta({}^{C}connectionPhase, \\
\quad {}^{C}connectionResources, {}^{C}connectionDesignResources, {}^{C}userAvailable)
\end{array}
$$

[Include the Δ list of $\mathtt{com\_}{}^{e}connectionDesignResourcesChange$.]

$u? : {}^{m}userIds$

$$
\begin{array}{l}
{}^{C}connectionPhase'(u?, u?) = {}^{M}conIdle \\
\forall v : {}^{m}userIds \mid v \neq u? \bullet {}^{C}connectionPhase'(u?, v) = \\
\quad select_{d}(\\
\quad\quad \langle {}^{C}connectionPhase(u?, v)\ eq\ {}^{M}conSetup \mapsto {}^{M}conIdle, \\
\quad\quad {}^{C}connectionPhase(u?, v)\ eq\ {}^{M}conEstablished \mapsto {}^{M}conTermTeardown, \\
\quad\quad {}^{C}connectionPhase(u?, v)\ eq\ {}^{M}conOrigTeardown \mapsto {}^{M}conIdle, \\
\quad\quad {}^{C}connectionPhase(u?, v)\ eq\ {}^{M}conBothTeardown \mapsto {}^{M}conTermTeardown\rangle, \\
\quad\quad {}^{C}connectionPhase(u?, v))
\end{array}
$$

$$
\begin{array}{l}
\texttt{effect\_}{}^{m}connectTRelReq \\
\Delta({}^{C}connectionPhase, \\
\quad {}^{C}connectionResources, {}^{C}connectionDesignResources, {}^{C}userAvailable) \\
\qquad\qquad \left[\begin{array}{l}\text{Include the } \Delta \text{ list of} \\ \texttt{com\_}{}^{e}connectionDesignResourcesChange.\end{array}\right] \\
u? : {}^{m}userIds \\
\hline
{}^{C}connectionPhase'(u?, u?) = {}^{M}conIdle \\
\forall\, v : {}^{m}userIds \mid v \neq u? \bullet {}^{C}connectionPhase'(v, u?) = \\
\quad select_{d}(\\
\qquad \langle {}^{C}connectionPhase(v, u?)\ eq\ {}^{M}conEstablished \mapsto {}^{M}conOrigTeardown, \\
\qquad {}^{C}connectionPhase(v, u?)\ eq\ {}^{M}conTermTeardown \mapsto {}^{M}conIdle, \\
\qquad {}^{C}connectionPhase(v, u?)\ eq\ {}^{M}conBothTeardown \mapsto {}^{M}conOrigTeardown\rangle, \\
\qquad {}^{C}connectionPhase(v, u?))
\end{array}
$$

$$
\begin{array}{l}
\texttt{com\_}{}^{e}connectionDesignResourcesChange \\
\Delta({}^{C}connectionPhase, \\
\quad {}^{C}connectionResources, {}^{C}connectionDesignResources, {}^{C}userAvailable) \\
\qquad\qquad \left[\begin{array}{l}\text{Include the } \Delta \text{ list of} \\ \texttt{com\_}{}^{e}connectionDesignResourcesChange.\end{array}\right] \\
\hline
\forall\, u, v : {}^{m}userIds \bullet \\
\quad ifthen({}^{C}connectionResources(u, v)\ eq\ {}^{M}connectionResourceAvail\ and \\
\qquad\quad {}^{C}connectionResources'(u, v)\ neq\ {}^{M}connectionResourceAvail, \\
\qquad {}^{C}connectionPhase'(u, v)\ eq \\
\qquad\quad select_{d}(\\
\qquad\qquad \langle {}^{C}connectionPhase(u, v)\ eq\ {}^{M}conSetup \mapsto {}^{M}conOrigTeardown, \\
\qquad\qquad {}^{C}connectionPhase(u, v)\ eq\ {}^{M}conEstablished \mapsto {}^{M}conBothTeardown\rangle, \\
\qquad\qquad {}^{C}connectionPhase(u, v)), \\
\qquad {}^{C}connectionPhase'(u, v)\ eq\ {}^{C}connectionPhase(u, v))
\end{array}
$$

${}^{P}ConTeardownCond$ – **Connection Teardown Condition**

The auxiliary functions ${}^{f}conTeardown\ (u, v)$ describe whether user u perceives either the connection from u to v or from v to u to be somewhere in the teardown phase. The functions are used in particular for the definition of ${}^{c}connectRelInd(u, r)$. Even though these functions are boolean-valued, we do not denote them as modes, since they are not part of a meaningful mode class.

$^P ConTeardownCond$

inherit $^P ConFunConnectionPhase$

$^f conTeardown : {}^m userIds \times {}^m userIds \rightarrow \mathbb{B}$

$$\forall u, v : {}^m userIds \bullet$$
$$\quad {}^f conTeardown(u, v) \Leftrightarrow$$
$$\quad\quad (({}^C connectionPhase(u, v) \in \{{}^M conOrigTeardown, {}^M conBothTeardown\})$$
$$\quad\quad \vee ({}^C connectionPhase(v, u) \in \{{}^M conBothTeardown, {}^M conTermTeardown\}))$$

A.3.3 Fundamental Relevant Events of the Environment

When we define the behaviour of the environment, we need the same events, modes, and auxiliary functions as for the system, but with swapped decorations "!" and "?". Unfortunately, CSP-OZ does not yet offer the renaming operation [? ↔ !]. Therefore, we copy, paste, and rename manually the single property that is actually needed here. A minor language extension will solve this inconvenience in the future.

$^P ConChannelsEnv$

inherit $^P UserFun$

chan ${}^m connectReq : [\, u!, v! : {}^m userIds \,]$	[Monitored event: user u requests a connection to user v.]
chan ${}^c connectInd : [\, u : {}^m userIds \,]$	[Controlled event: user u is offered a connection.]
chan ${}^m connectRsp : [\, u! : {}^m userIds \,]$	[Monitored event: user u accepts an offered connection.]
chan ${}^c connectCnf : [\, u : {}^m userIds \,]$	[Controlled event: the previous connection request of user u has been accepted.]
chan ${}^m connectORelReq : [\, u! : {}^m userIds \,]$	[Monitored event: user u requests to release the connection / all connections which he has initiated.]
chan ${}^m connectTRelReq : [\, u! : {}^m userIds \,]$	[Monitored event: user u requests to release the connection / all connections to him which he has not initiated.]
chan ${}^c connectRelInd : [\, u : {}^m userIds;\ r : \mathbb{F}\, {}^t connectRelReasons \,]$	[Controlled event: user u is indicated that a connection is released because of reasons r.]

A.3.4 Fundamental Requirements for the Environment

$^P ConEnvFun$ – **Connection Environment**

$^{P}ConEnvFun$

inherit $^{P}ConChannelsEnv$

The environment will always take part in any controlled event of the system.

$$\begin{aligned} \mathtt{main} \;=\;& {}^{c}connectInd \rightarrow \mathtt{main} \\ & \square\; {}^{c}connectCnf \rightarrow \mathtt{main} \\ & \square\; {}^{c}connectRelInd \rightarrow \mathtt{main} \end{aligned}$$

A.3.5 Fundamental Requirements for the System

$^{P}ConFun$ – **Connection**

$^{P}ConFun$

inherit

$^{P}ConFunUserAvailable$
$^{P}ConFunConnectionDesignResources$
$^{P}ConFunConnectionResources$
$^{P}ConFunConnectionPhase$
$^{P}ConTeardownCond$

The system will always take part in any monitored event of the environment.

$$\begin{aligned} \mathtt{main} \;=\;& {}^{m}connectReq \rightarrow \mathtt{main} \\ & \square\; {}^{m}connectRsp \rightarrow \mathtt{main} \\ & \square\; {}^{m}connectORelReq \rightarrow \mathtt{main} \\ & \square\; {}^{m}connectTRelReq \rightarrow \mathtt{main} \end{aligned}$$

We model the state of controlled variables by the enabling conditions of controlled events. Accordingly, the controlled events can occur (many times) as long as their respective enabling conditions are true.

enable_ $^{c}connectInd$

$u : {}^{m}userIds$

$\exists\, v : {}^{m}userIds \bullet {}^{C}connectionPhase(v, u) \in \{{}^{M}conSetup, {}^{M}conEstablished\}$

effect_ $^{c}connectInd$

$u : {}^{m}userIds$

---

true

enable_ $^{c}connectCnf$

$u : {}^{m}userIds$

$\exists\, v : {}^{m}userIds \bullet {}^{C}connectionPhase(v, u) = {}^{M}conEstablished$

effect_ $^{c}connectCnf$

$u : {}^{m}userIds$

---

true

enable_ $^{c}connectRelInd$

$u : {}^{m}userIds$
$r : \mathbb{F}\, {}^{t}connectRelReasons$

$\exists\, v : {}^{m}userIds \bullet {}^{f}conTeardown(u, v)$

effect_ $^{c}connectRelInd$

$u : {}^{m}userIds$
$r : \mathbb{F}\, {}^{t}connectRelReasons$

---

true

A.3.6 Changeable Mode Classes and Auxiliary Functions

$^{P}UserResAvail$ – **Availability of User Resources**

These auxiliary functions provide two common terms for further definitions:

$^{P}UserResAvail$

inherit $^{P}ConFun$

Auxiliary function $^{f}userConIdle(u, v)$: are the connections from user u to user v and from v to u idle, from u's point of observation?
Auxiliary function $^{f}userConNotAvail(u, v)$: does there exist another user w who already uses the connection resources of user v for a connection?

$$^{f}userConIdle : {}^{m}userIds \times {}^{m}userIds \rightarrow \mathbb{B}$$
$$^{f}userConNotAvail : {}^{m}userIds \times {}^{m}userIds \rightarrow \mathbb{B}$$

$$\begin{array}{l}
\forall\, u, v : {}^{m}users \bullet \\
\quad {}^{f}userConIdle(u, v) \Leftrightarrow \\
\qquad (\ (\ \ ({}^{C}connectionPhase(u, v) = {}^{M}conIdle) \\
\qquad\quad \vee\ ({}^{C}connectionPhase(u, v) = {}^{M}conTermTeardown)) \\
\qquad \wedge\ (\ \ ({}^{C}connectionPhase(v, u) = {}^{M}conIdle) \\
\qquad\quad \vee\ ({}^{C}connectionPhase(v, u) = {}^{M}conOrigTeardown))) \\
\forall\, u, v : {}^{m}users;\ h_u, h_v : {}^{m}humans \bullet \exists\, w : {}^{m}users \bullet \\
\quad {}^{f}userConNotAvail(u, v) \Leftrightarrow \\
\qquad (\ (\qquad (u \in {}^{m}userRoles(h_u) \wedge v \in {}^{m}userRoles(h_v)) \\
\qquad\qquad \Rightarrow (h_u = h_v)) \\
\qquad \vee\ ((w \neq u) \wedge \neg\, {}^{f}userConIdle(v, w)))
\end{array}$$

$^{P}ConInitIdle$ – **Connections Start Idle**

$^{P}ConInitIdle$

inherit $^{P}ConFun$

Relations on the initial modes of $^{C}connectionPhase(u, v)$: all connections are idle in the beginning.

Init

$$\begin{array}{l}
\forall\, u, v : {}^{p}initialUsers \bullet \\
\qquad {}^{C}connectionPhase(u, v) = {}^{M}conIdle
\end{array}$$

$^{P}UserOnlyHuman$ – **Only Humans Play User Roles**

This property might be changed in order to introduce automated answering machines and similar services.

$^{P}UserOnlyHuman$

inherit $^{P}ConFun$

Relations on mode classes $^{C}connectionResources(u,v)$: a user role may be played by humans only.

$$\forall u, v : {}^{m}users \bullet$$
$$(\neg \exists h : {}^{m}humans \bullet u \in {}^{m}userRoles(h))$$
$$\Rightarrow ((^{C}connectionResources(u,v) \neq {}^{M}connectionResourceAvail)$$
$$\wedge (^{C}connectionResources(v,u) \neq {}^{M}connectionResourceAvail))$$

$^{P}ClosedNet$ – **No Off-Network Connections**

$^{P}ClosedNet$

inherit $^{P}ConFun$

Relations on mode classes $^{C}connectionResources(u,v)$: connections can be established with users of this network only.

$$\forall u : {}^{m}users;\ v : {}^{m}userIds \bullet$$
$$(v \notin {}^{m}users)$$
$$\Rightarrow (^{C}connectionResources(u,v) \neq {}^{M}connectionResourceAvail)$$

$^{P}DConResDesir$ – **Design Phase Connection Resources, Desirable**

A desired, but not required, additional behaviour is described by:

$^{P}DConResDesir$

inherit $^{P}ConFun$

Relations on mode classes $^{C}connectionDesignResources(u,v)$: it is preferred that many design resources are available.

$$\forall u, v : {}^{m}users \bullet$$
$$^{C}connectionDesignResources(u,v) = {}^{M}connectionDesignResourceAvail$$

Remark: Peters [Pet99] currently works on *merit functions* for Functional Documentation which would allow to award numerical relative merits to different desired behaviours.

$^{P}ConResSimple$ – **Simple Connection Resources**

$^{P}ConResSimple$

inherit $^{P}ConFun$

Relations on mode classes ${}^{C}connectionResources(u, v)$: the user role to connect to must be valid, the user must be currently available, and the resources provided by design decisions must be sufficent.

$$
\begin{array}{l}
\forall\, u : {}^{m}users;\; v : {}^{m}userIds \bullet \\
\qquad ({}^{C}connectionResources(u, v) = {}^{M}connectionResourceAvail) \\
\qquad \Leftrightarrow ((v \in {}^{m}users) \\
\qquad\quad \wedge\ ({}^{C}connectionDesignResources(u, v) \\
\qquad\qquad = {}^{M}connectionDesignResourceAvail) \\
\qquad\quad \wedge\ ({}^{C}userAvailable(u, v) = {}^{M}userAvail))
\end{array}
$$

${}^{P}UserResSimple$ – **Simple User Resources**

${}^{P}UserResSimple$

`inherit` ${}^{P}UserResAvail$

Relations on mode classes ${}^{C}userAvailable(u, v)$: a user v is not available for a user u if there exist another user w who already uses the resources of v for a connection.

$$
\begin{array}{l}
\forall\, u, v : {}^{m}users \bullet \\
\qquad ({}^{C}userAvailable(u, v) = {}^{M}userNotAvail) \Leftrightarrow {}^{f}userConNotAvail(u, v)
\end{array}
$$

A.3.7 Changeable Requirements for the Environment

${}^{P}ConEnvRandom$ – **Connection Environment Behaves Randomly**

${}^{P}ConEnvRandom$

`inherit` ${}^{P}ConChannelsEnv$

The environment may generate any monitored event at all times.

$$
\begin{array}{lcl}
\mathtt{main} & = & {}^{m}connectReq \to \mathtt{main} \\
& & \sqcap\ {}^{m}connectRsp \to \mathtt{main} \\
& & \sqcap\ {}^{m}connectORelReq \to \mathtt{main} \\
& & \sqcap\ {}^{m}connectTRelReq \to \mathtt{main}
\end{array}
$$

A.3.8 Changeable Requirements for the System

${}^{P}ConnectRelIndReasonUNotAvail$ – **Connection Release Reason Always Unavailability**

${}^{P}ConnectRelIndReasonUNotAvail$

`inherit` ${}^{P}ConFun$

When a connection release indication is issued, the reason given is always that the user is not available.
This operation adds further restrictions on the already specified version of it.

effect_ $^{c}connectRelInd$

$u : {}^{m}userIds$
$r : \mathbb{F}\ {}^{t}connectRelReasons$

$r = \{{}^{=}uNotAvail\}$

A.3.9 Collection of all the Requirements of this Feature

The class $^{P}BasicConnectionFeatureSysModes$ collects all properties concerning events, modes, and auxiliary functions of the system:

$^{P}BasicConnectionFeatureSysModes$

inherit

$^{P}ConChannels$	[fundamental]
$^{P}ConFunUserAvailable$	[fundamental]
$^{P}ConFunConnectionDesignResources$	[fundamental]
$^{P}ConFunConnectionResources$	[fundamental]
$^{P}ConFunConnectionPhase$	[fundamental]
$^{P}ConTeardownCond$	[fundamental]
$^{P}UserResAvail$	[changeable]
$^{P}ConInitIdle$	[changeable]
$^{P}UserOnlyHuman$	[changeable]
$^{P}ClosedNet$	[changeable]
$^{P}DConResDesir$	[changeable]
$^{P}ConResSimple$	[changeable]
$^{P}UserResSimple$	[changeable]

The class $^{P}BasicConnectionFeatureEnvModes$ collects all properties concerning events of the environment:

$^{P}BasicConnectionFeatureEnvModes$

inherit

$^{P}ConChannelsEnv$	[fundamental]

The class $^{P}BasicConnectionFeatureEnv$ collects all properties concerning the environment:

$^{P}BasicConnectionFeatureEnv$

inherit

$^{P}BasicConnectionFeatureEnvModes$	[(see above)]
$^{P}ConEnvFun$	[fundamental]
$^{P}ConEnvRandom$	[changeable]

The class $^{P}BasicConnectionFeatureSys$ collects all properties concerning the system:

$^{P}BasicConnectionFeatureSys$

inherit

$^{P}BasicConnectionFeatureSysModes$	[(see above)]
$^{P}ConFun$	[fundamental]
$^{P}ConnectRelIndReasonUNotAvail$	[changeable]

A.4 Composition of a Simple Complete Specification

Now we compose a simple telephone system specification. The class $^{P}SimpleSpecificationSys$ composes the system parts of all included features. The class $^{P}SimpleSpecificationEnv$ composes the environment parts.

$^{P}SimpleSpecificationSys$

inherit
$^{P}UserSpaceFeatureSys$
$^{P}BasicConnectionFeatureSys$

$^{P}SimpleSpecificationEnv$

inherit
$^{P}UserSpaceFeatureEnv$
$^{P}BasicConnectionFeatureEnv$

The class $^{P}SimpleSpecification$ composes the environment and the system of a simple telephone system specification. This is the only structure of the specification that we require to appear in any implementation, too (compare the discussion in Section 4.1).

$^{P}SimpleSpecification$

inheritchannels $^{P}SimpleSpecificationEnv$

$main \;=\; {}^{P}SimpleSpecificationEnv \parallel {}^{P}SimpleSpecificationSys$

List of Figures

List of Tables

Index of Defined Terms and Their Uses

The numbers in the index are page numbers where the terms are referenced, the bold number denotes the page of definition.

Sorted Alphabetically

Sorted by Kind

Bibliography

[AFHB$^+$92] Alspaugh, T. A., Faulk, S. R., Heninger Britton, K., Parker, R. A., Parnas, D. L., and Shore, J. E. *Software requirements for the A-7E aircraft*. Report NRL/FR/5530–92-9194, Naval Research Lab., Washington DC 20375·5000, USA (31 Aug. 1992). Revision of NRL Memorandum Report 3876.

[AhGr95] Aho, A. V. and Griffeth, N. D. *Feature interactions in the global information infrastructure*. In "Proceedings of the 1995 Conference on Foundations of Software Engineering" (1995).

[BDC$^+$89] Bowen, T. F., Dworack, F. S., Chow, C. H., Griffeth, N., Herman, G. E., and Lin, Y.-J. *The feature interaction problem in telecommunication systems*. In "Seventh IEEE International Conference on Software Engineering for Telecommunication Systems" (July 1989).

[BoVe94] Bouma, L. G. and Velthuijsen, H., editors. *Feature Interactions in Telecommunications Systems*. IOS Press, Amsterdam (1994).

[Bre98a] Bredereke, J. *Avoiding feature interactions in the users' interface*. In Kimbler and Bouma [KiBo98], pp. 305–317.

[Bre98b] Bredereke, J. *Requirements specification and design of a simplified telephone network by functional documentation*. CRL Report 367, McMaster University, Hamilton, Ontario, Canada (Dec. 1998).

[Bre99a] Bredereke, J. *Maintaining telephone switching software requirements*. IEEE Commun. Mag. (1999). *Submitted.*

[Bre99b] Bredereke, J. *Maintaining telephone switching system requirements*. In "Participants' notes of Dagstuhl seminar 99071 - software engineering research and education: seeking a new agenda" (15–19 Feb. 1999).

[Bre99c] Bredereke, J. *Modular, changeable requirements in functional documentation for telephone switching*. Formal Methods in System Design (1999). Special issue on Tabular Notation. *Submitted.*

[CaCo96] Cameron, E. J. and Cohen, B. *Tutorial on feature interactions*. In Gotzhein, R. and Bredereke, J., editors, "FORTE/PSTV '96 Tutorial Notes", pp. 53–108. Univ. of Kaiserslautern, Germany (Sept. 1996).

[CaVe93] Cameron, E. J. and Velthuijsen, H. *Feature interactions in telecommunications systems*. IEEE Commun. Mag. **31**(8), 18–23 (Aug. 1993).

[CCG+98] Cameron, J., Cheng, K., Gallagher, S., Lin, F. J., Russo, P., and Søbirk, D. *Next generation service creation: Process, methodology, and tool integration.* In Kimbler and Bouma [KiBo98], pp. 299–304.

[CGL+94] Cameron, E. J., Griffeth, N. D., Lin, Y.-J., Nilson, M. E., Schnure, W. K., and Velthuijsen, H. *A feature interaction benchmark in IN and beyond.* In Bouma and Velthuijsen [BoVe94], pp. 1–23.

[ChOh95] Cheng, K. E. and Ohta, T., editors. *Feature Interactions in Telecommunications III.* IOS Press, Amsterdam (1995).

[CHW98] Coplien, J., Hoffman, D., and Weiss, D. *Commonality and variability in software engineering.* IEEE Software pp. 37–45 (Nov./Dec. 1998).

[CuWe98] Cuka, D. A. and Weiss, D. M. *Engineering domains: Executable commands as an example.* In Devanbu, P. and Poulin, J., editors, "5th Int'l Conf. on Software Reuse", pp. 26–34, Los Alamitos, CA, USA (2–5 June 1998). IEEE Computer Society.

[DBL97] Dini, P., Boutaba, R., and Logrippo, L., editors. *Feature Interactions in Telecommunication Networks IV.* IOS Press, Amsterdam (June 1997).

[DRS95] Duke, R., Rose, G., and Smith, G. *Object-Z: A specification language advocated for the description of standards.* Computer Standards & Interfaces **17**, 511–533 (1995).

[DuVi92] Duran, J. M. and Visser, J. *International standards for Intelligent Networks.* IEEE Commun. Mag. **30**(2), 34–42 (Feb. 1992).

[FæOl94] Færgemand, O. and Olsen, A. *Introduction to SDL-92.* Comp. Networks and ISDN Syst. **26**, 1143–1167 (1994).

[FBWK92] Faulk, S., Brackett, J., Ward, P., and Kirby, Jr., J. *The Core method for real-time requirements.* IEEE Software **9**(6), 22–33 (Sept. 1992).

[Fis97a] Fischer, C. *Combining Object-Z and CSP.* In Wolisz, A., Schieferdecker, I., and Rennoch, A., editors, "Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch", GMD-Studien Nr. 315, pp. 119–128. GMD, St. Augustin, Germany, Berlin, Germany (19–20 June 1997).

[Fis97b] Fischer, C. *CSP-OZ: a combination of Object-Z and CSP.* In Bowman, H. and Derrick, J., editors, "Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)", vol. 2, pp. 423–438. Chapman & Hall (July 1997).

[FiWe99] Fischer, C. and Wehrheim, H. *Model-checking CSP-OZ specifications with FDR.* In Araki, K., Galloway, A., and Taguchi, K., editors, "Proc. of the 1st Int'l. Conf. on Integrated Formal Methods – IFM'99". Springer (1999).

[GJKW97] Gupta, N. K., Jagadeesan, L. J., Koutsofios, E. E., and Weiss, D. M. *Auditdraw: Generating audits the FAST way.* In "3rd IEEE Int'l Symposium on Requirements Engineering", pp. 188–197, Annapolis, MD, USA (6–10 Jan. 1997). IEEE Computer Society Press.

[HJKL$^+$98] Heitmeyer, C., James Kirby, J., Labaw, B., Archer, M., and Bharadwaj, R. *Using abstraction and model checking to detect safety violations in requirements specifications*. IEEE Trans. Softw. Eng. **24**(11) (Nov. 1998).

[HJL96] Heitmeyer, C. L., Jeffords, R. D., and Labaw, B. G. *Automated consistency checking of requirements specifications*. IEEE Trans. on Software Engineering and Methodology **5**(3), 231–261 (July 1996).

[Hoa85] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice-Hall (1985).

[IEE92] IEEE Communications Society. *1st International Workshop on Feature Interactions in Telecommunications Software Systems*, St. Petersburg, Florida, USA (Dec. 1992).

[ITU93] ITU-T. *Q.121x-Series Intelligent Network Recommendations for Capability Set 1* (1993).

[ITU94] ITU-T, Recommendation Z.100 (03/93). *CCITT Specification and Description Language (SDL)* (1994).

[ITU97a] ITU-T. *Q.122x-Series Intelligent Network Recommendations for Capability Set 2* (1997).

[ITU97b] ITU-T. *Q.12xx-Series Intelligent Network Recommendations* (1997).

[Kat93] Katz, S. *A superimposition control construct for distributed systems*. ACM Trans. Prog. Lang. Syst. **15**(2), 337–356 (Apr. 1993).

[KiBo98] Kimbler, K. and Bouma, L. G., editors. *Feature Interactions in Telecommunications and Software Systems V*. IOS Press, Amsterdam (Sept. 1998).

[KiVe95] Kimbler, K. and Velthuijsen, H. *Feature interaction benchmark*. Discussion paper for the panel on benchmarking at FIW'95, Kyoto, Japan (Oct. 1995).

[Mil98] Miller, S. P. *Specifying the mode logic of a flight guidance system in CoRE and SCR*. In "Second Workshop on Formal Methods in Software Practice", Clearwater Beach, Florida, USA (4–5 Mar. 1998).

[MSRP98] Margaria, T., Steffen, B., Rückert, R., and Posegga, J., editors. *Services And Visualization : Towards User-Friendly Design ; Selected Papers ACoS'98, VISUAL'98, AIN'97*, no. 1385 in Lecture Notes in Computer Science. Springer (1998).

[MTJ93] Mierop, J., Tax, S., and Janmaat, R. *Service interaction in an object-oriented environment*. IEEE Commun. Mag. **31**(8), 46–51 (Aug. 1993).

[PaMa95] Parnas, D. L. and Madey, J. *Functional documents for computer systems*. Sci. Comput. Programming **25**(1), 41–61 (Oct. 1995).

[Par72] Parnas, D. L. *On the criteria to be used in decomposing systems into modules*. Commun. ACM **15**(12), 1053–1058 (1972).

[Par76] Parnas, D. L. *On the design and development of program families*. IEEE Trans. Softw. Eng. **2**(1), 1–9 (Mar. 1976).

[Par79] Parnas, D. L. *Designing software for ease of extension and contraction*. IEEE Trans. Softw. Eng. **SE-5**(2), 128–138 (Mar. 1979).

[Pet99] Peters, D. K. *Deriving Real-Time Monitors from System Requirements Documentation*. PhD thesis, McMaster Univ., Hamilton, Canada (1999). *To appear.*

[PlRy98] Plath, M. and Ryan, M. *Plug and play features*. In Saake and Türker [SaTü98], pp. 78–105. URL: http://wwwiti.cs.uni-magdeburg.de/iti_db/veroeffentlichungen/98/SaaTue98.html.

[Ros94] Roscoe, A. W. *Model-checking CSP*. In Roscoe, A. W., editor, “A Classical Mind, Essays in Honour of C.A.R. Hoare”, pp. 353–378. Prentice-Hall (1994).

[Ros97] Roscoe, A. W. *The Theory and Practice of Concurrency*. Prentice-Hall (1997).

[SaTü98] Saake, G. and Türker, C., editors. *Proc. of the 1st FIREworks Workshop – FIREworks '98*, no. 10 in Preprint-Reihe, Magdeburg, Germany (15–16 May 1998). Fakultät für Informatik, Universität Magdeburg. URL: http://wwwiti.cs.uni-magdeburg.de/iti_db/veroeffentlichungen/98/SaaTue98.html.

[Smi92] Smith, G. P. *An Object-Oriented Approach to Formal Specification*. PhD thesis, Univ. of Queensland, Australia (Oct. 1992).

[Spi95] Spivey, J. M. *The Z notation: a reference manual*. Series in Computer Science. Prentice-Hall, New York, 2nd ed. (1995).

[vG99] von Garrel, J. *Parsing, Typechecking und Transformation von CSP-OZ nach Jass*. Masters thesis, Univ. of Oldenburg, Dept. of Comp. Sci. (July 1999).

[vS92] van Schouwen, A. J. *The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems*. CRL Report 242, McMaster University, TRIO, Hamilton, Canada (Feb. 1992).

[vSPM93] van Schouwen, A. J., Parnas, D. L., and Madey, J. *Documentation of requirements for computer systems*. In “IEEE Int'l. Symposium on Requirements Engineering – RE'93”, pp. 198–207, San Diego, Calif., USA (4–6 Jan. 1993). IEEE Comp. Soc. Press.

[Wir99] Wirtz, B. *Übersetzung des Daten- und Verhalten-integrierenden Spezifikationsformalismus CSP-OZ nach CSP*. Masters thesis, Univ. of Oldenburg, Dept. of Comp. Sci. (1999). *To appear.*

[WoDa96] Woodcock, J. and Davies, J. *Using Z – Specification, Refinement, and Proof*. Prentice-Hall (1996).

[Zav97] Zave, P. *'Calls considered harmful' and other observations: A tutorial on telephony*. In Margaria, T., editor, “2nd International Workshop on Advanced Intelligent Networks '97 – AIN'97, Proceedings”, Tech. Rep. MIP-9710, pp. 1–20. Univ. of Passau, Germany (4–5 July 1997). Also in [MSRP98].