Requirements Specification and Design of a Simplified Telephone Network by Functional Documentation

Jan Bredereke

MacMaster University, CRL
1280 Main Street West
Hamilton, Ontario L8S 4K1
Canada

email: jan.bredereke@topmail.de
URL: http://www.poboxes.com/jan.bredereke

CRL Report 367
Dec. 1998

Abstract

Feature interaction problems are currently a major roadblock to extending and changing telephone switching systems, which have to follow market needs quickly. Such problems appear already in the requirements specifications. We avoid certain kinds of feature interactions by employing a more modular specification structure. First, we show how the details of the users' interface, like buttons and ring tones, can be encapsulated and separated from the system's functionality. Such an architecture could be realized even within the structure of the current standard for the Intelligent Network (IN). Second, we argue that the requirements for the Plain Old Telephone Service (POTS) comprehend several different concerns, and that separate requirement concerns should be separated in the specification. We employ the Functional Documentation approach [PaMa95, vSPM93, vS92a] and investigate how it can be extended to group requirement concerns, and to cover not only one but a sequence of development contracts. Our approach is related to the standard refinement approach for software development. But when we refine a specification to introduce new services, we order the refinement steps (i.e., the new required properties) by the likeliness by which they must be taken back, and we document the dependences between these steps explicitly. In the appendix, we provide a case study that applies our ideas.

Contents

Chapter 1

Introduction

Feature interactions in telecommunication systems increasingly obstruct the introduction of new features. They are an enstablished research subject by now [KiBo98, DBL97, ChOh95, BoVe94]. The causes for feature interactions are many, therefore many remedies will have to be applied together. A substantial number of the causes can be classified as related to the *maintenance* of telephone switching software. These software systems rank among the largest in the world, and due to their success they have a long history of modifications and enhancements.

As a general rule, in order to support the maintenance task, we advocate the use of precise documentation of the requirements and the design, both for a telephone switching system and for its add-on features. More specifically, we aim at avoiding certain kinds of feature interaction by employing a more modular specification structure. Modularization aims at limiting the scope of changes to one module only, thus reducing the amount of work and the likeliness of mistakes. In this report, we investigate two particular approaches enhancing modularization.

In Chapter 2, we discuss the typical current structure of requirements documents with respect to the users' interface. The users' interface can comprehend quite different hardware, and in particular it can comprehend hardware that has only few syntactic signals available (twelve buttons, a hook switch, and a few signal tones). The current set of available services, when taken together, requires a large number of signals, much more than are available physically on most telephone devices. Physical signals have to be reused in different modes of operation. Unfortunately, there is not yet any systematic coordination among the services about the use of the physical signals. On the contrary, the definitions of several services implicitly assume exclusive access to the user's telephone device. Of course, when this assuption is violated, undesired feature interactions can, and frequently do, occur. The concern of the users' interface is spread out far through the requirements documents both for the basic service and for the additional services. We find that the information hiding principle is not yet applied satisfactory. We propose a more suitable structure for modelling the users' interface, and we relate it to the architecture of the current standard for the Intelligent Network (IN) [ITU93b, ITU97, DuVi92]. An earlier version of this chapter appeared in [Bre98].

In Chapter 3, we extend our view to the requirements on the switching system's behaviour in general, and we propose a way to structure the requirements document that facilitates its maintenance. The requirements for current telephone switching systems are specified rather monolithic. This manifests itself, e.g., in the notion of *POTS* (Plain Old Telephone Service) which is used when it comes to adding new services to these systems. The requirements for new services are usually defined relative to POTS (if they don't specifically extend another add-on service, which in turn is defined relative to POTS). POTS is assumed to be well-known and well understood. An example is the current standard for the Intelligent Network (IN). It defines an extension interface for new services that can be added to POTS, where POTS essentially can be provided by any of the currently existing switching systems. The so-called *detection points* are distributed along a kind of finite state automaton (or rather, a pair of automata) that describes the set-up and tear-down of a conventional call. This architecture defines its major interface essentially along a line that is justified by history only: the capabilities of the currently existing switching software systems. POTS includes quite different concerns: voice connection establishment, users' interface, billing, routing, etc. The consequence is that new services are attached frequently to the same (few) detection points even though they have quite different concerns. This leads to several undesired feature interactions. (For example when the order of activating the features is relevant. Think of Abbreviated Dialling, which translates numbers, and Originating Call Screening, which should block calls to certain numbers.) We argue that the requirements for POTS should be separated into different concerns, and that this should lead to a redefinition of the interfaces at which new services may be added.

The requirements specification of a system prescribes what can be observed at the interface to its environment. Every requirements specification should not only say what the system is supposed to do, but it should also say explicitly on what the system can rely on, and what the environment must provide. These two aspects must be distinguished precisely, in order that the responsibilities of the software developer and his customer are clear. This is also true for the requirment specifications of system extensions. The environment for an extension is mostly the system already existing. But we argue that it is not wise to just take the entire current system as prerequisite. If one does this, every feature invalidates the prerequisites for all the other features, since only one feature may rely on an unmodified base system. In such an approach, feature interactions become likely. Therefore, the requirements document for the existing system should be modular, and any extension should name as few as possible of these modules as prerequisites. This reduces the likeliness for interactions. Furthermore, if the "real" dependences are made explicit, it becomes possible to check them automatically by syntactic means only.

In the appendix, we provide a case study that specifies the requirements for a telephone switching system from the users' point of view. Since specifying an entire telephone switching system is way beyond the means of our research project, the case study concentrates on a core topic for telephone switching: it provides a "call" concept that can be extended into different directions. In our case study, we take care that we include *explicitly* all concepts that are relevant for the scope of the case study. For example, we distinguish between a telephone device, a human, and a human's social roles. POTS did not make these distinctions. Therefore, we cannot distinguish between a POTS directory number denoting the service hotline of a company, and a POTS directory number denoting a specific employee of a manager. As soon as we try to assign another employee to the hotline, for example by a forwarding feature, we may run into problems if another feature requires access to another role of this human. For example, if the employee forwards *all* connections to a colleague, then his boss will not be able to return an earlier connection attempt from him. When we distinguish between the role of the service hotline, and the role of an in-house employee, there arises no conceptual problem anymore.

Similarly, we add explicitly the concept of call screening to our requirements document first, in the form of a new service that just introduces the necessary extension interface, and only afterwards we add some screening features.

In Chapter 4, we summarize our results and discuss possible future work.

Chapter 2

Avoiding Feature Interactions When Modelling the Users' Interface

2.1 Introduction

In our general approach, we aim at avoiding a certain kind of feature interaction by employing a more modular structure for the design of a telephone switching system and its features. In this chapter, we investigate the architecture of the users' interface of such a system.

We propose to separate the concern of the users' interface more clearly from the concern of the behaviour of features, in particular when new features are introduced into a telephone switching system. Of course, this helps to handle complexity by a divide-and-conquer technique, but in particular it enables us to validate formally the absence of a certain kind of feature interaction in a small part of the entire system, while we avoid this kind of feature interaction entirely in the remaining parts.

As motivation, we present one example for a feature interaction that is caused by the way the users' interface often is expressed. It can occur between a *Credit Card Calling* (CCC) feature and a *Voice Mail* feature. The example is taken from the feature interaction benchmark by Cameron et. al. [CGL+94]. Credit Card Calling 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 his own, a subscriber of some voice mail service can call his own phone number (maybe using the CCC feature) and then press the "#" button followed by some identification to indicate that he is the subscriber. These features can't work together properly since the telephone system has no way to determine whether pressing the "#" button means that the caller just doesn't want to leave a message, or whether he is the subscriber.[1]

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

The problem in the example is that two features are active concurrently, but they are defined in a way that assumes that each of them is the only feature communicating via the associated user interface. There is no general provision that controls the access to this resource by different features. Today, hundreds of features have been conceived, with many of them requiring user interaction. Nevertheless, many features could be used concurrently and independently, if the users' interface resource, and in particular, the signals that a user can generate (or receive) on a standard phone, were not so limited.

There are some more examples of feature interactions in the benchmark [CGL+94] that are caused in this way: *Call Waiting* and *Three-Way Calling* (ambiguous flash hook), *Multi-location Business Service-Extension Dialing* and *CENTREX* (ambiguous numbering plan), *Call Waiting (CW)* and *Personal Communication Services (PCS)* (definition of CW for PCS user X ignores that a phone may be shared with PCS user Y not subscribing to CW), *Call Waiting* and *Call Waiting* (a user may take part in two concurrent CW contexts; hanging up can be ambiguous with respect to ringing back for held parties, if both contexts are in different modes), and *Call Waiting* and *Three-Way Calling* (ambiguous hang up in a complex context).

Still, features are often described as if each of them had exclusive control of the users' interface. Notably, the Feature Interaction Detection Contest instructions [GBGO98] take exactly this approach. They specify features in terms of "Flash", "Off-hook", "DialTone", "StartRinging" etc. Many of the case study specifications in the proceedings of previous feature interaction workshops [DBL97, ChOh95, BoVe94] are expressed similarly. Even though they aggregate individual digits to a "dialled number", in general these requirement specifications of features make assumptions about details of the users' interface hardware.

There are some exceptions in the literature, for example Blom et al. [BBK95] assume a "parser" that receives signals from the outside (e.g., flash hook), and delivers abstract events, such as "Transfer", "Switch", and "3Way", depending on the state of the system. Other exceptions are Gibson [Gib97] who recommends to distinguish clearly between abstract actions/signals in the requirements model and concrete actions/signals in the implementation model, Bruns et. al. [BMS98] who also distinguish abstract and concrete events, and Zibman et. al. [ZWO+95] who propose to use "logical names" instead of "specific signals". Even though Cattrall et. al. [CHJB95] do not address low-level signals, they work with abstract signals entirely. Griffeth and Velthuijsen [GrVe94] attempt to derive the intentions behind low-level signals by a goal profile stored in an automated negotiating agent.

The Intelligent Network (IN) standard [DuVi92, ITU93b] is intended to facilitate the introduction of new features. Nevertheless, it prescribes only partially how user interactions with these features are processed, and thus there is no general, coordinated way for user interaction processing beyond the provisions for basic call processing. Partially, this is alleviated by imposing restrictions on a concurrent activation of features, but this prevents many useful features from being specified. These restrictions have been reduced in the IN capability set 2 (CS-2) [ITU97], but in the IN CS-2 architecture there is still no mandatory synchronization protocol among the features concerning user interaction.

In Section 2.2, we introduce to our way of formal requirements specification, and to how this can lead to a formal description of a design. Using this, we sketch in Section 2.3 how most of the requirements and the design of a feature can be specified precisely, without refering to any hardware details, and how these details can be encapsulated. In Section 2.4, we apply this to the Intelligent Network architecture. In Section 2.5, we show that the approach allows validation of the encapsulated users' interface for the absence of a certain kind of feature interaction. Section 2.7 summarizes the results and gives an outlook.

2.2 Specifying the Requirements and the Design

2.2.1 Formal Distinction of the Pre-Existing Environment

Usually, we are not in a position to design the requirements for an entirely new telephone switching system. For example, we might not have a choice of the hardware that the users will use. Similarly, we might be required to use pre-existing software, in particular some telephone switching system with a software interface conforming to the Intelligent Network recommendations [ITU93b, ITU97]. Nevertheless, we *can* express the requirements on any part of the system, in particular, on new telephony features, on the semantic level.

The requirements document must contain additionally a description of the relevant aspects of the pre-existing parts. The implementer should realize the semantic-level specification by making use of these pre-existing means. In the requirements document, the assumed environment should be distinguished clearly from the new system to construct, such that the document is a self-contained, complete description. We are interested, in particular, in formal requirements documents. They avoid ambiguities and allow for validation (compare Section 2.5). Therefore, the distinction between the system to construct and the environment to assume should be formal, too.

Please note that the boundary between the system to construct and its environment usually is *not* the users' interface; these are two completely different kinds of boundaries, which should not be confused.

2.2.2 Formal Specification Technique "Functional Documentation"

In addition to identifying all environmental entities that are of relevance, and distinguishing them, a formal requirements specification must indicate whether the system to construct can control these environmental entities or not. For example, the user's decision to release and reconnect cannot be controlled, and neither can the triggering of Detection Points (DPs) by the underlying basic call processing software. But the system can control the arming of a DP, and it can control procedure calls to the underlying software.

Any entities that are only to be constructed are *not* part of the formal requirements specification. They are part of the design. For example, the requirements specification must not prescribe any local state components, e.g., Personal Identification Number (PIN) lists, if these entities do not exist before the system is constructed. The required behaviour of a system to construct should be stated only in terms of the history of its environment's behaviour.

Functional Documentation [PaMa95, vSPM93, vS92b] is a specification technique that allows formal specification of the requirements of a system in terms of environmental entities only. Furthermore it distinguishes them clearly into *controlled variables* and *monitored variables*, denoted by two vectors of time-functions $\underset{\sim}{c}^t$ and $\underset{\sim}{m}^t$, respectively. The environment to be assumed is specified as a relation *NAT*. The relation NAT describes how the relevant part of the world may behave over time without or despite the system to construct:

- $\mathsf{domain}(\mathrm{NAT})$ exactly contains the instances of $\underset{\sim}{m}^t$ allowed by the environmental constraints, and
- $\mathsf{range}(\mathrm{NAT})$ exactly contains the instances of $\underset{\sim}{c}^t$ allowed by the environmental constraints.
- $(\underset{\sim}{m}^t, \underset{\sim}{c}^t) \in \mathrm{NAT}$ if and only if the environmental constraints allow the controlled quantities to take on the values described by $\underset{\sim}{c}^t$, if the values of the monitored quantities are described by $\underset{\sim}{m}^t$.

The world, as it may behave with the system present, is described by the intersection $\mathrm{NAT} \cap \mathrm{REQ}$. The relation *REQ* may restrict the behaviour of controlled variables only, but not of monitored variables:

- $\mathsf{domain}(\mathrm{REQ})$ exactly contains the instances of $\underset{\sim}{m}^t$ allowed by the environmental constraints, and
- $\mathsf{range}(\mathrm{REQ})$ exactly contains the instances of $\underset{\sim}{c}^t$ allowed by a correct system.
- $(\underset{\sim}{m}^t, \underset{\sim}{c}^t) \in \mathrm{REQ}$ if and only if the system should permit the controlled quantities to take on the values described by $\underset{\sim}{c}^t$ when the values of the monitored quantities are described by $\underset{\sim}{m}^t$.

2.2.3 From Requirements to Design

The Four Variable Model

In the design phase, we make decisions about how the requirements are to be implemented. the *Four Variable Model* [PaMa95] splits the system into an input part, an output part, and a software part, described by the three relations IN, OUT, and SOF, respectively. The approach describes the boundary between IN and SOF by a vector of *input variables*, and the boundary between SOF and OUT by a vector of *output variables*. IN is a relation between the monitored and the input variables, and OUT is a relation between the output and the controlled variables. Thus there is a connection to the requirements specification, and the correctness of the design can be checked formally. When using relational composition, $(\mathrm{NAT} \cap (\mathrm{IN} \cdot \mathrm{SOF} \cdot \mathrm{OUT})) \subseteq \mathrm{REQ}$ must hold. We can also express this kind of composition of the components by a logical "and", if we add a suitable quantification over the variables [PaMa95].

Generalization of the Four Variable Model w.r.t. Variables

In our application domain, we feel the need to decompose a system into more and different components. As we will see below, we prefer to keep input and output together, thus joining the relations IN and OUT, and joining the vectors of input and output variables. On the other hand, we need more than one such component.

Therefore, we use one vector of design variables, instead of the two vectors of input and output variables. Furthermore, we allow an arbitrary number of relations that describe components of the design. These components are composed by a kind of logical "and", again, but in the form of an intersection operator.

In the following, some of our relations will additionally refer to a vector of time-functions $\underset{\sim}{v}^t$ introduced during design. Thus, relations become ternary, with REQ and NAT having an empty vector of design time-functions $\underset{\sim}{v}^t$ as a special case. Furthermore, we will employ relations over subvectors of $\underset{\sim}{m}^t$, $\underset{\sim}{c}^t$, and $\underset{\sim}{v}^t$, and we will compose these relations. The next section introduces these concepts formally.

Formalizing the Variable-Extended Composition

In the definition of NAT in Section 2.2.2, $\mathsf{domain}(NAT)$ is the projection $\pi_1(\mathrm{NAT})$ of the relation NAT to the first element position of the relation pairs $(\underset{\sim}{m}^t, \underset{\sim}{c}^t)$, and $\mathsf{range}(NAT)$ is the projection $\pi_2(\mathrm{NAT})$ of the relation NAT to the second element position.

For a ternary relation R, we define $\mathsf{mon}(R) \stackrel{\mathrm{def}}{=} \pi_1(R)$, $\mathsf{ctrl}(R) \stackrel{\mathrm{def}}{=} \pi_2(R)$, and $\mathsf{des}(R) \stackrel{\mathrm{def}}{=} \pi_3(R)$. We can regard the projection $\mathsf{mon}(R)$ as a generalization of $\mathsf{domain}(R)$ and the projection $\mathsf{ctrl}(R)$ as a generalisation of $\mathsf{range}(R)$. The projection $\mathsf{des}(R)$ is the additional third design time-function vector. We assume that each of the monitored, controlled, and design time-functions (i.e., variables) has a unique name, and we label all the positions of all the time-function vectors with the respective names. We assume some total ordering over the names of time-functions, and further we assume without loss of generality that all vectors of time-functions have their components sorted.

When $r = (\underset{\sim}{m}^t, \underset{\sim}{c}^t, \underset{\sim}{v}^t)$ is a triple of labelled time-function vectors, and R is a ternary relation on labelled time-function vectors, and when R contains no other time-function names than those of the vector triple r (but maybe less), then we denote the *restriction* of r to the time-functions whose names appear in R as $r{\downarrow_{\mathrm{R}}}$. (In the preceding, we treat each of the three vectors separately, a labelled time function may appear in more than one vector, for example in the case where a variable is both monitored and controlled.)

For example, be $f_1^t, \ldots, f_{10}^t$ time functions ranging over the boolean values, and be

$$r = \left(\left\langle \overset{\overset{\textsf{button1}}{\downarrow}}{f_1^t}, \overset{\overset{\textsf{button2}}{\downarrow}}{f_2^t}, \overset{\overset{\textsf{hook}}{\downarrow}}{f_3^t} \right\rangle, \left\langle \overset{\overset{\textsf{hook}}{\downarrow}}{f_3^t}, \overset{\overset{\textsf{ring}}{\downarrow}}{f_4^t} \right\rangle, \left\langle \right\rangle \right)$$

and be

$$R = \left\{ \left(\left\langle \overset{\overset{\textsf{button1}}{\downarrow}}{f_5^t}, \overset{\overset{\textsf{button2}}{\downarrow}}{f_6^t} \right\rangle, \left\langle \overset{\overset{\textsf{hook}}{\downarrow}}{f_7^t} \right\rangle, \left\langle \right\rangle \right), \left(\left\langle \overset{\overset{\textsf{button1}}{\downarrow}}{f_8^t}, \overset{\overset{\textsf{button2}}{\downarrow}}{f_9^t} \right\rangle, \left\langle \overset{\overset{\textsf{hook}}{\downarrow}}{f_{10}^t} \right\rangle, \left\langle \right\rangle \right) \right\}$$

and be the time-function name ordering a lexicographical one. Then, the restriction of r to the time-functions of R is

$$r{\downarrow}_{\mathrm{R}} = \left(\left\langle \overset{\overset{\textsf{button1}}{\downarrow}}{f_1^t}, \overset{\overset{\textsf{button2}}{\downarrow}}{f_2^t} \right\rangle, \left\langle \overset{\overset{\textsf{hook}}{\downarrow}}{f_3^t} \right\rangle, \left\langle \right\rangle \right)$$

For two relations R, S, we define the *universal combined relation* $\Theta_{R,S}$ as the relation that contains exactly those labelled time-functions in $\mathsf{mon}(\Theta_{R,S})$, $\mathsf{ctrl}(\Theta_{R,S})$, and $\mathsf{des}(\Theta_{R,S})$ that appear in at least one of R or S in the respective vectors, with $\Theta_{R,S}$ being the largest relation doing so. This means that $\Theta_{R,S}$ is defined over all time-functions in question, and that it combines any allowed values of monitored variables with any allowed values of controlled variables and any allowed values of design variables. This is useful (only) as a starting point for later restrictions.

Finally, we can define the *variable-extended composition* of relations:

$$R \mathbin{\hat{\cap}} S \overset{\text{def}}{=} \{ r \mid r \in \Theta_{R,S} \wedge r{\downarrow}_{\mathrm{R}} \in R \wedge r{\downarrow}_{\mathrm{S}} \in S \}$$

The result is a relation that is defined over all variables in question, like $\Theta_{R,S}$ above, but which allows only that behaviour to which *both* relations R and S agree. If one of the relations is not defined for some variable, then this aspect of the resulting relation is determined only by the other relation.[2]

2.3 Encapsulating the Users' Interface

2.3.1 Specifying Feature Requirements on the Semantic Level

A human-computer interface allows a human and a machine to interact; for this, they must use a language. As in other areas concerned with languages, one can distinguish the

[2] This informal explanation shows that the variable-extended composition of relations is related to the *demonic-meet* operator introduced by Khédri and others [DBBMS$^+$95, DTK94], which takes two relations as arguments and renders a relation describing all the behaviour to which none of the argument relations disagree.

syntax and the semantics. The model of Foley and Wallace [FvDFH90, FoWa74] (compare also [Shn97]) is often used for (graphical) human-computer interfaces. They distinguish two major components: the form and the meaning. For their kinds of interfaces, they continue and differentiate more levels. For the purposes of this chapter, we only need the distinction between the specific form of an interaction, i.e., the syntax, and its meaning, i.e., the semantics.

The lessons learned with the human-computer interface can and should be applied to the user-telephone interface, too. We propose to specify the requirements for a telephone switching system, including any additional features, on the semantic level. For the moment, we assume that we are interested in the requirements of an entire such system, and not only a part of it. If we specify the behaviour of the system solely on the semantic level, then all the feature interaction problems listed in Section 2.1 simply disappear from the requirements document, since they have become a design problem.

In the example of Credit Card Calling (CCC) and Voice Mail (VM), we now have separate sets of semantic signals from the user to the system for each of the features. The CCC feature adds something like a *ReleaseAndReconnect* semantic signal to the system, and the VM feature adds something like a *VMlogin* semantic signal. If the functionality of two features is not related, they should always have separate sets of semantic signals.

We propose that a semantic signal should reflect (a) a *user's decision* to perform some action, for example to request a communication connection to some other user, (b) or a *user's perception* that some other user (or the system) has decided to perform some action, for example to request a communication connection with him.

But this move has not yet solved the problem. Eventually, all of the semantic signals must still be mapped onto the few syntactic signals, and back. The exact mapping is a concern of the users' interface, and it has little to do with the functionality of the telephone switching system and its features. According to the well known information-hiding principle (e.g., [Par72]), it should therefore be encapsulated as a secret of only one design module, which we call the *users' interface module* (UIF). It will still be difficult to find a satisfactory mapping. But if all aspects of this problem are separated and concentrated into a single module, it will be easier to understand its semantics, and thus to conceive a solution, and the module will be easier to change later. The latter is of particular importance to the feature interaction problem. Furthermore, it will be easier to validate the correctness of a solution, as we will see in Section 2.5.

We have to stress that an information-hiding module such as the UIF module is *not* necessarily implemented by some centralized software program or in a single hardware component [Par72]. The different access functions, for example the mapping routines, may be distributed over many software packages and to many physical locations. But the module must be *maintained* as a single unit, with any changes distributed in a consistent manner. We will discuss this issue further in Section 2.4.

2.3.2 Explicit User Manual Relation MAN in Design

Now, we discuss how the semantic signals can be related to the pre-existing entities of the users' interface. For the moment, we assume that we design a single feature only, with no other features already in the environment. In Section 2.3.3 below, we remove this restriction and discuss the general case of many features.

Figure 2.1(a) shows the components of the requirements document. We group the variables (both monitored and controlled) into three vector triples, as shown. The first, the vector triple of *semantic variables*, describes the semantic signals by the users. Furthermore, we split the description of the relevant environment into two relations. The first relation *USR* describes what the users might do on the semantic variables (USR is strongly non-deterministic since many aspects of user behaviour are not relevant for the system). The second relation *IOHARD* describes some pre-existing input/output hardware (and maybe software), by relating a vector triple of *physical variables* to a vector triple of *software variables*. The requirements of the feature are described by the relation REQ. IOHARD and the low-level physical and software variables have not yet any connection to REQ. We deviate from the standard approach insofar as we define REQ only over the semantic variables, not over all variables mentioned. Nevertheless, (USR $\hat{\cap}$ IOHARD) $\hat{\cap}$ REQ still describes the allowed behaviour of the system, using the definition of composition from the previous section. The entire environment NAT can be derived from the composition of USR and IOHARD:

$$\mathrm{NAT} \stackrel{\mathrm{def}}{=} \mathrm{USR} \mathbin{\hat{\cap}} \mathrm{IOHARD}$$

In Figure 2.1(b), we propose a *design structure* for these requirements that encapsulates the users' interface. There are two important aspects in this design. First, we introduce a software reflection of the semantic variables (which cannot be observed directly by any machine). These design variables, the *semantic software variables*, separate the software into an input/output driver part *IODRV* and another version of the feature's requirements *FEATREQ*. FEATREQ is usually very similar or identical to REQ, modulo variable renaming; the difference is that the semantic software variables can be observed by the software, while the semantic variables in the users' minds cannot.

The second important aspect is how the semantic variables in the users' minds are related to the physical variables in the design. We propose to describe this relation by an explicit relation *MAN*. It documents explicitly how the users translate their decisions on the semantic level into physical actions, and how physical actions translate to users' perceptions. These translations do not exist before the system is designed, therefore MAN is part of the design. The system designer conceives these translation rules, and usually he documents them in a user manual (hence the name MAN).

For the validation purposes in Section 2.5, we need to write down this relation explicitly and formally. It should be noted that the formal document supports validation, but we should also provide an informal version that is structured differently, which the users can understand easily.

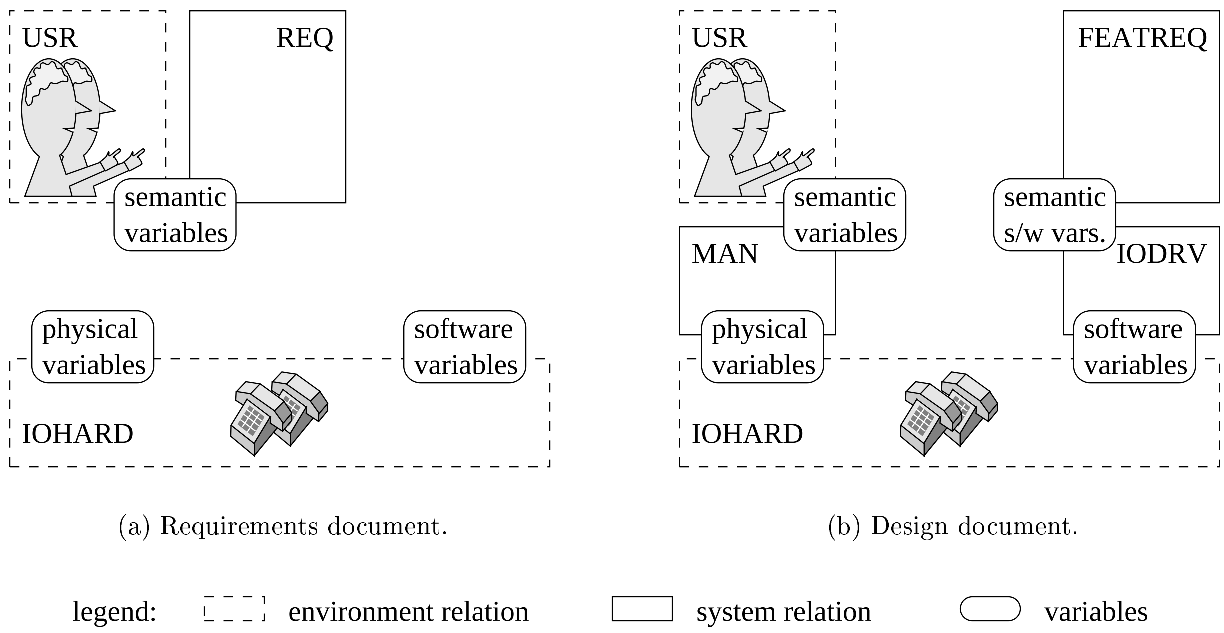

Figure 2.1: Structure of the documents.

The users' interface relation, for short *UIF*, is composed of the relations MAN, IOHARD, and IODRV:

$$\mathrm{UIF} \stackrel{\mathrm{def}}{=} \mathrm{MAN} \mathbin{\hat{\cap}} \mathrm{IOHARD} \mathbin{\hat{\cap}} \mathrm{IODRV}$$

The users' interface relation is a part of the design, since some of its components do not exist before the system is constructed. The users' interface UIF is a *module in the information-hiding sense*, as discussed in Section 2.3.1. Its *secret* is how the high-level semantic variables are mapped to the low-level syntactic physical and software variables. This encapsulation becomes possible by introducing an explicit user manual relation MAN.

Overall, all the relations in Figure 2.1(b) with solid lines together form the design *DSGN* and must implement the relation REQ in Figure 2.1(a), in the given environment. Formally, we write this as:

$$(\mathrm{DSGN} \mathbin{\hat{\cap}} \mathrm{NAT}) \mathrel{\hat{\subseteq}} (\mathrm{REQ} \mathbin{\hat{\cap}} \mathrm{NAT})$$

$$\text{with} \quad \mathrm{NAT} \stackrel{\mathrm{def}}{=} \mathrm{USR} \mathbin{\hat{\cap}} \mathrm{IOHARD}$$

$$\text{and} \quad \mathrm{DSGN} \stackrel{\mathrm{def}}{=} \mathrm{MAN} \mathbin{\hat{\cap}} \mathrm{IODRV} \mathbin{\hat{\cap}} \mathrm{FEATREQ}$$

Analogously to the composition $\hat{\cap}$, we define the variable-extended implication predicate $R \mathrel{\hat{\subseteq}} S$ (for relations S that have no design variables)

$$\begin{aligned} R \mathrel{\hat{\subseteq}} S \stackrel{\mathrm{def}}{=} \; & (\forall r \in R : r{\downarrow_{\mathrm{S}}} \in S) \wedge \\ & (\Theta_{R,S} = \Theta_{R,R}) \wedge (\{\underset{\sim}{m}^t{\downarrow_{\mathrm{S}}} \mid \underset{\sim}{m}^t \in \mathsf{mon}(R)\} = \mathsf{mon}(S)) \end{aligned}$$

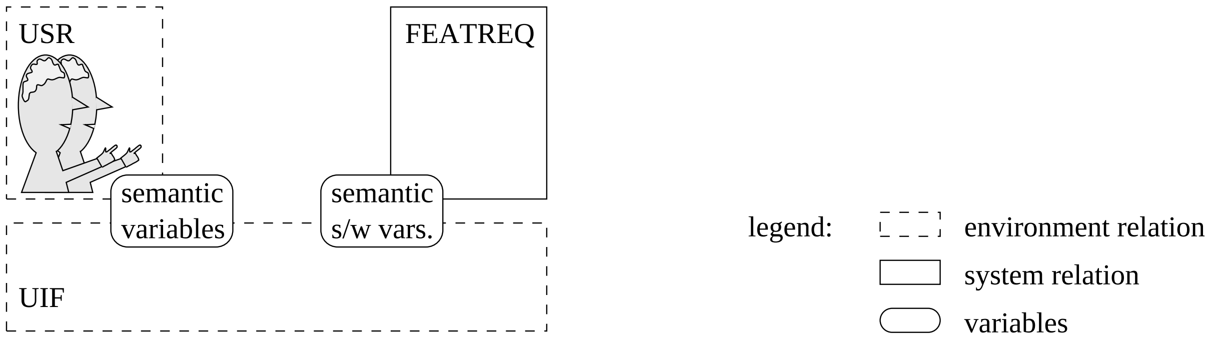

Figure 2.2: Design with a users' interface that maps semantic variables one-to-one onto semantic software variables.

2.3.3 Coordinating the Encoding of Semantic Signals

We have seen how the users' interface aspects of *one* feature can be encapsulated in the design. But how can we coordinate this for many features, and with the base system? The answer is, of course, that there must be one single users' interface module in the environment of all these features, and the concerns of the users' interface must not be part of the design of the functionality of the features at all. That means that we have to provide once and for all an information-hiding module that provides a one-to-one relation between the semantic signals in the mind of the users and a representation of them as semantic software signals, as shown in Figure 2.2.

The users' interface module must be *extensible*; the module must allow pairs of semantic mental variables and semantic software variables to be added to its interface. Now, we have split the job of implementing the requirements of a new feature into two parts: implementing the behaviour of the feature, and extending the users' interface module, if necessary.

We have to stress that, as already mentioned in Section 2.3.1, a module in the information-hiding sense does not need to be implemented in one centralized spot.

Adding new semantic variables to the users' interface module may be done either offline during the feature design process, or it may be done dynamically by an online variable registration facility in the users' interface. When a designer extends the user interface module offline, he can consider "good" human-computer interface design, and he can perform extensive validation of the correctness of his result, as discussed in Section 2.5 below. The drawback of offline extension is that all features that will access the users' interface must be known beforehand. In a dynamic and heterogenous network, this might prove difficult.

Alternatively, we can design a set of generic semantic variables beforehand, for which we design a workable mapping to the syntactic variables, and then let an automated agent within the users' interface module assign these generic variables to arbitrary features on request. This assignment process must include a suitable online instruction of the affected users, for example announcements. If we decide for an automated registration agent, the users' interface module must include some special variables that features can use to issue registration requests.

There has already been some work in the above direction. Zibman et. al. [ZWO+95] take a more general view and propose an architecture out of agents that separates concerns, in conjunction with a processing model. The architecture includes a terminal agent, which, amongst others, manages the dynamic assignment of signals to a new service. Since the authors focus more on general aspects, this is not detailed further; also, they present no formal model of their agent architecture. Keck [Kec98] proposes a service admission control scheme for the IN that keeps track of the resources relevant to a session and that notifies services in case of possible interactions. Also related is the work on Open Distributed Processing (ODP) [FLdM95], which attempts to identify several useful abstractions and to separate several communication-related concerns. Van der Linden [vdL94], for example, proposes to use an ODP architecture to beat feature interactions, and the TINA initiative [Ab+97] customizes the ODP ideas for telecommunication networks, and in particular proposes an architecture that includes user agents. Weiss and Gray [WeGr97] present another agent approach in the telecommunications domain, a multi-agent architecture for networked applications.

Our discussion on extending the users' interface module excluded the question of how the functionality of the telecommunication system should be extended, i.e., how the relation FEATREQ in Figure 2.2 relates to and modifies the base system and other features. This is outside the scope of this chapter, it is basically the question of how to prevent feature interactions not related to the users' interface.

2.4 Application to the Existing IN Architecture

How can we apply the above general principle to a concrete framework like the IN architecture?

The IN standard allows to build new services on top of the basic call processing by using Service Independent Building Blocks (SIBs), and by connecting these with so-called Service Logic [ITU93b]. One of the SIBs defined in the IN CS-1 standard is a "user interaction" SIB. The standard states that this SIB provides a call party with information (e.g. announcements) and/or collects information from a call party. The announcements can be, for example: DTMF tones, a customized or generic audio message, or network progression tones (e.g. dial tone, busy tone, etc.). The collected information can be, for example: DTMF tones, audio, or IA5 String text [ITU93a]. The parameters of this SIB include, among others, the announcement ID, the number of repetitions, and the number of the expected digits. Thus, the user interaction SIB clearly provides a user interface at the syntactic level, according the model of Foley and Wallace.

Furthermore, there are other SIBs that share user interface related concerns, for example "queue" (plays announcements), "verify" (performs a syntactic check of user input), and "translate" (relies, e.g., on the structure of directory numbers).

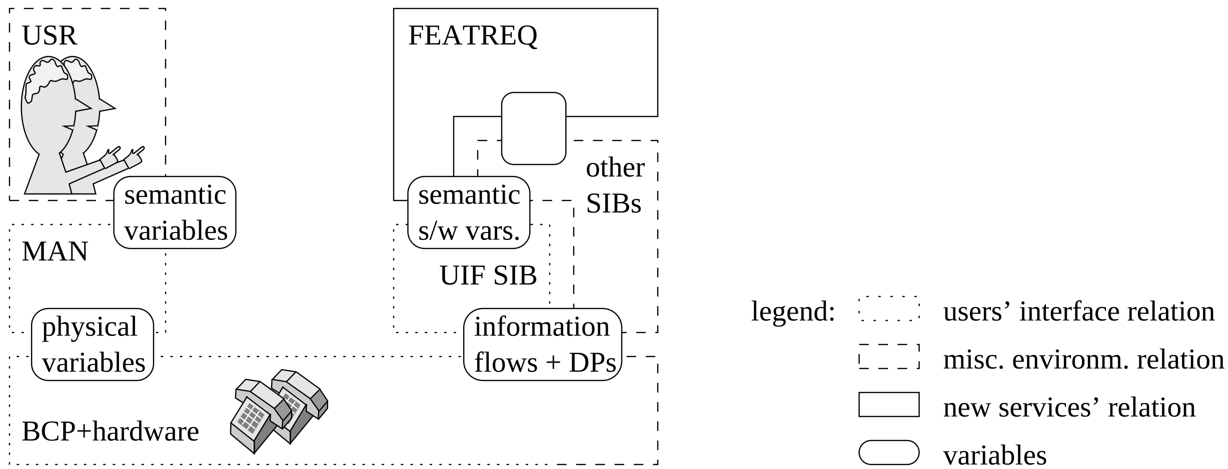

Figure 2.3: Encapsulated users' interface in the IN architecture.

The IN SIBs are intended to be reusable software components. A suitable SIB can be used as part of an information-hiding module that encapsulates the users' interface and that provides variables at the semantic level. Such a SIB can be used as part of the environment of a new service to conceive.

For this, we need a redesigned set of SIBs where only the "users' interface" (UIF) SIB is concerned with the details of the users' interface. In order that other concerns can be encapsulated, too, SIBs should be designed to be built hierarchically one on another. Fortunately, the concept of hierarchical SIBs is introduced in IN CS-2 [ITU97].

Furthermore, a UIF SIB will be so complex that its design should be broken down futher into sub-SIBs. For example, there should be a registration agent, as discussed in the previous section.

Our overall design architecture looks like Figure 2.3, with the "software variables" now being the Informations Flows of the Distributed Functional Plane, plus the Detection Points, which together underly the definition of the SIBs. The "IODRV" module is now the rewritten UIF SIB. The users' interface module is completed by the relevant aspects of the underlying Basic Call Process (BCP), and, of course, by the user manual relation MAN.

A certain difficulty with the concept of the Intelligent Network arises from its separation of the Basic Call Process and of IN service processing. The Basic Call Process performs user interaction, too, which can be controlled by the IN services only partially. Only when this user interaction leads to triggering a detection point, for example when a dialled number during call setup is completed, the IN service processing can notice and influence this user interaction. Furthermore, the behaviour of the Basic Call Process between the detection points cannot be changed by the IN service processing, which makes changes to the syntax of the users' interface hard. For example, when a user goes on-hook, this can induce some irreversible state changes before the next detection point is triggered, thus we cannot reassign an arbitrary meaning to this syntactic signal. Therefore, the translation between syntactic and semantic signals has less degrees of freedom, leaving even less freely usable syntactic signals for new services.

Nevertheless, we can design a set of generic semantic variables for the users' interface module with a workable mapping to the syntactic variables, as proposed above. But any service requesting dynamically for the assignment of such a generic variable must be prepared that no more variables are available, and the service must react by a graceful degradation. Alternatively, we might provide an arbitrary number of generic semantic variables, but with increasingly reduced quality of usage comfort, such that a service has to decide whether to accept an assignment offered by the registration agent, or to degrade gracefully, again.

The semantic variables of the users' interface are each associated to only one user, since they reflect mental processes. Therefore, the corresponding syntactic physical variables of one user are associated to one physical Service Control Point (SCP) only. As long as a user always employs the same line, all users' interface processing related to him can be performed at the same location. As soon as a user is allowed to roam and to use his IN services on different lines, the UIF SIB becomes distributed. Now, the implementation of the UIF SIB incurs substantial additional organizatorial and technical complexity. Nevertheless, the UIF SIB still should be regarded as a single module, and its problems should be separated from other concerns of IN service functionality. In this case, the UIF SIB certainly has a sub-structure. For example, the variable registration agent can be a sub-module of its own, encapsulating the assignment strategy for the generic semantic variables.

2.5 Validation of the Users' Interface

Modelling a feature's behaviour at the semantic level, and separating the concerns of the users' interface, additionally allows for a formal validation specifically of the users' interface. The design specification for the feature's users' interface states a set of one-to-one correspondences between semantic signals in the mind of the users and a representation of them as software signals (compare Figures 2.1(b), 2.2, and 2.3). A successful design of the extended users' interface must provide a pair of mutually inverse mapping relations to syntactic signals. This task may be difficult when there are few syntactic signals. But the design can be verified formally, and iterated until it is proven correct.

There are two kinds of properties that we can verify. The first is whether the design indeed implements the *functional requirements* we have stated formally, that is, whether there is indeed a one-to-one mapping of mental to software semantic signals. Possible errors include (a) a *translation to an incorrect*, i.e., another, semantic *signal*, when the two mapping functions MAN and IODRV do not fit properly, and, in particular, (b) a *non-deterministic interpretation* of syntactic signals, which constitutes one of the dreaded signalling ambiguities.

Even without having an explicit relation MAN, we can do a check for a non-deterministic interpretation, by generating and checking all permutations of syntactic signals. This is a common approach [DBL97, ChOh95, BoVe94], but a disadvantage is that it may lead to false warnings, since there are (possibly many) state/signal combinations that are reachable only when the system is not used according to its usage rules, and the users then should not expect the system's features to work as intended anyway.

The second kind of properties of the users' interface design that we can validate are on *usability*. In the area of human-computer interface research, there have been identified numerous properties which are desirable or undesirable (cf., e.g., [Shn97]), and several of them can be checked even formally [Jac83]. For example, (a) *almost-alike states* of the users' interface [Par69] tend to confuse users, (b) *interactive deadlock* [DDH83] prevents users from leaving a state due to (in particular) insufficient help available, and (c) the amount of *character level ambiguity* [Thi82] should be minimized, which is a measure derived from the number of different modes in which each keystroke has different meanings. The latter optimization is of particular interest in our context, where there are many more semantic signals than syntactic signals.

In this report, we are only concerned about how to gather the information necessary for a validation step, as one of its preconditions. We do not discuss how to perform an actual validation step; this is a wide area of research itself [DBL97, ChOh95, BoVe94].

2.6 Generalization of the Four Variable Model

In Section 2.3, we have seen how we can decompose a *design* into smaller components step by step. A sub-component in a design document becomes a requirement document in the next step. This process continues until we arrive at already existing components that we can use by simply buying them. These components can consist of software (e.g., libraries), or hardware (e.g., buttons), or both (e.g., Ethernet card). They can even be humans.

The notion "four variable model" has been coined in the literature on Functional Documentation, it refers to the monitored, controlled, input, and output variables. The starting point for the design process is the requirements document with only one boundary (system–environment) and thus two vectors of variables (monitored and controlled). The first design step then introduces a new boundary into the system (e.g., UIF–APPLREQ), and thus two new vectors of variables (input and output)[3], rendering a total of four kinds of variables.

As we have seen in Section 2.3, this design step can be applied again recursively, introducing further pairs of sets of input and output variables. Furthermore, a design step does not necessarily introduce one boundary at a time, we may as well split one component into more than two sub-components (e.g., IODRV, IOHARD, MAN). With such a more complex structure, the variable sets do not need to come in pairs anymore. Therefore, the four variable model can regarded as a special case of a more general n variable model.[4]

[3]Input and output variables are the sub-systems' boundaries; they are abstractions of entities that exist in the world before and independent of the sub-systems. They do not exist in the world before the design of the top-level system begins, but they do exist before the sub-systems are designed. With this respect, they are very different from the time functions used in the requirements document, which have no real-world counterpart.

[4]Before we achieve a complete and formal generalization to a n variable model, some more work is required, though. In particular, our current approach does not yet specify which of the sub-components

2.7 Summary and Outlook

The causes for feature interactions are many, therefore many remedies will have to be applied together. In previous work we investigated the detection and resolution of feature interactions by formal methods and tool support [Bre97, Bre96]. This rendered promising results; nevertheless, one should try to avoid problems in the first place, if one can. Therefore, in this chapter we aim at avoiding a certain kind of feature interaction by employing a more modular structure for the design of a telephone switching system, and its features.

We proposed to apply results from the research on human-computer interfaces (HCI) to the specification of requirements of telecommunication systems. In particular, these requirements should not be expressed on the level of syntactic variables, such as "flash hook" or certain button presses, but in terms of semantic variables such as "ReleaseAndReconnect" (for Credit Card Calling). A semantic variable reflects a decision of a user to perform some action, or a perception of a user of someone else's action.

In this way, feature interaction problems related to the users' interface disappear from the requirements document and become a design problem. For the design, we proposed to separate concerns and proposed to introduce an information-hiding module encapsulating the details of the translation of semantic signals to syntactic signals. This confines the feature interaction problems related to the users' interface to this module only. In contrast, for example the Feature Interaction Detection Contest instructions [GBGO98] mingle these problems freely with all kinds of other problems, thus increasing complexity and making solutions harder.

We also briefly discussed how the general approach can be applied to the existing Intelligent Network (IN) architecture in particular. Finally, the encapsulation of the details of the users' interface, together with an entirely formal description of it, provides the necessary description that enables a formal validation of the users' interface, in particular for feature interaction problems related to it.

In recent work, Miller et. al. have specified the mode logic of a flight guidance system in CoRE and SCR (which are variants of Functional Documentation), and they have found that the requirements specification of the user interface should have been separated in quite a similar way as proposed here [Mil98]. More results will become available later.

The users' interface is only one of the resources of a telecommunication system. Therefore, a telecommunication system architecture that avoids (at least many) feature interaction problems should encapsulate the details of other resources, too. More work on this is necessary. A related approach is the application of an agent architecture in the telecommunications domain, on which some work exists. The advent of a host of new features of telecommunications systems, which we expect to work concurrently and independently, is with some respects comparable to the concurrency and its problems when multi-user time-sharing processing was introduced for computer operating systems. The basic ideas of encapsulation developed at that time are not new anymore, but important tasks are now (a) to apply this knowledge to the specific structure of telecommunication systems, while

controls which design variable, and which sub-components monitor it only.

they change and become more and more complex; and (b) to combine this with formal methods and the tool support these methods provide.

Chapter 3

Must and May – Requirements Maintenance

3.1 Requirements Maintenance and Functional Documentation

Currently, providers are working hard to add many new features to their telephone switching systems. Typically, such a feature is more or less self-contained. Furthermore, the set of requirements for the entire system is so huge that the designers of any new feature can know and handle only a subset of them. Unfortunately, the current structure of these systems forces the designers of a new feature to consider quite a large share of the entire system. This is one of the reasons for the dreaded feature interactions.

In this report, we employ the Functional Documentation approach [PaMa95, vSPM93, vS92a] to the specification of telephone switching system requirements. This approach takes great care to avoid duplication of information in the documents. Such duplication would make it it hard to change the documents consistently. This is an important issue for our application area, since successful software like a telephone switching system will have a long history of maintanance and changes.

The Functional Documentation approach proposes to structure a requirements document into several sections, for example the description of the monitored/controlled variables, the definition of the mode classes, the description of the environmental constraints, the description of mode transitions, the description of the values of the controlled variables, etc. These categories cut through the entire document and do not consider any application specific structure.

For the design of a new feature, we need a view of the system that presents only the aspects relevant for this task. One solution that has been proposed is to select the appropriate controlled variables, and to extract only those parts of the description that are necessary for the definition of the values of these variables. This approach is supported, for example, by the SCR toolset [HBGL95, BhHe97].

But for our task, we do not only need a certain view of the system, we often need to modify the system in order to fit in the new feature. Therefore, we need to know the consequences of making changes. We would like to know answers to two questions:

1. Does the change, made in the partial view, cause inadverted and undesired further changes in parts of the specification that are not included in the current view?

2. Is the property that we are changing "fundamental" to some feature, or is it just some behaviour needed for "technical" reasons?

The former question is the question whether undesired feature interactions exist. The latter question deserves some more explanation. When we conceive a voice mail feature, we may deem it fundamental that the mailbox owner may access his mailbox remotely. We now assume that the boundary of our design job are the physical signals issued by the users, therefore these physical signals have to be included into the requirements specification, and we must state how the mailbox owner can access his mailbox by pressing buttons. But, in the end, neither the mailbox owner nor the service provider care too much about the actual button bindings, as long as they are well known to the users and there are no conflicts with other features. Later, we may design other features that force us to reassign the physical signals; this does not impair the original voice mail feature, as long as they become well known again and do not cause conflicts. In this example, the possibility of access is a fundamental requirement, and the means for it are changeable requirements.

So, we want to distinguish for each feature which properties are fundamental, and which are changeable. This requires that we distinguish clearly between the specifications of the different features in the first place. Therefore, we modify the standard Functional Documentation approach with this respect:

> we propose to *structure the requirements specification along the application structure*, i.e., by the features. For each individual feature, there shall be a description according to the standard approach, and a little additional information, described below.

When conducting our case study, we tried out both approaches to structuring, and we found that grouping by features encouraged us stronger to separate different concerns, which is particularly important in our application area. For example, we started out with specifying parts of the Plain Old Telephone Service, and the specification included a description of a connection. But only after we restructured the specification, we isolated a generic connection feature, and separated it from a voice channel feature. This structure will allow us to reuse the generic connection feature unaltered if we introduce a video connection feature. It turned out that we could identify finer-grained self-contained features than we expected. This improved the separation of different concerns in our requirements specification.

An important argument in favor of the standard Functional Documentation structuring approach is that it avoids duplication of information. We found that our modified approach still has this property. For example, having both a voice and a video connection feature does not duplicate the generic connection feature. The generic connection feature is a feature in its own right, and the two other features are defined by simply referencing it.

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 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.

The Functional Documentation approach stresses 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 should be kept, such that the system's specification can be *relabelled as environment's specification* for the next task. Usually, some changes will be necessary to this description, we discuss below how to handle them. Furthermore, some parts of the system may not be relevant for the next change, and can be left out of the specification of the change requirements, we discuss this below, too.

3.2 Requirements Maintenance and Refinement

The formal refinement approach to the specification of software requirements is well understood [Par98, Abr96, BaWr90, Bac90, BaSe96, Mor88, Mor90, Mor87, Heh90, BBB$^+$85, Hoa85, Lam94, Pnu92]. In this approach, we add more details to the description step by step; each such step removes some non-determinism from the specification. There are two basic ways of using refinement, *external* refinement and *internal* refinement. With external refinement, we gradually extend the specified requirements description until the requirements document is complete. Formal rigidity ensures that all the properties added on the way are present in the final product. With internal refinement, we start out with a complete requirements document and gradually add implementation details until we arrive at an executable implementation. Formal rigidity, in particular correctness proofs or property preserving transformations, can guarantee the correctness of the implementation with respect to the requirements. In both cases, we have a linear sequence of refinement steps. In this chapter, we are concerned with external refinement only.

Unfortunately, both refinement approaches, in the above form, are not well suited for modifying and extending a system that already has been installed and run. The reason is that *a revision of an early refinement step invalidates all subsequent steps.* This is also a general problem of other top-down software development approaches.

Parnas [Par76] defines *program families* as sets of programs with extensive common properties. He proposes to keep the results of the intermediate (vertical) 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 above 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, again, sets of requirements with extensive common properties. The requirements specification of a large software system should not contain any design decisions, but nevertheless it must be organized somehow for readability and maintainability. These decisions about the organization of the requirements already have a substantial impact on their maintainability.

Most descriptions of a specific requirement from the set of requirements depend on other parts of the requirements specification. This can make large requirements specifications hard to change, similar to how large programs can be hard to change. We therefore apply the above ideas to requirements specifications and

> we propose to *distinguish formally between those properties* of the initial system that we deem *fundamental*, and those properties that may *change* in later revisions.

This allows us to design many extension features to a feature independently, if they rely only on the fundamental properties of the feature. Even though any new feature may alter the system's behaviour in many ways (in the "changeable" part), this will not invalidate previous extensions which have been designed without knowledge of the features to come after them.

Our approach may be regarded as a continuation of the information hiding principle (compare, e.g., [Par72]). It demands to separate fundamental design decisions from those likely to change; the latter should be encapsulated as a secret into one module[1]. The changeable properties above should be organized such that those most likely to change are easiest to

[1] A "module" doesn't need to be a single implementation object like a procedure or a binary. For example, the inline-capabilities of modern object-oriented languages and compilers allow to keep the definition of an object in one place while the actual machine code is spread out far.

change, which means that these changes affect a small scope only. In effect, this extends and formalizes this aspect in the approach of Functional Documentation, which labels the parts of a specification as either "fundamental" or "changeable", to a multi-level approach. Besides on the idea of program families [Par76], our approach similarly builds on the ideas on "designing software for ease of extension and contraction" and the *uses-hierarchy* [Par79].

Our approach is an external refinement approach, but

> *we propose to order our external refinement steps by the likeliness by which they must be taken back*, with the most unlikely first.

Furthermore, we propose to state the dependences of the external refinement steps explicitly, thus rendering a directed acyclic graph (DAG, compare Section 3.4 below) of required properties. When we specify further members of the requirements family, we thus produce a *tree of DAGs of required properties.* Each requirements specification is the composition of all specified required properties in a DAG, and the individual requirements specifications together form a tree by sharing more or less parts.

3.3 Structure of a Requirements Specification

We propose to structure a requirements specification[2] into the following sections.

1. an *introductory section*, as in the standard Functional Documentation approach, that explains, e.g., notation where necessary, and gives a general background and overview.
2. a *sequence of sections describing one feature each.*
3. an *index section* which allows to look up all the pages on which the defined terms are used; this index can be generated automatically.
4. the usual *lists of figures and of tables* and a *bibliography section.*

Additionally, there could be a *dictionary section* explaining the terms in the index, and a *glossary section*, as proposed by van Schouwen [vS92a]. We did not investigate the use of the latter two sections. An example for the structure proposed may be found in the Annex of this report.

Each of the sections describing one feature mentioned in item 2 above should have the following substructure:

2.1. a subsection on *the fundamental requirements for the environment* of the feature,

2.2. a subsection on *the fundamental requirements for the system*, i.e., the feature itself,

[2] In this report, we use the term "requirements specification" for both system requirements and software requirements, since the approach can be applied to both. Nevertheless, our requirements typically will be implemented by software plus some standard computing platform.

2.3. a subsection on *the changeable requirements for the environment*,

2.4. a subsection on *the changeable requirements for the system*, and

2.5. a optional subsection that presents a *graphical representation of the dependences among the properties* of this section (see Section 3.4 below).

Each of these subsections should contain a sequence of one or more *sub-subsections with the respective properties.* A subsection may contain only one property, but sometimes there may be more than one property in the respective category. Generally, any structural unit that is empty may be omitted.

Each sub-subsection describing a single property should address the following issues in the following order (omitting anything not applicable, again). This structure is an adapted copy of the remaining global document structure proposed by van Schouwen. There should be a description of

1. *which properties' descriptions are relied on* by this property description, and *which properties are replaced*,
2. the *data types* introduced by this property,
3. the *specification parameters* (if the specification is generic),
4. the *monitored/controlled variables* introduced for this property,
5. the *mode classes* and their *initial values*,
6. the *mode transition relations*,
7. the *time functions* and their *initial values*, and
8. the *values of the controlled variables.*

This list and this order was sufficient for our case study; further experiences might necessitate more categories or maybe a modified order in some special cases.

Every such description in our case study has either the form of a table, or it is printed in a box. Since every table has a frame box, the entire formal description of a property is a logical conjunction of all material printed in boxes. There may be additional informal explanations and motivations.

In the beginning of each sub-subsection, we state on which properties the current property *relies*, and which properties are *replaced* by it. In the first case, the new property specializes the named pre-existing properties, in particular by reducing non-determinism. In the second case, the named properties are specifically excluded from the final semantics of this property, and only the predecessor properties of these properties are relied on. Please see also Section 3.6 on the semantics of a specification.

3.4 Composition of Properties

When we write a requirements specification, we write a set of properties. When we enhance a specification by refinement, we add a new property that modifies one or more pre-existing properties. When writing new properties, we do not want to repeat all other parts of the specification that we need to express the new thing. For example, when we write a credit card calling feature, we do not want to write again what "connection" and "billing" mean.

We avoid duplicating definitions and relations by stating at the beginning of each property's description on which other properties it *relies* on. The semantics of this is that all named properties are included literally into the current description. It is not allowed to redefine identifiers, and name clashes must be avoided. (Future work may lift these restrictions by a more elaborate scheme.) The inclusion is recursive. A property description may add a further relation, i.e., a controlled variable relation, on variables inherited from another property, the resulting relation is an intersection of the relations. This effectively refines the inherited property by removing some non-determinism.

The relies-upon-relation among properties renders us a dependence graph. We demand that it must be acyclic. Furthermore, environment properties may rely only on other environment properties, while system properties may rely on both environment and other system properties. As an additional guideline, fundamental properties should rely only on fundamental properties, while changeable properties may rely on both fundamental and other changable properties. If this guideline is not followed, adverse feature interactions are more likely, see the next section.

When changeable properties rely on other changeable properties, we obtain a *hierarchy of changeable properties*. Future work could work this out further, here, we are content with the basic distinction between "fundamental" and "changeable".

The above rules result in a *Directed Acyclic Graph (DAG)* of properties, both for each individual feature, and for the entire system. This idea of a formal DAG of properties is related to the idea of the *uses-hierarchy* [Par79].

In our approach, we could derive some kind of measure of how easily a specification can be changed. The specification can be changed more easily when the DAG is "bushy" and when the paths are short. Each change to a node will affect only few other nodes. When the DAG resembles more a linear order, it is probable that a change will cut off many nodes, and that a lot of work will be required to write substitutes. Also, fundamental properties should be close to the root node(s), and easily changeable properties close to the leaves. Of course, our actual measure of the quality of the specification structure depends on the expectations of which nodes will be changed later. Therefore, we hesitate to develop a more formal notion of quality measure, since its key parameter usually is not known precisely.

As long as features are composed by a kind of logical "and", no feature interactions among safety properties are possible, since all safety properties are preserved. 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". Such requirements can be found frequently, since these will be the expectations of customers and users. 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 some 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.[3]

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. In previous work [Bre97, Bre96], we restricted the admissible changes to the coarser granularity of (Estelle) transitions. Now,

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

This explicit interface consists in properties labelled as "changeable" which are designed to be replaced. A prerequisite for this approach is a suitably fine-grained structure of requirements, such that the designer of a new feature will find an appropriate position for a replacement.

Another prerequisite is that the basic concepts necessary for the new feature are already part of the requirements document. If they are not, we argue that they should be introduced first, in order to maintain the long-term consistency of the requirements document. An example is the Screening Fundamentals feature of the case study in the appendix. It allows us to introduce some screening features smoothly, without "mis-using" hooks that were designed for some other purpose. If the system to modify does not contain the notion of connection screening, we are to expect a number of feature interactions that are due to violated assumptions; for example about when a connection will surely complete.

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. Therefore, such feature operators are a specialization of the superimposition operator concept proposed by Katz [Kat93]. Such operators provide an explicit interface for the added feature: what it relies on as a prerequisite, what it provides, and what it modifies. But the system to be extended, in general, has no defined interface in this approach.

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

3.5 Detecting Feature Interactions

Due to our approach of replacing properties only explicitly and completely, it is clear which of the stated requirements still hold after adding a feature. Furthermore, it is clear in the replacement process whether any fundamental property has been removed. We consider such a removal as an adverse feature interaction. When a changeable property is removed, this is a feature interaction, too, but by definition it can not be adverse. It is merely a "technical change" to the feature in question.

When a property is replaced, then we label all properties that rely on this property as "removed", as stated in the previous section. This can lead to a situation where a property p_1 removes a property p_2, which is relied upon, directly or indirectly, by a property p_3. Even if p_2 is a changeable property, and thus poses no problem with respect to feature interactions, property p_3 may be a fundamental property of some other feature, which is indirectly labelled as removed by this dependence. Such situations can arise when a fundamental property relies on a changeable property. These situations of adverse feature interactions can be detected by a syntactic check of the directed acyclic graph (DAG) of properties.

The composition process may encounter situations in which two properties should replace the same property. Technically, this is allowed in our approach. The property to remove is removed, and both new properties are conjoined into the global meaning of the specification. 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. All safety properties are still valid, so no harm is done with this respect, but in requirements specifications there is usually at least the (maybe implicit) lifeness requirement that the system should not deadlock; which is violated.

If a situation is encountered in which two properties should replace the same property, the specifier should investigate whether the two new properties are compatible with respect to lifeness. This can be done by any means from informal inspection to formal validation. For an example see our case study Section A.8. The advantage of our approach is here the fine-grained structure of properties we propose: The validation is easier if the properties to check are "small"[4].

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. In particular, we have to check whether the environmental properties required in the one specification have not been modified (i.e., replaced) in the other. Such a check concerns the syntactic structure of the specifications and can be performed automatically by suitable tools.

In order to reduce the likeliness of feature interaction problems, the developer of a feature

[4]These are qualitative arguments only, we don't want to actually measure the "size" of a property. Our point is that experience has shown that a divide-and-conquer technique often helps.

should not take a snapshot of the entire current specification DAG as the starting point, but only a pruned version of it. He should cut away as many nodes as possible and retain only those nodes on whose properties he must rely on as environment of his new feature.

3.6 Sketch of a Semantics of the Specification

The semantics of the Functional Documentation approach for documenting requirements is based on a set of monitored/controlled variables, and two relations over them: the environment relation NAT and the system requirements relation REQ, compare Section 2.2.2. In Section 2.2.3, we extended this approach by *variable-extended composition*, and by a set of design variables. We do not use design variables in the documentation of requirements, therefore, this third set is always empty in a requirements document. But we do use our variable-extended composition operator $\hat{\cap}$ for the composition of properties.

The variable-extended composition operator $\hat{\cap}$ is basically a logical "and" for properties, adjusted for the possible difference of the sets of variables over which these properties are defined. We chose to conjoin properties in this way in order to make sure that the operation preserves all safety properties. Safety properties state that "nothing bad may happen", therefore the operation of reducing what can happen by a logical "and" cannot invalidate such properties. This subject is discussed formally, e.g., by Abadi and Lamport [AbLa95].[5]

For a property that does not rely on any other property, and does not replace any other property either, we define the meaning of the corresponding sub-subsection to be the same as if this sub-subsection were a self-contained specification of either NAT or REQ in the standard Functional Documentation approach. An exception to this is that the time-function vectors are additionally labelled by the variable names, as described in Section 2.2.3, and that the resulting relation has a third vector of time-functions, which is always empty.

For a property that does rely on other properties, we define the meaning of the corresponding subsection in the same way, after we have performed a textual inclusion of the data types, specification parameters, monitored/controlled variables, and mode classes in the (recursively) referenced sub-subsections.

If a property replaces other properties, this recursive textual inclusion is performed in exactly the same way, except that all items from the first level of reference are *not* included.

If a property is replaced by some other property in the same requirements specification document, we label it as "removed". Also, we label all properties as "removed" that rely on an undefined property.

We define the semantics of a feature to be the variable-extended composition of all its not-removed properties, and of all not-removed properties referenced in the above recursive inclusion process. The relation REQ of this feature is defined as the variable-extended composition of all "fundamental or changeable requirements for the system" for this feature,

[5] An open question is how to conjoin lifeness properties, which state that "something good must eventually happen".

without recursive inclusion. The relation NAT of this feature is defined as the variable-extended composition of all “fundamental or changeable requirements for the environment”, and of all requirements referenced in the recursive inclusion process, no matter in which section.

If a fundamental property of a feature is labelled as “removed”, then this property is missing from the feature, and we have a case of an adverse feature interaction. This is discussed further in the next section.

We define the semantics of an entire requirements document, consisting of several features, analogously, except that the relation REQ now is composed of all “requirements for the system”, and the relation NAT is composed of all “requirements for the environment”, without the above reassignment.

The idea behind the different assignments of properties to REQ and NAT is to reflect what should be developed, according to the document. If a single feature is to be developed (i.e., added to a running system), then everything else in the document describing the system is in the environment of this feature. If the entire system, consisting of many features, is to be constructed, then we have no incremental development, but only a document for a single development contract. The only difference to the standard Functional Documentation approach then is that this document is structured hierarchically in order to facilitate later incremental development.

Currently, this description of the semantics is a sketch only, due to time constraints. Working it out in a more detailed and formal fashion is for future work.

Chapter 4

Summary and Future Work

In this report, we investigated two ways to avoid feature interaction problems in telephone switching systems. Both aim at improving the modularisation of the requirements document, and the first also of the design of the system. By modularisation, we mean the separation of concerns. It enables us to apply a divide-and-conquer technique in order to cope with the inherent complexity of such large systems, and it, in particular, helps limiting the scope of changes to one module only, thus reducing the amount of maintenance work and the likeliness of mistakes.

In Chapter 2, we proposed to separate the concerns of the users' interface of a telephone switching system from the concerns of switching itself, and of the various feature functionalities that are added to these systems. We proposed to apply results from the research on human-computer interfaces (HCI) to the specification of requirements of telecommunication systems. After we had confined the feature interactions in question to one information-hiding module, we showed what is necessary to detect them by formal validation. We discussed briefly how the general approach can be applied to the existing Intelligent Network (IN) architecture in particular.

In Chapter 3, we investigated the incremental composition of requirements. When a successful system evolves over time, there is not only one contract between a developer and a customer, but an entire sequence of contracts. The Functional Documentation approach [PaMa95, vSPM93, vS92a] is suitable for specifying precisely the requirements of a system, such that they can become the base of a development contract. We investigated how this approach can be extended to cover a sequence of contracts, while maintaining an overall system requirements document at the same time, that describes the current status quo of the system. In addition to this overall document, the developer and the customer must specify precisely what the extent of the next incremental change or extension is, and on what the developer can rely upon in doing his job.

To achieve this, we proposed a structure for the requirements document that is different from the one in the standard Functional Documentation approach. We proposed to group the requirements by features, with each such group being essentially a miniature requirements document in the standard style.

One important aim of the standard approach is to avoid duplication of information in the documents. We, too, achieve this by using a hierarchical structure of property definitions that only reference the other parts of the specification they need, instead of copying them.

Our approach is related to the refinement approach for requirements specification. But we avoid a disadvantage that the standard refinement approach has with respect to incremental changes to the requirements specification. In the standard external refinement approach, there exists a linear sequence of refinement steps that finally lead to a complete requirements specification. In case one of these steps has to be taken back, all subsequent steps are invalidated, including all the work necessary for performing them. We reduce the necessary work for incremental changes and extensions by two measures: first, we use an explicit directed acyclic graph (DAG) instead of a linear sequence (or a linear sequence with implicit dependence information). Second, we distinguish between *fundamental* and *changeable* properties, and we *order the refinement steps by the likeliness by which they must be taken back*, with the most unlikely first. Our approach renders a *family of requirements specifications*, and it is therefore closely related to the work on families of programs [Par76], where decisions taken during the design phase of program development are ordered in a similar fashion as we order the properties in a requirements document.

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. In our approach, *the system to modify has an explicit interface* where it can be modified. This explicit interface consists in properties labelled as "changeable" which are designed to be replaced. Only if the module to extend has an explicit interface, the approach guarantees to protect its information-hiding secret. A prerequisite for our approach is a suitably fine-grained structure of requirements, such that the designer of a new feature will find an appropriate position for a replacement. In case that we want to add a new feature that cannot be accomodated for by these explicit interfaces, this is a sign that we need a new concept, which in turn should be added explicitly first, before it is used.

In this report, we experiment with a composition operator that is suitably restricted in the kinds of change it can effect. The operator allows only to either extend a property without changing it, or to replace the property entirely by a new property, by stating this explicitly.

In the appendix, we present a case study that applies our ideas. It presents a requirements specification of a simplified telephone switching system. The user interface variables are described on the semantic level only, thus separating them from any syntactic details, as proposed in Chapter 2. Limited time prevented us from elaborating on the details of the syntactic users' interface in the case study. The structure of the requirements specification is the one proposed in Chapter 3.

Much and interesting future work remains to be done. Besides more work on the case study, in particular on including more features, we would like to work out our ideas in Chapter 2 and 3 more formally. Also, more experience with the practical application of our general guidelines is needed: How can we order our refinement steps by the likeliness by which they must be taken back, when such an ordering may require a thorough knowledge of the application domain to be successful. And furthermore, there exist other recommendations about which steps to take first (for an overview, see [Par98]), and it still needs to be investigated how these recommendations can be reconciled with ours.

Appendix A

Case Study

A.1 Introductory Remarks to the Case Study

The reader of this case study should be generally familiar with Functional Documentation, we do not present a basic introduction here. Parnas and Madey [PaMa95] introduce to the concepts of the approach, while van Schouwen [vS92a] presents a version of the tabular notation on which we build.

A.1.1 Notational Conventions

We generally adhere to the notational conventions proposed by van Schouwen [vS92a]. The most important deviations are listed below.

All items defined formally, such as variables or mode classes, are typeset in a $\mathsf{sansserif}$ font. Furthermore, they are prefixed with a superscript letter that indicates their kind, for example as in $^{c}\mathsf{voiceChannel}$, according to the scheme in Table A.1.

While a mode may be regarded as a predicate over the history, thus as a function from histories to boolean values, a *time function* is a function from histories to a (usually) non-boolean range. For example, it may render the last user identity dialled.

For mode transition relations, we use the tabular notation proposed in the A-7 requirements document [AFHB^{+}92], extended by Peters [Pet98]. This notation allows to look at the mode transitions both with respect to any specific mode as well as with respect to any specific condition. Both is needed when dealing with communication systems. The other forms of mode transition tables (and state transition tables) collected in the survey by Abraham [Abr97] appear to be less suited with this respect.

Peters proposes a set of possible entries for mode transition tables, which are {@T, @F, t, f, T, F, –, $*$}. We extend it by t$'$ and f$'$. The meaning of these entries is listed in Table A.2, where p is the current value of the predicate in question, and p' is the next value.

Table A.1: letters to indicate kind of identifier

letter	kind of identifier
m	monitored variable
c	controlled variable
i	input variable[a]
o	output variable[a]
t	type
p	specification parameter
$=$	constant
M	mode
C	mode class
T	mode transition relation
f	time function
r	relation on variables or time functions
P	property

[a]Input and output variables are used in the system design document only.

Table A.2: meaning of possible entries in a table

entry	condition
@T	$\neg p \wedge p'$
@F	$p \wedge \neg p'$
t	p
f	$\neg p$
T	$p \wedge p'$
F	$\neg p \wedge \neg p'$
–	$p = p'$
$*$	$\stackrel{=}{}$true
t′	p'
f′	$\neg p'$

A.1.2 Precisions and Tolerances

The mode transition relations and output relations in the rest of this appendix describe the ideal behaviour of the system. The real behaviour derives from this when we first apply a precision function to any monitored variable, and afterwards apply a tolerance function to any controlled variable. For example, this allows for a non-zero length of polling cycles over input variables and for delays in which the controlled variables can change to the required values. Compare the discussion by van Schouwen [vS92a]. The ideal behaviour REQ on the one hand and the precisions and tolerances on the other hand are separate concerns, therefore they are treated separately.

Precisions can be used to describe how exact a continuous value like a temperature must be measured. This does not apply here since we have boolean monitored variables only. But we still need certain kinds of precisions: when a boolean variable changes,

- what is the minimal length of time that the new value holds until the system *may* notice it,
- what is the minimal length of time that the new value holds until the system *must* notice it, and
- after which (minimal, maximal) *delay* does the effect of noticing take place.

The third kind of deviation from immediate noticing is useful for specifying a distributed system. Even though a change of value may happen only for a short period (still satisfying the first kind of precision), we may want to allow much more time to pass until the system actually changes its current mode. This allows for implementations that include long transmission lines.

In our context, tolerances appear for boolean controlled variables only, and thus as timing tolerances. There are minimal and maximal delays after which the corresponding mode change must be followed by a change of the controlled variable.

Details on precisions and tolerances are not worked out in this case study for lack of time. They would include, for example, a statement on how long it may last from a connect request (e.g., completing a dialled number) until the corresponding connect indication is given (e.g., the corresponding phone starts to ring). Obviously, we have to allow for some delay due to the distributed nature of the envisaged system. Additionally, a complete specification would have to state an upper bound on this delay. Such a delay can become important when several monitored variables change their values within a short period of time. This can lead to race conditions, of which the outcome may depend on this precision.

A.2 Basic Environment Feature

This "feature" is classified as a feature for orthogonality only, it contributes some very basic information on the environment into which all the other features will be embedded. We introduce the notions of *humans*, *users*, and *devices*.

A.2.1 Fundamental Requirements for the Environment

A.2.1.1 ^{P}**HumanFun** – Humans

Relies on properties: none

Table A.3: data types for ^{m}humans

name	description	values
^{t}humanIds	the (finite) set of identities of humans that have telephone access	Irrelevant, but type allows for equality test.

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.

Table A.4: specification parameters for ${}^{m}\mathsf{humans}$

name	**description**	**type**
${}^{p}\mathsf{initialHumans}$	the initial (finite) set of humans that have telephone access	$\mathcal{P}({}^{t}\mathsf{humanIds})$

Table A.5: monitored variables for ${}^{m}\mathsf{humans}$

name	**mon./ ctrl.**	**description**	**type**
${}^{m}\mathsf{humans}$	mon.	the current (finite) set of humans that can use the telephone system	$\mathcal{P}({}^{t}\mathsf{humanIds})$

Table A.6: relation on initial values for ${}^{m}\mathsf{humans}$

${}^{m}\mathsf{humans}(t = 0) = {}^{p}\mathsf{initialHumans}$

A.2.1.2 ${}^{P}\mathsf{\textbf{DeviceFun}}$ – Telephone Devices

Relies on properties: ${}^{P}\mathsf{HumanFun}$

Table A.7: data types for ${}^{m}\mathsf{devices}$

name	**description**	**values**
${}^{t}\mathsf{deviceIds}$	the (finite) set of identities of telephone devices	Irrelevant, but type allows for equality test.

Table A.8: specification parameters for ${}^{m}\mathsf{devices}$

name	**description**	**type**
${}^{p}\mathsf{initialDevices}$	the initial (finite) set of telephone devices	$\mathcal{P}({}^{t}\mathsf{deviceIds})$
$\forall h \in {}^{p}\mathsf{initialHumans} :$ ${}^{p}\mathsf{initialPossessedDevices}(h)$	the initial (finite) set of telephone devices that human h has under his physical control	$\mathcal{P}({}^{t}\mathsf{deviceIds})$

Table A.9: monitored variables for ${}^{m}\mathsf{devices}$

name	**mon./ ctrl.**	**description**	**type**
${}^{m}\mathsf{devices}$	mon.	the current (finite) set of telephone devices that can be used	$\mathcal{P}({}^{t}\mathsf{deviceIds})$
for all $h \in {}^{t}\mathsf{humanIds}$: ${}^{m}\mathsf{possessedDevices}(h)$	mon.	the (finite) set of telephone devices that human h currently has under his physical control	$\mathcal{P}({}^{t}\mathsf{deviceIds})$

Table A.10: relation on initial values for ${}^{m}\mathsf{devices}$ and ${}^{m}\mathsf{possessedDevices}$

${}^{m}\mathsf{devices}(t = 0) = {}^{p}\mathsf{initialDevices}$
${}^{m}\mathsf{possessedDevices}(t = 0) = {}^{p}\mathsf{initialPossessedDevices}$

A.2.1.3 ^{P}**UserFun** – User Roles

Relies on properties: ${}^{P}\mathsf{HumanFun}$

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.

Table A.11: data types for ${}^{m}\mathsf{users}$

name	**description**	**values**
${}^{t}\mathsf{userIds}$	the (finite) set of identities of user roles that have telephone access	Irrelevant, but type allows for equality test.

Table A.12: specification parameters for $^{m}\mathsf{users}$

name	description	type
$^{p}\mathsf{initialUsers}$	the initial (finite) set of user roles that have telephone access	$\mathcal{P}(^{t}\mathsf{userIds})$
$\forall u \in {}^{p}\mathsf{initialUsers}$: $^{p}\mathsf{initialUserRoles}(u)$	the initial (finite) set of user roles that user u plays	$\mathcal{P}(^{t}\mathsf{userIds})$

Table A.13: monitored variables for $^{m}\mathsf{users}$

name	mon./ ctrl.	description	type
$^{m}\mathsf{users}$	mon.	the current (finite) set of users (i.e., roles that humans can play) relevant to the telephone system	$\mathcal{P}(^{t}\mathsf{userIds})$
for all $h \in {}^{t}\mathsf{humanIds}$: $^{m}\mathsf{userRoles}(h)$	mon.	the (finite) set of users (i.e., roles) that human h currently plays	$\mathcal{P}(^{t}\mathsf{userIds})$

Table A.14: relations between $^{m}\mathsf{humans}$ and $^{m}\mathsf{userRoles}$: any human plays at least one relevant user role, as a private person.

$\forall h \in {}^{m}\mathsf{humans}$: $^{=}\mathsf{privatePersonRole}(h) \in {}^{m}\mathsf{userRoles}(h)$

Table A.15: relations on initial values for $^{m}\mathsf{users}$ and $^{m}\mathsf{userRoles}(h)$

$^{m}\mathsf{users}(t=0) = {}^{p}\mathsf{initialUsers}$
$\forall h \in {}^{p}\mathsf{initialHumans}$: $^{m}\mathsf{userRoles}(h)(t=0) = {}^{p}\mathsf{initialUserRoles}(h)$

A.2.2 Changeable Requirements for the Environment

A.2.2.1 ^{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.

Relies on properties: ${}^{P}\mathsf{HumanFun}$

Table A.16: relation on ${}^{m}\mathsf{humans}$: humans are neither added nor removed during the lifetime of the system.

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

A.2.2.2 ^{P}**DevNoRem** – Telephone Devices neither Added nor Removed

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

Relies on properties: ${}^{P}\mathsf{DeviceFun}$

Table A.17: relation on ${}^{m}\mathsf{devices}$: telephone devices are neither added nor removed during the lifetime of the system.

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

A.2.2.3 ^{P}**DevAssocNoRem** – Telephone Device Associations neither Added nor Removed

Relies on properties: ${}^{P}\mathsf{DeviceFun}$, ${}^{P}\mathsf{HumNoRem}$

Table A.18: relations on ${}^{m}\mathsf{possessedDevices}(h)$: telephone device associations are neither added to nor removed from a human during the lifetime of the system.

$\forall h \in {}^{p}\mathsf{initialHumans}:$ ${}^{m}\mathsf{possessedDevices}(h) = {}^{p}\mathsf{initialPossessedDevices}(h)$

A.2.2.4 ^{P}**UserNoRem** – Users neither Added nor Removed

Relies on properties: ${}^{P}\mathsf{UserFun}$

Table A.19: relation on ${}^{m}\mathsf{users}$: users are neither added nor removed during the lifetime of the system.

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

A.2.2.5 ^{P}**UserRolNoRem** – User Roles neither Added nor Removed

Relies on properties: ^{P}UserFun, ^{P}HumNoRem

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.

Table A.20: relations on $^{m}\mathsf{userRoles}(h)$: user roles are neither added to nor removed from a human during the lifetime of the system.

$$\forall h \in {}^{p}\mathsf{initialHumans} :$$
$${}^{m}\mathsf{userRoles}(h) = {}^{p}\mathsf{initialUserRoles}(h)$$

A.2.2.6 ^{P}**UserOneHuman** – One Human per User Role

Relies on properties: ^{P}UserFun

Table A.21: relations on $^{m}\mathsf{userRoles}(h)$: currently, a user role may be played by at most one human at a time.

$$\forall h_u, h_v \in {}^{m}\mathsf{humans}, u \in {}^{m}\mathsf{users} :$$
$$(u \in {}^{m}\mathsf{userRoles}(h_u) \wedge u \in {}^{m}\mathsf{userRoles}(h_v)) \Rightarrow (h_u = h_v)$$

A.2.3 Property Dependences

Figure A.1 on page 42 shows the uses-hierarchy among the properties of the requirements of the basic environment feature. Figure A.2 on page 42 explains the graphical notational conventions for all uses-hierarchy diagrams in this report.

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

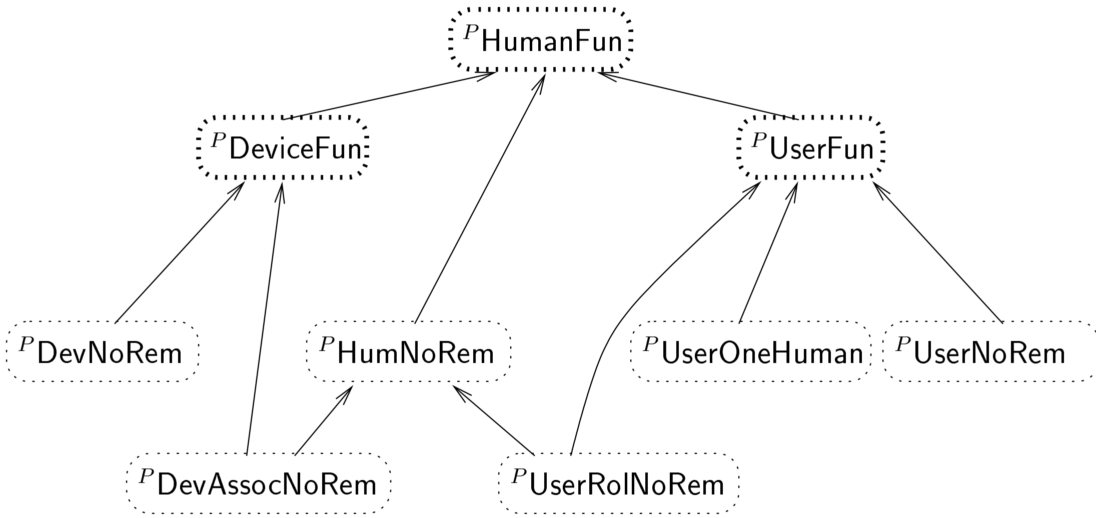

Figure A.1: The uses-hierarchy among the properties of the basic environment feature.

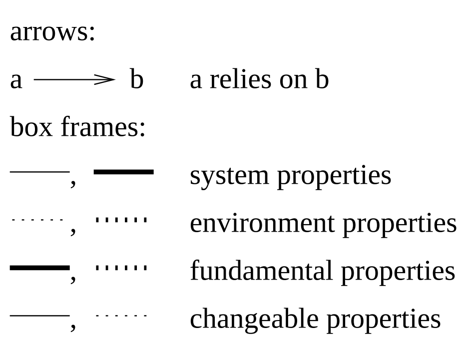

Figure A.2: The legend for all uses-hierarchy diagrams.

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.

Please note that we usually define not only one mode class at a time in the following, but a set of mode classes for each pair of users $u, v \in {}^t\mathsf{userIds}$. Furthermore, the definitions of mode transition relations make restrictions only for users $u, v \in {}^m\mathsf{users}$, with ${}^m\mathsf{users} \subseteq {}^t\mathsf{userIds}$. 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 Chapter 2, 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 Requirements for the Environment

A.3.1.1 ${}^P\mathsf{ConEnv}$ – Connection Environment

Relies on properties: ${}^P\mathsf{UserFun}$

Table A.22: monitored and controlled variables for connection management

name	**mon./ ctrl.**	**description**	**type**
for all $u, v \in {}^t\mathsf{userIds}$: ${}^m\mathsf{connectReq}(u, v)$	mon.	binary command: user u requests and then maintains a connection to user v (as long as true)	${}^t\mathsf{bool}$
for all $u \in {}^t\mathsf{userIds}$: ${}^m\mathsf{connectRsp}(u)$	mon.	binary command: user u accepts and then maintains an offered connection (as long as true)	${}^t\mathsf{bool}$

Table A.22: monitored and controlled variables for connection management (cont'd)

name	mon./ ctrl.	description	type
for all $u \in {}^{t}\mathsf{userIds}$: ${}^{c}\mathsf{connectInd}(u)$	ctrl.	binary signal: the user u is offered a connection and then it is maintained (as long as true)	${}^{t}\mathsf{bool}$
for all $u \in {}^{t}\mathsf{userIds}$: ${}^{c}\mathsf{connectCnf}(u)$	ctrl.	binary signal: the previous connection request of user u has been accepted, the connection is maintained (as long as true)	${}^{t}\mathsf{bool}$
for all $u \in {}^{t}\mathsf{userIds}$: ${}^{c}\mathsf{connectTeardownInd}(u)$	ctrl.	binary signal: the current connection or connection request of user u is torn down, and user u has still to acknowledge this.	${}^{t}\mathsf{bool}$
for all $u \in {}^{t}\mathsf{userIds}$: ${}^{c}\mathsf{teardown\text{-}Reason(uNotAvail)}(u)$	ctrl.	binary signal: if the current connection (request) of user u is torn down, this is due to the other user not being available for a connection.	${}^{t}\mathsf{bool}$

A.3.2 Fundamental Requirements for the System

A.3.2.1 ${}^{P}\mathsf{ConFun}$ – Connection

Relies on properties: ${}^{P}\mathsf{ConEnv}$

The modes of mode classes ${}^{C}\mathsf{userAvailable}(u, v)$ 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.

Table A.23: mode classes ${}^{C}\mathsf{userAvailable}(u, v)$

for all $u, v \in {}^{t}\mathsf{userIds}$:

mode	informal description
${}^{M}\mathsf{userAvail}(u, v)$	user v is engaged in no connection and therefore can accept a connection from user u.
${}^{M}\mathsf{userNotAvail}(u, v)$	user v is engaged in some connection and therefore cannot accept a connection from user u.
initial mode: any	

There is no guaranteed behaviour yet, except that the modes partition the state space.

The modes of mode classes ${}^{C}\mathsf{connectionDesignResources}(u, v)$ 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.

Table A.24: mode classes ${}^{C}\mathsf{connectionDesignResources}(u, v)$

for all $u, v \in {}^{t}\mathsf{userIds}$:

mode	**informal description**
${}^{M}\mathsf{connectionDesignResourceAvail}(u, v)$	there are sufficient lines, trunks, etc. to connect user u to user v.
${}^{M}\mathsf{connectionDesignResourceNotAvail}(u, v)$	there are not sufficient lines, trunks, etc. to connect user u to user v.
initial mode: any	

The corresponding mode transition relations will not be specified completely in the requirements document. The requirements document will specify only the minimal requirements on the resources, leaving non-deterministic choices. Only a concrete design will specify the system behaviour completely with respect to these mode classes, since it depends on the design of the network structure.

The minimal requirement does not demand that any resources are available, only that the modes partition the state space.

The modes of mode classes ${}^{C}\mathsf{connectionResources}(u, v)$ 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.

Table A.25: mode classes ${}^{C}\mathsf{connectionResources}(u, v)$

for all $u, v \in {}^{t}\mathsf{userIds}$:

mode	**informal description**
${}^{M}\mathsf{connectionResourceAvail}(u, v)$	sufficient resources for a connection from user u to user v are available.
${}^{M}\mathsf{connectionResourceNotAvail}(u, v)$	sufficient resources for a connection from user u to user v are not available.
initial mode: any	

The modes of mode classes ${}^{C}\mathsf{connectionPhase}(u, v)$ 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.

Table A.26: mode classes ${}^{C}\mathsf{connectionPhase}(u, v)$

for all $u, v \in {}^{t}\mathsf{userIds}$:

mode	informal description
${}^{M}\mathsf{conIdle}(u, v)$	there exists no connection between users u and v.
${}^{M}\mathsf{conSetup}(u, v)$	user v is being alerted of a connection request by user u.
${}^{M}\mathsf{conEstablished}(u, v)$	user u is engaged in a connection to user v and both can use it for communication; user u has initiated this connection.
${}^{M}\mathsf{conBothTeardown}(u, v)$	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.
${}^{M}\mathsf{conOrigTeardown}(u, v)$	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.
${}^{M}\mathsf{conTermTeardown}(u, v)$	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.
initial mode: any	

Table A.27: relations on mode classes ${}^{C}\mathsf{connectionResources}(u, v)$: 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.

$$\begin{aligned}
&\forall u \in {}^{m}\mathsf{users},\ v \in {}^{t}\mathsf{userIds} : \\
&{}^{M}\mathsf{connectionResourceNotAvail}(u, v) \Leftarrow \quad {}^{M}\mathsf{connectionDesignResourceNotAvail}(u, v) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee\ \mathrm{inmode}({}^{M}\mathsf{userNotAvail}(u, v))
\end{aligned}$$

The mode transition relations for ${}^{C}\mathsf{connectionPhase}(u, v)$ describe the changes between the connection phases.

Table A.28: mode transition relations ${}^{T}\mathsf{connectionPhaseTrans}(u, v)$

$\forall u \in {}^{m}\mathsf{users},\ v \in {}^{t}\mathsf{userIds}$:
${}^{T}\mathsf{connectionPhaseTrans}(u, v) =$

current mode	**conditions** ${}^{m}\mathsf{connectReq}(u,v)$	inmode(${}^{M}\mathsf{connection\text{-}ResourceAvail}(u,v)$)	${}^{m}\mathsf{connectRsp}(v)$	**next mode**
${}^{M}\mathsf{conIdle}(u,v)$	@T	t′	–	${}^{M}\mathsf{conSetup}(u,v)$
	@T	f′	–	${}^{M}\mathsf{conOrigTeardown}(u,v)$
${}^{M}\mathsf{conSetup}(u,v)$	–	T	@T	${}^{M}\mathsf{conEstablished}(u,v)$
	@F	$*$	–	${}^{M}\mathsf{conIdle}(u,v)$
	–	@F	$*$	${}^{M}\mathsf{conOrigTeardown}(u,v)$
${}^{M}\mathsf{conEstablished}(u,v)$	–	$*$	@F	${}^{M}\mathsf{conOrigTeardown}(u,v)$
	@F	$*$	–	${}^{M}\mathsf{conTermTeardown}(u,v)$
	–	@F	–	${}^{M}\mathsf{conBothTeardown}(u,v)$
${}^{M}\mathsf{conOrigTeardown}(u,v)$	@F	$*$	–	${}^{M}\mathsf{conIdle}(u,v)$
${}^{M}\mathsf{conTermTeardown}(u,v)$	–	$*$	@F	${}^{M}\mathsf{conIdle}(u,v)$
${}^{M}\mathsf{conBothTeardown}(u,v)$	–	$*$	@F	${}^{M}\mathsf{conOrigTeardown}(u,v)$
	@F	$*$	–	${}^{M}\mathsf{conTermTeardown}(u,v)$

Please note that two users may try to connect to each other (nearly) simultaneously, creating a race condition. For example, when user v issues a connection request (and we are now additionally assuming the property ${}^{P}\mathsf{UserResSimple}$ from page 52), this implies that he cannot be connected to anymore, from that very moment on; compare the definition of ${}^{M}\mathsf{userNotAvail}(u, v)$ in property ${}^{P}\mathsf{UserResSimple}$). As a result of this race condition, both users may fail to connect to the other. Below, this situation is described by and included in the case

$$@\mathrm{T}({}^{m}\mathsf{connectReq}(u, v)) \land \mathrm{f}'({}^{M}\mathsf{connectionResourceAvail}(u, v))$$

Please note that we assume an interleaving interpretation for simultaneous value changes of (the binary) monitored variables. Please note also that this does not preclude (spontaneous) mode changes to occur simultaneously with variable value changes; therefore, the table below contains both "–" and "$*$" entries, as defined in Table A.2 on page 35.

The time functions ${}^{f}\mathsf{conTeardown}(u, v)$ 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 ${}^{T}\mathsf{teardownReasonTrans(uNotAvail)}(u)$ and

${}^{T}\mathsf{teardownReasonTrans(screened)}(u)$. Even though these functions are boolean-valued, we do not denote them as modes, since they are not part of a meaningful mode class.

Table A.29: the time functions ${}^{f}\mathsf{conTeardown}(u, v)$

$\forall u, v \in {}^{m}\mathsf{users}$: ${}^{f}\mathsf{conTeardown}(u, v) = \quad \mathrm{inmode}({}^{M}\mathsf{conOrigTeardown}(u, v))$ $\vee\ \mathrm{inmode}({}^{M}\mathsf{conBothTeardown}(u, v))$ $\vee\ \mathrm{inmode}({}^{M}\mathsf{conBothTeardown}(v, u))$ $\vee\ \mathrm{inmode}({}^{M}\mathsf{conTermTeardown}(v, u))$

Table A.30: the relations for the controlled variables ${}^{c}\mathsf{connectInd}(u)$, ${}^{c}\mathsf{connectCnf}(u)$, and ${}^{c}\mathsf{connectTeardownInd}(u)$.

$\forall u \in {}^{m}\mathsf{users}$: ${}^{c}\mathsf{connectInd}(u) = \exists v \in {}^{m}\mathsf{users} : (\ \mathrm{inmode}({}^{M}\mathsf{conSetup}(v, u))$ $\vee\ \mathrm{inmode}({}^{M}\mathsf{conEstablished}(v, u)))$
$\forall u \in {}^{m}\mathsf{users}$: ${}^{c}\mathsf{connectCnf}(u) = \exists v \in {}^{m}\mathsf{users} : \mathrm{inmode}({}^{M}\mathsf{conEstablished}(u, v))$
$\forall u \in {}^{m}\mathsf{users}$: ${}^{c}\mathsf{connectTeardownInd}(u) = \exists v \in {}^{m}\mathsf{users} : {}^{f}\mathsf{conTeardown}(u, v)$

Table A.31: the transition relations for the controlled variables ${}^{c}\mathsf{teardownReason(uNotAvail)}(u)$

$\forall u, v \in {}^{m}\mathsf{users}$:
${}^{T}\mathsf{teardownReasonTrans(uNotAvail)}(u) =$

current value	conditions			next value
	${}^{f}\mathsf{conTeardown}(u,v)$	$\mathrm{inmode}({}^{M}\mathsf{connectionDesign\text{-}ResourceAvail}(u,v))$	$\mathrm{inmode}({}^{M}\mathsf{userAvail}(u,v))$	
${}^{=}\mathsf{false}$	@T	T	f′	${}^{=}\mathsf{true}$
${}^{=}\mathsf{true}$	@F	$*$	$*$	${}^{=}\mathsf{false}$
initial value: ${}^{=}\mathsf{false}$				

A.3.3 Changeable Requirements for the Environment

A.3.3.1 ^{P}**ConReqInitIdle** – Connection Requests Start Idle

Relies on properties: ${}^{P}\mathsf{ConEnv}$

Table A.32: relations on initial values for ${}^{m}\mathsf{connectReq}(u, v)$ and ${}^{m}\mathsf{connectRsp}(u)$: no connection requests or responses in the beginning.

$\forall u, v \in {}^{p}\mathsf{initialUsers}$: $\neg{}^{m}\mathsf{connectReq}(u, v)$
$\forall u \in {}^{p}\mathsf{initialUsers}$: $\neg{}^{m}\mathsf{connectRsp}(u)$

A.3.4 Changeable Requirements for the System

A.3.4.1 ^{P}**ConInitIdle** – Connections Start Idle

Relies on properties: ${}^{P}\mathsf{ConReqInitIdle}$, ${}^{P}\mathsf{ConFun}$

Table A.33: relations on the initial modes of ${}^{C}\mathsf{connectionPhase}(u, v)$: all connections are idle in the beginning.

$$\forall u, v \in {}^{p}\mathsf{initialUsers} :$$
$$(t = 0) \Rightarrow \text{inmode}({}^{M}\mathsf{conIdle}(u, v))$$

A.3.4.2 ^{P}**UserOnlyHuman** – Only Humans Play User Roles

Relies on properties: ${}^{P}\mathsf{ConFun}$

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

Table A.34: relations on mode classes ${}^{C}\mathsf{connectionResources}(u, v)$: a user role may be played by humans only.

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

A.3.4.3 ^{P}**ClosedNet** – No Off-Network Connections

Relies on properties: ${}^{P}\mathsf{ConFun}$

Table A.35: relations on mode classes ${}^{C}\mathsf{connectionResources}(u, v)$: connections can be established with users of this network only.

$$\forall u \in {}^{m}\mathsf{users}, v \in {}^{t}\mathsf{userIds} :$$
$${}^{M}\mathsf{connectionResourceNotAvail}(u, v) \Leftarrow v \notin {}^{m}\mathsf{users}$$

A.3.4.4 ^{P}**DConResDesir** – Design Phase Connection Resources, Desirable

Relies on properties: ${}^{P}\mathsf{ConFun}$

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

Table A.36: relations on mode classes ${}^{C}\mathsf{connectionDesignResources}(u, v)$: it is preferred that many design resources are available.

$$\forall u, v \in {}^{m}\mathsf{users} : \\ {}^{M}\mathsf{connectionDesignResourceAvail}(u, v)$$

Remark: Peters [Pet98] currently works on *merit functions* which would allow to award numerical relative merits to different desired behaviours.

A.3.4.5 ^{P}**ConResSimple** – Simple Connection Resources

Relies on properties: ${}^{P}\mathsf{ConFun}$

Table A.37: relations on mode classes ${}^{C}\mathsf{connectionResources}(u, v)$: 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.

$$\begin{aligned} &\forall u \in {}^{m}\mathsf{users},\ v \in {}^{t}\mathsf{userIds} : \\ &{}^{M}\mathsf{connectionResourceNotAvail}(u, v) = \quad v \notin {}^{m}\mathsf{users} \\ &\qquad \vee\ {}^{M}\mathsf{connectionDesignResourceNotAvail}(u, v) \\ &\qquad \vee\ \mathrm{inmode}({}^{M}\mathsf{userNotAvail}(u, v)) \end{aligned}$$

A.3.4.6 ^{P}**UserResAvail** – Availability of User Resources

Relies on properties: ${}^{P}\mathsf{ConFun}$

This provides two auxiliary terms for further definitions:

Table A.38: time function ${}^{f}\mathsf{userConIdle}(u, v)$: are the connections from user u to user v and from v to u idle, from u's point of observation?

$$\begin{aligned} &\forall u, v \in {}^{m}\mathsf{users} : \\ &{}^{f}\mathsf{userConIdle}(u, v)) = \quad (\ \mathrm{inmode}({}^{M}\mathsf{conIdle}(u, v)) \\ &\qquad \vee\ \mathrm{inmode}({}^{M}\mathsf{conTermTeardown}(u, v))) \\ &\qquad \wedge\ (\ \mathrm{inmode}({}^{M}\mathsf{conIdle}(v, u)) \\ &\qquad \vee\ \mathrm{inmode}({}^{M}\mathsf{conOrigTeardown}(v, u))) \end{aligned}$$

Table A.39: time function ${}^f\mathsf{userConNotAvail}(u, v)$: does there exist another user w who already uses the connection resources of user v for a connection?

$$
\begin{array}{l}
\forall u, v \in {}^m\mathsf{users} : \\
{}^f\mathsf{userConNotAvail}(u, v)) = (\quad \forall h_u, h_v \in {}^m\mathsf{humans} : \\
\qquad\qquad (\quad (u \in {}^m\mathsf{userRoles}(h_u) \wedge v \in {}^m\mathsf{userRoles}(h_v)) \\
\qquad\qquad \Rightarrow (h_u = h_v)) \\
\qquad \vee\ \exists w \in {}^m\mathsf{users} : \\
\qquad\qquad w \neq u \wedge \neg\ {}^f\mathsf{userConIdle}(v, w))
\end{array}
$$

A.3.4.7 ^{P}**UserResSimple** – Simple User Resources

Relies on properties: ^{P}UserResAvail

Table A.40: relations on mode classes ${}^C\mathsf{userAvailable}(u, v)$: 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.

$$
\begin{array}{l}
\forall u, v \in {}^m\mathsf{users} : \\
\text{inmode}({}^M\mathsf{userNotAvail}(u, v)) = {}^f\mathsf{userConNotAvail}(u, v)
\end{array}
$$

A.3.5 Property Dependences

Figure A.3 on page 55 shows the uses-hierarchy among the properties of the requirements of the basic connection feature.

A.4 Voice Channel Feature

This feature provides only the very basics of a voice channel. This is sufficient for our case study, and it can be extended later when necessary.

For a discussion on separating connection control and the actual voice channels, see also the discussion in [Zav97].

A.4.1 Fundamental Requirements for the Environment

A.4.1.1 ^{P}**BinVoiceChanFun** – Two-Sided Voice Channel

Relies on properties: ^{P}UserFun, ^{P}DeviceFun

Table A.41: controlled variables for the voice channels

name	mon./ ctrl.	description	type
for all $u, v \in {}^t\mathsf{userIds}$: ${}^c\mathsf{voiceChannel}(u, v)$	ctrl.	binary signal: describing whether two users are connected by a voice channel, enabling them to converse in both directions	${}^t\mathsf{bool}$

For this kind of simple voice channel, we fundamentally require that two users taking part in the same side of a voice channel must be associated with the same device. Therefore, the implementer will have to provide a voice channel between two devices only, even if the constellation of users is more complex.

Table A.42: relations on ${}^m\mathsf{users}$ and ${}^m\mathsf{devices}$ for a ${}^c\mathsf{voiceChannel}$

$$
\begin{array}{l}
\forall u, v, w \in {}^m\mathsf{users} : \forall h_u, h_w \in {}^m\mathsf{humans} : \forall d_u, d_w \in {}^m\mathsf{devices} : \\
\quad (\ \ u \in {}^m\mathsf{userRoles}(h_u) \qquad\qquad \wedge w \in {}^m\mathsf{userRoles}(h_w) \\
\quad \wedge\ d_u \in {}^m\mathsf{possessedDevices}(h_u)\ \wedge d_w \in {}^m\mathsf{possessedDevices}(h_w) \\
\quad \wedge\ (\ \ ({}^c\mathsf{voiceChannel}(u, v) \wedge {}^c\mathsf{voiceChannel}(w, v)) \\
\qquad \vee\ ({}^c\mathsf{voiceChannel}(v, u) \wedge {}^c\mathsf{voiceChannel}(v, w)))) \\
\Rightarrow (d_u = d_w)
\end{array}
$$

A.4.2 Changeable Requirements for the System

A.4.2.1 ^{P}**BinVoiceChanSimple** – Simple Voice Channel

Relies on properties: ${}^P\mathsf{BinVoiceChanFun}$, ${}^P\mathsf{ConFun}$

This property provides only a very simple and basic relation between a voice channel and a connection. Later, we could have no voice channel for some connections, but video channels instead. Also, we could suspend a voice channel temporarily, without terminating the connection.

Table A.43: relations on the controlled variables ${}^c\mathsf{voiceChannel}$

$$
\begin{array}{l}
\forall u, v \in {}^m\mathsf{users} : \\
{}^c\mathsf{voiceChannel}(u, v) = \mathrm{inmode}({}^M\mathsf{conEstablished}(u, v))
\end{array}
$$

A.4.3 Property Dependences

Figure A.4 on page 55 shows the uses-hierarchy among the properties of the requirements of the voice channel feature.

A.5 Screening Fundamentals Feature

This feature provides the fundamental “hooks” for any future connection screening features. It does not provide any user visible service itself, except of the introduction of a new controlled variable.

There are no additional requirements for the environment.

A.5.1 Fundamental Requirements for the System

A.5.1.1 ^{P}**ConScreenFun** – Connection Screening

Relies on properties: ^{P}UserResAvail

Replaces properties: ^{P}UserResSimple

Table A.44: controlled variables for connection screening

name	**mon./ ctrl.**	**description**	**type**
for all $u \in {}^{t}$userIds : ^{c}teardown-Reason(screened)(u)	ctrl.	binary signal: if the current connection (request) of user u is torn down, and the other user is not available for a connection, this is due to a failed screening.	^{t}bool

The modes of mode classes ^{C}connectionScreen(u, v) describe whether all connection setup screening conditions for a connection from user u to user v are fulfilled, in particular either at the start of alerting v, or at the start of establishing a connection to v.

Table A.45: mode classes ^{C}connectionScreen(u, v)

for all $u, v \in {}^{t}$userIds :

mode	**informal description**
^{M}connectionScreenOk(u, v)	a connection from user u to user v is allowed.
^{M}connectionScreenBlock(u, v)	a connection from user u to user v is not allowed.
initial mode: any	

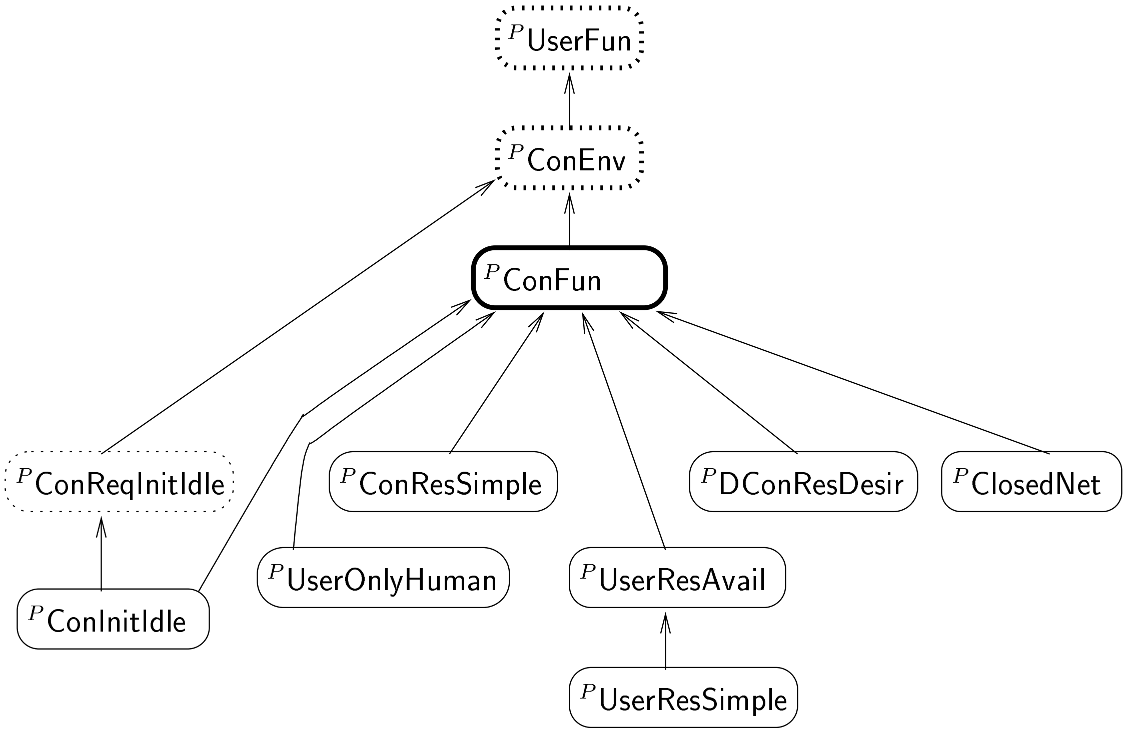

Figure A.3: The uses-hierarchy among the properties of the basic connection feature.

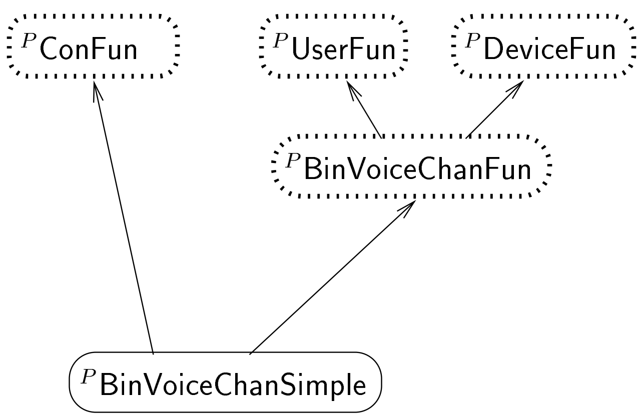

Figure A.4: The uses-hierarchy among the properties of the voice channel feature.

Depending on the added features, the modes of mode class ${}^{C}\mathsf{connectionScreen}(u, v)$ might change arbitrarily over time. There is *no guaranteed restriction of the behaviour* for these mode classes. In particular, this means that we cannot rely on that a connection will be established eventually. Similarly, there is no guarantee that anybody will be notified of a screening event.

The next property is a partial replacement for ${}^{P}\mathsf{UserResSimple}$. The changeable property ${}^{P}\mathsf{UserResScreen}$ below will complete the replacement.

Table A.46: relations on mode classes ${}^{C}\mathsf{userAvailable}(u, v)$: a user v is not available for a user u if the connection screening has objections.

$$\forall u, v \in {}^{m}\mathsf{users} : \\ \mathrm{inmode}({}^{M}\mathsf{userNotAvail}(u, v)) \Leftarrow \neg \mathrm{inmode}({}^{M}\mathsf{connectionScreenOk}(u, v))$$

A.5.2 Changeable Requirements for the System

A.5.2.1 ^{P}**UserResScreen** – User Resources with Screening

Relies on properties: ${}^{P}\mathsf{ConScreenFun}$

This property is an extension of ${}^{P}\mathsf{UserResSimple}$. Please note that the term $\mathrm{inmode}({}^{M}\mathsf{userNotAvail}(u, v))$, as defined in ${}^{P}\mathsf{UserResSimple}$, implies the term as defined here.

Table A.47: relations on mode classes ${}^{C}\mathsf{userAvailable}(u, v)$: a user v is not available for a user u if and only if there exists another user w who already uses the resources of v for a connection, or the connection screening has objections.

$$\forall u, v \in {}^{m}\mathsf{users} : \\ \mathrm{inmode}({}^{M}\mathsf{userNotAvail}(u, v)) = \quad {}^{f}\mathsf{userConNotAvail}(u, v) \\ \qquad \vee \neg\, \mathrm{inmode}({}^{M}\mathsf{connectionScreenOk}(u, v))$$

A.5.2.2 ^{P}**ConNoScreen** – No Connection Screening Yet

Relies on properties: ${}^{P}\mathsf{ConScreenFun}$

Table A.48: relations on ${}^{M}\mathsf{connectionScreenOk}(u,v)$: the screening shall not interfere with any connection setup.

$\forall u, v \in {}^{m}\mathsf{users}$: $\mathrm{inmode}({}^{M}\mathsf{connectionScreenOk}(u,v))$

Any modifications to this part are expected to set the modes such that they can be checked either immediately after a ${}^{m}\mathsf{connectReq}(u,v)$ signal or immediately after a ${}^{m}\mathsf{connectRsp}(v)$ signal. But it is admissible, too, that a screening condition turns ${}^{=}\mathsf{false}$ while two users already have a conversation. This is handled similar to a connection resource suddenly lost.

A.5.2.3 ^{P}ConScreenTell – Tell User that Screening Happened

Relies on properties: ^{P}ConScreenFun, ^{P}ConFun

Table A.49: the transition relations for the controlled variables ${}^{c}\mathsf{teardownReason(screened)}(u)$: the user is notified when screening happened

$\forall u, v \in {}^{m}\mathsf{users}$:
${}^{T}\mathsf{teardownReasonTrans(screened)}(u) =$

current value	conditions			next value
	${}^{f}\mathsf{conTeardown}(u,v)$	$\mathrm{inmode}({}^{M}\mathsf{connectionDesign\text{-}ResourceAvail}(u,v))$	$\mathrm{inmode}({}^{M}\mathsf{connection\text{-}ScreenOk}(u,v))$	
${}^{=}\mathsf{false}$	@T	T	f′	${}^{=}\mathsf{true}$
${}^{=}\mathsf{true}$	@F	$*$	$*$	${}^{=}\mathsf{false}$
initial value: ${}^{=}\mathsf{false}$				

A.5.3 Property Dependences

Figure A.5 on page 61 shows the uses-hierarchy among the properties of the requirements of the screening fundamentals feature.

A.6 Black List of Devices

This feature allows, for a telephone device, to disable incoming connections from specified other devices.

A.6.1 Fundamental Requirements for the Environment

A.6.1.1 ^{P}**BlacklistDevEnv** – Device Black List Environment

Relies on properties: ^{P}DeviceFun

Table A.50: monitored variables for the black list of devices

name	**mon./ ctrl.**	**description**	**type**
for all $d \in {}^{t}\mathsf{deviceIds}$: ${}^{m}\mathsf{blacklistDev}(d)$	mon.	the set of devices that are not allowed to have a connection to device d.	$\mathcal{P}({}^{t}\mathsf{deviceIds})$

A.6.2 Fundamental Requirements for the System

A.6.2.1 ^{P}**BlacklistDevFun** – Device Black List

Relies on properties: ^{P}BlacklistDevEnv

Replaces properties: ^{P}ConNoScreen

The following auxiliary time function translates device screening into user screening, in order to simplify some expressions:

Table A.51: time functions ${}^{f}\mathsf{blacklistDevUsers}(v)$: the set of users u that are not allowed to have a connection to user v.

$$
\begin{aligned}
&\forall v \in {}^{m}\mathsf{users} : \\
&\forall h_u, h_v \in {}^{m}\mathsf{humans} : \\
&\forall d_u, d_v \in {}^{m}\mathsf{devices} : \\
&{}^{f}\mathsf{blacklistDevUsers}(v) = \{u \mid \quad u \in {}^{m}\mathsf{userRoles}(h_u) \\
&\qquad \wedge\ d_u \in {}^{m}\mathsf{possessedDevices}(h_u) \\
&\qquad \wedge\ d_u \in {}^{m}\mathsf{blacklistDev}(d_v) \\
&\qquad \wedge\ d_v \in {}^{m}\mathsf{possessedDevices}(h_v) \\
&\qquad \wedge\ v \in {}^{m}\mathsf{userRoles}(h_v)\}
\end{aligned}
$$

Table A.52: relations on ${}^{M}\mathsf{connectionScreenOk}(u, v)$: no user u associated with a device on the black lists that are associated with user v may have a connection to v.

$$\forall u, v \in {}^{m}\mathsf{users} :$$
$$\neg(\text{inmode}({}^{M}\mathsf{connectionScreenOk}(u, v)) \wedge u \in {}^{f}\mathsf{blacklistDevUsers}(v))$$

A.6.3 Changeable Requirements for the Environment

A.6.3.1 ^{P}**BlacklistDevNoConSet** – Change Device Black List only when no Connection

Relies on properties: ^{P}BlacklistDevEnv, ^{P}ConFun

This property allows for cheaper implementations, where the signalling capabilities are limited during a connection.

Table A.53: relations on ${}^{m}\mathsf{blacklistDev}(d)$: the device black list may change only when the device is not involved in a connection.

$$\begin{array}{l}
\forall u, v \in {}^{m}\mathsf{users} : \\
\forall h_u \in {}^{m}\mathsf{humans} : \\
\forall d_u \in {}^{m}\mathsf{devices} : \\
\quad (\ \ u \in {}^{m}\mathsf{userRoles}(h_u) \\
\quad \wedge\ d_u \in {}^{m}\mathsf{possessedDevices}(h_u) \\
\quad \wedge\ {}^{m}\mathsf{blacklistDev}(d_u) \neq {}^{m}\mathsf{blacklistDev}(d_u)') \\
\Rightarrow {}^{f}\mathsf{userConIdle}(u, v)
\end{array}$$

A.6.4 Changeable Requirements for the System

A.6.4.1 ^{P}**BlacklistDevOnly** – Device Black List is the only Screening Feature

Relies on properties: ^{P}BlacklistDevFun

Table A.54: relations on ${}^{M}\mathsf{connectionScreenOk}(u, v)$: there is no screening except when ^{P}BlacklistDevFun says so.

$$\forall u, v \in {}^{m}\mathsf{users} :$$
$$\neg\text{inmode}({}^{M}\mathsf{connectionScreenOk}(u, v)) = u \in {}^{f}\mathsf{blacklistDevUsers}(v)$$

Please note that this property implies property ^{P}BlacklistDevFun.

A.6.5 Property Dependences

Figure A.6 on page 61 shows the uses-hierarchy among the properties of the requirements of the device black list feature.

A.7 Black List of Users

This feature allows, for a telephone user, to disable incoming connections from specified other users. In contrast to the previous feature, this feature distinguishes different users at the same telephone device. We deliberately investigate how our approach can handle similar but slightly distinct features.

A.7.1 Fundamental Requirements for the Environment

A.7.1.1 $^{P}\mathsf{BlacklistUsrEnv}$ – User Black List Environment

Relies on properties: $^{P}\mathsf{UserFun}$

Table A.55: monitored variables for the black list of users

name	mon./ ctrl.	description	type
for all $u \in {}^{t}\mathsf{userIds}$: $^{m}\mathsf{blacklistUsr}(v)$	mon.	the set of users that are not allowed to have a connection to user v.	$\mathcal{P}({}^{t}\mathsf{userIds})$

A.7.2 Fundamental Requirements for the System

A.7.2.1 $^{P}\mathsf{BlacklistUsrFun}$ – User Black List

Relies on properties: $^{P}\mathsf{BlacklistUsrEnv}$

Replaces properties: $^{P}\mathsf{ConNoScreen}$

Table A.56: relations on $^{M}\mathsf{connectionScreenOk}(u, v)$: no user u on the black list of user v may have a connection to v.

$$\forall u, v \in {}^{m}\mathsf{users} :$$
$$\neg(\mathrm{inmode}({}^{M}\mathsf{connectionScreenOk}(u, v)) \wedge u \in {}^{m}\mathsf{blacklistUsr}(v))$$

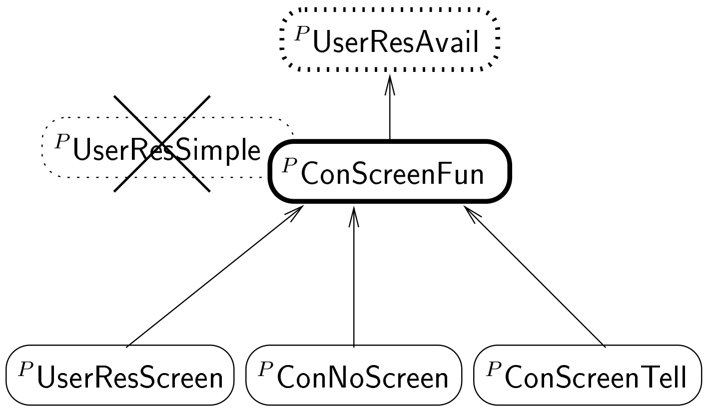

Figure A.5: The uses-hierarchy among the properties of the screening fundamentals feature.

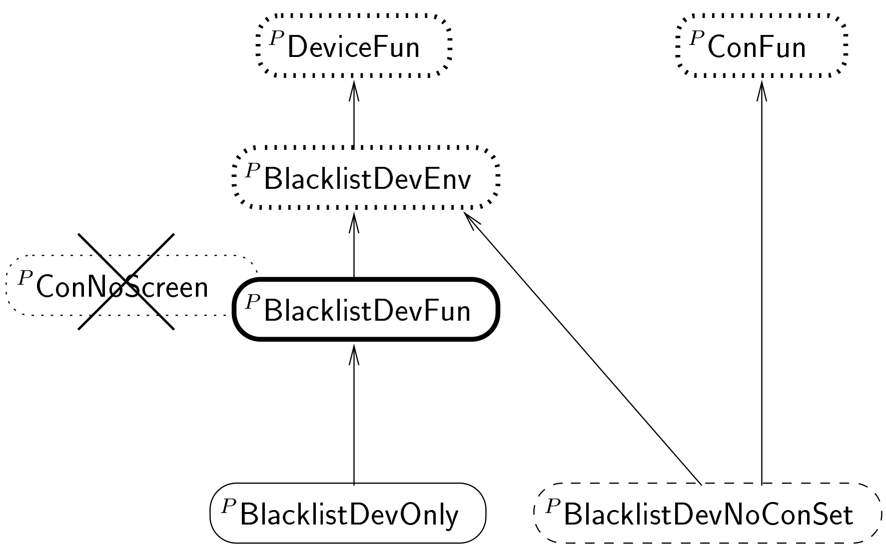

Figure A.6: The uses-hierarchy among the properties of the device black list feature.

A.7.3 Changeable Requirements for the Environment

A.7.3.1 ^{P}**BlacklistUsrNoConSet** – Change User Black List only when no Connection

Relies on properties: $^{P}\mathsf{BlacklistUsrEnv}$, $^{P}\mathsf{ConFun}$

This property allows for cheaper implementations, where the signalling capabilities are limited during a connection.

Table A.57: relations on $^{m}\mathsf{blacklistUsr}(u)$: the user black list may change only when the user is not involved in a connection.

$$\forall u, v \in {}^{m}\mathsf{users} :$$
$$^{m}\mathsf{blacklistUsr}(u) \neq {}^{m}\mathsf{blacklistUsr}(u)' \Rightarrow {}^{f}\mathsf{userConIdle}(u, v)$$

A.7.4 Changeable Requirements for the System

A.7.4.1 ^{P}**BlacklistUsrOnly** – User Black List is the only Screening Feature

Relies on properties: $^{P}\mathsf{BlacklistUsrFun}$

Table A.58: relations on $^{M}\mathsf{connectionScreenOk}(u, v)$: there is no screening except when $^{P}\mathsf{BlacklistUsrFun}$ says so.

$$\forall u, v \in {}^{m}\mathsf{users} :$$
$$\neg \mathrm{inmode}({}^{M}\mathsf{connectionScreenOk}(u, v)) = u \in {}^{m}\mathsf{blacklistUsr}(v)$$

Please note that this property implies property $^{P}\mathsf{BlacklistUsrFun}$.

A.7.5 Property Dependences

Figure A.7 on page 64 shows the uses-hierarchy among the properties of the requirements of the user black list feature.

A.8 FI Detection and Resolution for the Black Lists A.6 and A.7

When composing the entire set of features, a simple syntactic check reveals that there exists a property that is replaced twice by another property. Both property $^{P}\mathsf{BlacklistDevFun}$ and property ^{P}BlacklistUsrFun replace property ^{P}ConNoScreen. This indicates a possible feature interaction.

A manual analysis shows immediately that these two properties are not contradictory. Both features are quite similar. The device black list may be easier to implement, if user identification is difficult, and the user black list has a higher usage value. It is up to the customer whether he deploys one, the other, or both features.

Further analysis of the identified features shows that the properties ^{P}BlacklistDevOnly and ^{P}BlacklistUsrOnly are contradictory. (It is not possible to satisfy both in the general case, when we keep in mind that this must not restrict the monitored variables, in particular the values for the black lists.) This feature interaction is a clear conflict between the intents of these properties: both properties state that its feature is the only screening feature.

But this feature interaction is resolved easily, since both conflicting properties are marked as changeable, and since the necessary replacement property is obvious:

A.8.1 Changeable Requirements for the System

A.8.1.1 ^{P}BlacklistDevUsrOnly – Device and User Black Lists are the only Screening Features

Replaces properties: ^{P}BlacklistDevOnly, ^{P}BlacklistUsrOnly

Table A.59: relations on $^M\mathsf{connectionScreenOk}(u, v)$: there is no screening except when ^{P}BlacklistDevFun or ^{P}BlacklistUsrFun say so.

$$\begin{aligned}
&\forall u, v \in {}^m\mathsf{users} : \\
&\quad \neg\mathrm{inmode}({}^M\mathsf{connectionScreenOk}(u, v)) \\
&= u \in {}^f\mathsf{blacklistDevUsers}(v) \vee u \in {}^m\mathsf{blacklistUsr}(v)
\end{aligned}$$

Please note that this property implies both properties ^{P}BlacklistDevFun and ^{P}BlacklistUsr-Fun; the feature interaction resolution does not violate the fundamental properties of the involved features.

A.8.2 Property Dependences

Figure A.8 on page 64 shows the uses-hierarchy among the properties of the requirements of the black lists FI resolution feature.

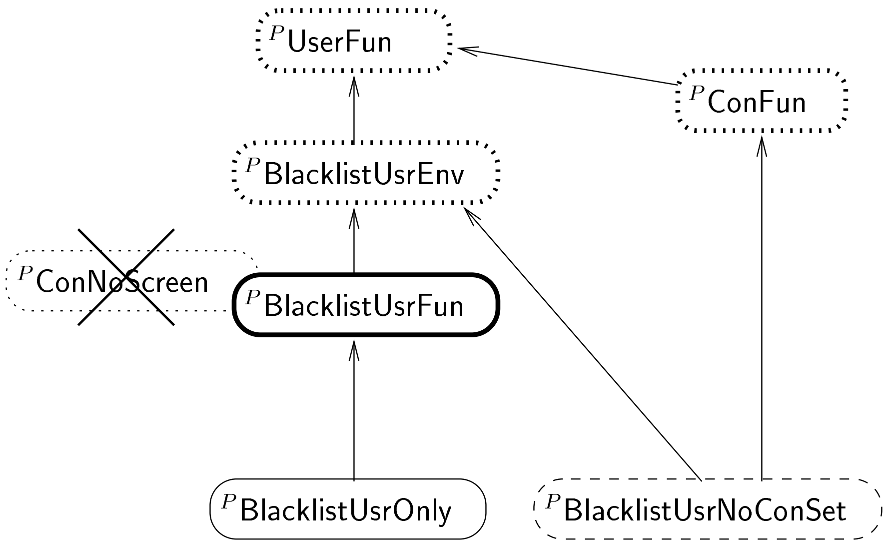

Figure A.7: The uses-hierarchy among the properties of the user black list feature.

Figure A.8: The uses-hierarchy among the properties of the black lists FI resolution feature.

A.9 Follow Human Connection Forwarding

This feature allows a human to move from one telephone device to another while continuously playing his associated user roles. The effect is that he can be reached at whichever device he currently is located.

In order to facilitate an implementation, we require from the environment that a human changes devices only when he is not involved in a connection. This is a changeable requirement.

The details of registering and de-registering the location of a human belong the the concern of the users' interface. Therefore, in the description of the feature's functionality we use only the associations of humans, devices, and users that we have introduced already as monitored variables.

Please note that this feature has no real system requirements part at all. Its fundamental property is that it removes the property that says that humans must not move at all. The details of moving are just a changeable property.

A.9.1 Fundamental Requirements for the System

A.9.1.1 ^{P}**FollowHumanFun** – Follow Human Forwarding

Replaces properties: ^{P}DevAssocNoRem

Table A.60: vacuously true system requirement that just discards the property ^{P}DevAssocNoRem.

$^{=}$true

A.9.2 Changeable Requirements for the Environment

A.9.2.1 ^{P}**FollowHumanEnv** – Follow Human Forwarding Environment

Relies on properties: ^{P}UserResAvail, ^{P}DeviceFun

Table A.61: relations on ${}^{m}\mathsf{possessedDevices}(h)$: telephone device associations are changed only when the human is not involved in a connection.

$$\begin{array}{l} \forall u, v \in {}^{m}\mathsf{users} : \\ \forall h_u \in {}^{m}\mathsf{humans} : \\ \quad (\ u \in {}^{m}\mathsf{userRoles}(h_u) \\ \quad \wedge {}^{m}\mathsf{possessedDevices}(h_u) \neq {}^{m}\mathsf{possessedDevices}(h_u)') \\ \Rightarrow {}^{f}\mathsf{userConIdle}(u, v) \end{array}$$

A.9.3 Property Dependences

Figure A.9 on page 73 shows the uses-hierarchy among the properties of the requirements of the follow human connection forwarding feature.

A.10 Transfer User Role to Another Human

This feature allows a human to tranfer one of his user roles to another human. The destination is determined by another user role (which might be a ${}^{=}\mathsf{privatePersonRole}(\mathrm{h})$ of a human h). The ${}^{=}\mathsf{privatePersonRole}(h)$ itself may never be transfered, as stated in property ${}^{P}\mathsf{HumanFun}$. If a transfer is done during a connection, the new human inherits the connection with all rights and obligations, possibly he becomes even the originating side of the connection. The transfer is only possible, in case a connection exists, if suitable resources are available, and if the transfered-to user accepts.

Please note that the changeable property ${}^{P}\mathsf{UserOneHuman}$ states that a user role may be played by at most one human. This is not invalidated by this feature. Even more, if the property ${}^{P}\mathsf{UserOneHuman}$ should be changed or dropped, this feature is not affected at all.

Please also note that this feature might be extended by several add-on features providing more convenience: for example, the ${}^{=}\mathsf{privatePersonRole}(h)$ of the transfering human h might be notified by an extra controlled variable of the success of the transfer. And if a connection exists, the connection partner may be put on hold first, in order to facilitate the signalling between the transfering user and the system.

This feature specifies mainly requirements for the environment, not for the system. The set of user roles for a human is a monitored variable. Therefore, any changes to this set are performed by the environment, and the system must be informed about them properly, only. The actual main obligation of the system is to rearrange the voice connections between telephone devices in order to reflect the rearranged user roles. But this duty of the system is already specified in the description of a connection in feature A.3. The enhancement which this feature provides is to relax the restrictions on user role changes.

A.10.1 Fundamental Requirements for the Environment

A.10.1.1 ^{P}UserTransferEnv – User Role Transfer Environment

Relies on properties: ^{P}ConFun

Replaces properties: ^{P}UserRolNoRem

Table A.62: monitored and controlled variables for user role transfer

name	mon./ ctrl.	description	type
for all $u, v \in {}^{t}$userIds : ^{m}userTransferReq(u, v)	mon.	binary command: user u requests a transfer of this user role to a human playing user role v	^{t}bool
for all $u \in {}^{t}$userIds : ^{m}userTransferRsp(u)	mon.	binary command: user u accepts an offered user role transfer (by @T)	^{t}bool
for all $u \in {}^{t}$userIds : ^{c}userTransferInd(u)	ctrl.	binary signal: the user u is offered a user role transfer	^{t}bool

The modes of mode classes ^{C}userTransferPhase(u, v) describe whether there is currently a user role transfer ongoing between the two users u and v.

Table A.63: mode classes ^{C}userTransferPhase(u, v)

for all $u, v \in {}^{t}$userIds :

mode	informal description
^{M}userTransferIdle(u, v)	no user role transfer is ongoing between users u and v.
^{M}userTransferSetup(u, v)	user v is being alerted of a user role transfer request by user u.
^{M}userTransferTeardown(u, v)	a user role transfer of user u to user v has been denied for some reason, and user u must still acknowledge this.
initial mode: any	

The modes of mode classes ^{C}userTransferResources(u, v) describe whether there are sufficient of the diverse resources available to perform a user role transfer from user u to user v.

Table A.64: mode classes ${}^{C}\mathsf{userTransferResources}(u, v)$

for all $u, v \in {}^{t}\mathsf{userIds}$:

mode	informal description
${}^{M}\mathsf{userTransferResourceAvail}(u, v)$	sufficient resources for a user role transfer from user u to user v are available.
${}^{M}\mathsf{userTransferResourceNotAvail}(u, v)$	sufficient resources for a user role transfer from user u to user v are not available.
initial mode: any	

The mode transition relations for ${}^{C}\mathsf{userTransferPhase}(u, v)$ describe the changes between the user role transfer phases.

Table A.65: mode transition relations ${}^{T}\mathsf{userTransferPhaseTrans}(u, v)$

$\forall u \in {}^{m}\mathsf{users},\ v \in {}^{t}\mathsf{userIds}$:
${}^{T}\mathsf{userTransferPhaseTrans}(u, v) =$

current mode	conditions: ${}^{m}\mathsf{userTransferReq}(u, v)$	conditions: $\mathrm{inmode}({}^{M}\mathsf{userTransferResourceAvail}(u, v))$	conditions: ${}^{m}\mathsf{userTransferRsp}(v)$	next mode
${}^{M}\mathsf{userTransferIdle}(u, v)$	@T	t′	–	${}^{M}\mathsf{userTransferSetup}(u, v)$
	@T	f′	–	${}^{M}\mathsf{userTransferTeardown}(u, v)$
${}^{M}\mathsf{userTransferSetup}(u, v)$	$*$	T	@T	${}^{M}\mathsf{userTransferIdle}(u, v)$
	@F	$*$	–	${}^{M}\mathsf{userTransferIdle}(u, v)$
	–	@F	$*$	${}^{M}\mathsf{userTransferTeardown}(u, v)$
${}^{M}\mathsf{userTransferTeardown}(u, v)$	@F	$*$	–	${}^{M}\mathsf{userTransferIdle}(u, v)$

The following table states, among others, that a user role transfer must be preceded by a proper request/response of the involved users, but only if some non-idle connection is affected. This means that a human will know if a non-idle user is transfered to him. The case of idle users is left open here in the fundamental requirements. The second major statement is that suitable resources must be available for a transfer to take place.

Table A.66: the relations for the monitored variables ${}^{m}\mathsf{userRoles}(h)$, ${}^{m}\mathsf{userTransferReq}(u,v)$, and ${}^{m}\mathsf{userTransferRsp}(u)$.

$$
\begin{array}{l}
\forall u, w \in {}^{m}\mathsf{users} : \\
\forall h_u, h_v \in {}^{m}\mathsf{humans} : \\
\exists v \in {}^{m}\mathsf{users} : \\
\quad (\ \ @\mathrm{F}(u \in {}^{m}\mathsf{userRoles}(h_u)) \wedge @\mathrm{T}(u \in {}^{m}\mathsf{userRoles}(h_v))) \\
\Rightarrow (\ \ \neg(\mathrm{inmode}({}^{M}\mathsf{conEstablished}(u,w)) \vee \mathrm{inmode}({}^{M}\mathsf{conEstablished}(w,u))) \\
\quad \vee (\ \ @\mathrm{F}(\mathrm{inmode}({}^{M}\mathsf{userTransferSetup}(u,v))) \\
\qquad \wedge @\mathrm{T}(\mathrm{inmode}({}^{m}\mathsf{userTransferRsp}(v))) \\
\qquad \wedge \mathrm{T}(v \in {}^{m}\mathsf{userRoles}(h_v))))
\end{array}
$$

$$
\begin{array}{l}
\forall u, v \in {}^{m}\mathsf{users} : \\
\forall h_u, h_v \in {}^{m}\mathsf{humans} : \\
\quad \mathrm{T}(\mathrm{inmode}({}^{M}\mathsf{userTransferResourceAvail}(u,v))) \\
\vee \neg(@\mathrm{F}(u \in {}^{m}\mathsf{userRoles}(h_u)) \wedge @\mathrm{T}(u \in {}^{m}\mathsf{userRoles}(h_v)) \wedge v \in {}^{m}\mathsf{userRoles}(h_v))
\end{array}
$$

A.10.2 Fundamental Requirements for the System

A.10.2.1 ${}^{P}\mathsf{UserTransferFun}$ – User Role Transfer Fundamentals

Relies on properties: ${}^{P}\mathsf{UserTransferEnv}$

The modes of mode classes ${}^{C}\mathsf{userTransferDesignResources}(u,v)$ describe whether there are sufficient resources such as signalling lines available to perform a user transfer. These are the resources that will be introduced by design decisions only, and which are not part of the requirements specification.

Table A.67: mode classes ${}^{C}\mathsf{userTransferDesignResources}(u,v)$

for all $u, v \in {}^{t}\mathsf{userIds}$:

mode	**informal description**
${}^{M}\mathsf{userTransferDesignResourceAvail}(u,v)$	there are sufficient signalling lines etc. to transfer user role u to user role v.
${}^{M}\mathsf{userTransferDesignResourceNotAvail}(u,v)$	there are not sufficient signalling lines etc. to transfer user role u to user role v.
initial mode: any	

The corresponding mode transition relations will not be specified completely in the requirements document. The requirements document will specify only the minimal requirements on the resources, leaving non-deterministic choices. Only a concrete design will specify the system behaviour completely with respect to these mode classes, since it depends on the design of the network structure.

The minimal requirement does not demand that any resources are available, only that the modes partition the state space.

Table A.68: the relations for the controlled variables $^{c}\mathsf{userTransferInd}(u)$.

$\forall u \in {}^{m}\mathsf{users}$: $^{c}\mathsf{userTransferInd}(u) = \exists v \in {}^{m}\mathsf{users} : \mathrm{inmode}(^{M}\mathsf{userTransferSetup}(v, u))$

A.10.3 Changeable Requirements for the Environment

A.10.3.1 ^{P}**UTransferReqInitIdle** – User Transfer Requests Start Idle

Relies on properties: $^{P}\mathsf{UserTransferEnv}$

Table A.69: relations on initial values for $^{m}\mathsf{userTransferReq}(u, v)$ and $^{m}\mathsf{userTransferRsp}(u)$: no user transfer requests or responses in the beginning.

$\forall u, v \in {}^{p}\mathsf{initialUsers}$: $\neg^{m}\mathsf{userTransferReq}(u, v)$
$\forall u \in {}^{p}\mathsf{initialUsers}$: $\neg^{m}\mathsf{userTransferRsp}(u)$

A.10.3.2 ^{P}**UserRolPreserve** – User Roles are Preserved

Relies on properties: $^{P}\mathsf{UserFun}$

User roles cannot be lost or newly introduced.

Table A.70: the relations for the monitored variables $^{m}\mathsf{userRoles}(h)$.

$\forall u \in {}^{m}\mathsf{users}$: $\quad(\exists h_u \in {}^{m}\mathsf{humans} : @\mathrm{F}(u \in {}^{m}\mathsf{userRoles}(h_u)))$ $\Leftrightarrow (\exists h_v \in {}^{m}\mathsf{humans} : @\mathrm{T}(u \in {}^{m}\mathsf{userRoles}(h_v)))$

A.10.3.3 ^{P}**UserRolNoTransferIdle** – User Roles not Transfered when Idle

Relies on properties: $^{P}\mathsf{ConFun}$

User roles may be transfered only when some connection exists. (In this case, the receiving human is asked. Otherwise, he wouldn't necessarily.)

Table A.71: the relations for the monitored variables ${}^{m}\mathsf{userRoles}(h)$.

$\forall u, w \in {}^{m}\mathsf{users}$: $\forall h_u, h_v \in {}^{m}\mathsf{humans}$: $\quad(\text{inmode}({}^{M}\mathsf{conEstablished}(u, w)) \vee \text{inmode}({}^{M}\mathsf{conEstablished}(w, u)))$ $\vee \neg(@\mathrm{F}(u \in {}^{m}\mathsf{userRoles}(h_u)) \wedge @\mathrm{T}(u \in {}^{m}\mathsf{userRoles}(h_v)))$

A.10.4 Changeable Requirements for the System

A.10.4.1 ^{P}**UTransferInitIdle** – User Transfers Start Idle

Relies on properties: ${}^{P}\mathsf{UTransferReqInitIdle}$

Table A.72: relations on the initial modes of ${}^{C}\mathsf{userTransferPhase}(u, v)$: all user transfers are idle in the beginning.

$\forall u, v \in {}^{p}\mathsf{initialUsers}$: $(t = 0) \Rightarrow \text{inmode}({}^{M}\mathsf{userTransferIdle}(u, v))$

A.10.4.2 ^{P}**TransferClosedNet** – No Off-Network Transfers

Relies on properties: ${}^{P}\mathsf{UserTransferEnv}$

Table A.73: relations on mode classes ${}^{C}\mathsf{userTransferResources}(u, v)$: transfers can be performed to users of this network only.

$\forall u \in {}^{m}\mathsf{users}, v \in {}^{t}\mathsf{userIds}$: ${}^{M}\mathsf{userTransferResourceNotAvail}(u, v) \Leftarrow v \notin {}^{m}\mathsf{users}$

A.10.4.3 ^{P}**DUserTransferResDesir** – Design User Transfer Resources, Desirable

Relies on properties: ${}^{P}\mathsf{UserTransferFun}$

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

Table A.74: relations on mode classes ${}^{C}\mathsf{userTransferDesignResources}(u, v)$: it is preferred that many design user transfer resources are available.

$$\forall u, v \in {}^{m}\mathsf{users} : \\ {}^{M}\mathsf{userTransferDesignResourceAvail}(u, v)$$

Remark: Peters [Pet98] currently works on *merit functions* which would allow to award numerical relative merits to different desired behaviours.

A.10.4.4 ^{P}**UserTransferResSimple** – Simple User Transfer Resources

Relies on properties: ${}^{P}\mathsf{UserTransferFun}$

Table A.75: relations on mode classes ${}^{C}\mathsf{userTransferResources}(u, v)$: the user role to connect to must be valid, the user must not currently be asked for another transfer already, and the resources provided by design decisions must be sufficent.

$$\begin{aligned} &\forall u \in {}^{m}\mathsf{users},\ v \in {}^{t}\mathsf{userIds} : \\ &{}^{M}\mathsf{userTransferResourceNotAvail}(u, v) = \quad v \notin {}^{m}\mathsf{users} \\ &\qquad\qquad \vee\ {}^{M}\mathsf{userTransferDesignResourceNotAvail}(u, v) \\ &\qquad\qquad \vee\ {}^{c}\mathsf{userTransferInd}(v) \end{aligned}$$

A.10.5 Property Dependences

Figure A.10 on page 73 shows the uses-hierarchy among the properties of the requirements of the user transfer feature.

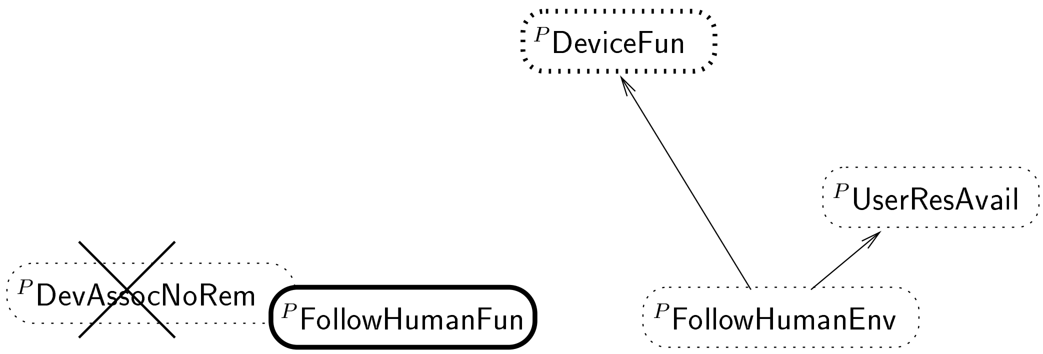

Figure A.9: The uses-hierarchy among the properties of the follow human connection forwarding feature.

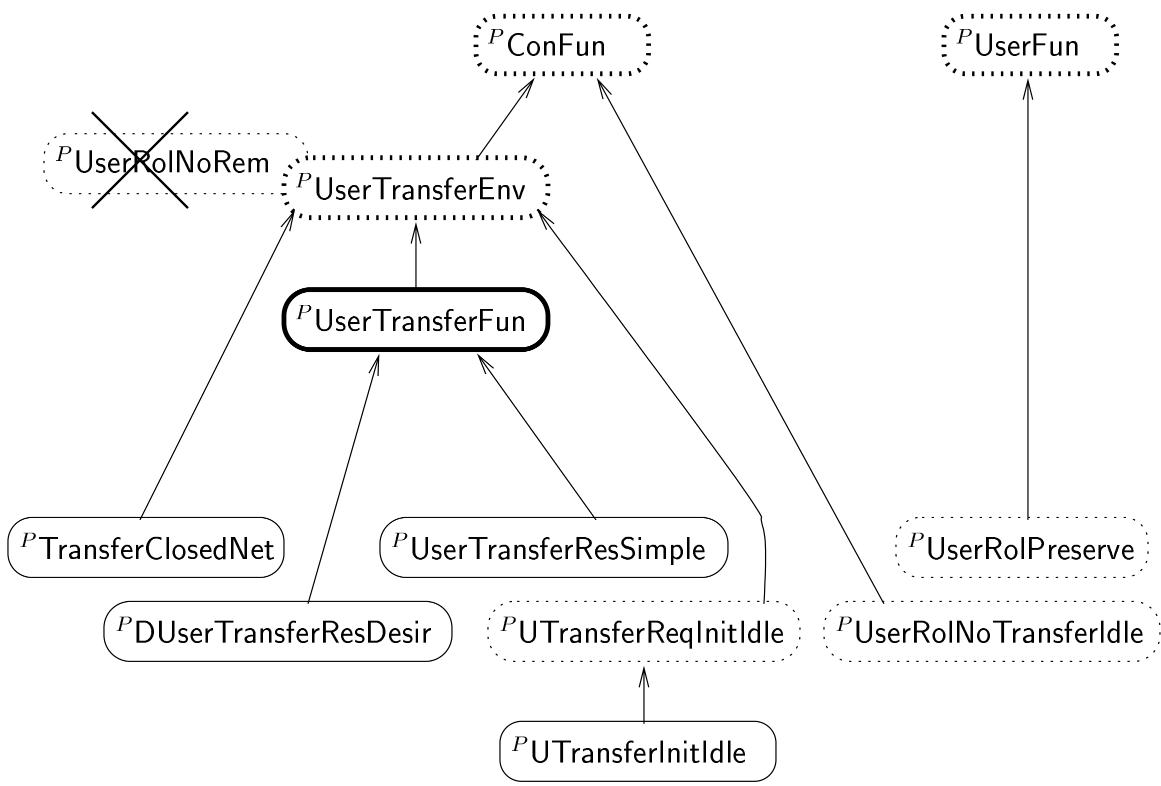

Figure A.10: The uses-hierarchy among the properties of the user transfer feature.

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.

The index allows for a coarse syntax check of the document, since there should be exactly one definition for each term (except for ^{t}bool).

Bibliography

[Ab+97] Abarca, C. et al.. *Service architecture*. Deliverable, TINA-Consortium, URL: http://www.tinac.com/ (16 June 1997). Version 5.0.

[AbLa95] Abadi, M. and Lamport, L. *Conjoining specifications*. ACM Trans. Prog. Lang. Syst. **17**(3), 507–534 (May 1995).

[Abr96] Abrial, J.-R. *The B Book: Assigning Programs to Meaning*. Cambridge University Press (1996).

[Abr97] Abraham, R. *Evaluating generalized tabular expressions in software documentation*. CRL Report 346, McMaster University, Hamilton, Canada (Feb. 1997).

[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.

[Bac90] Back, R. J. R. *Refinement calculus, part II: Parallel and reactive programs*. In de Bakker et al. [dBdRR90], pp. 67–93.

[BaSe96] Back, R. J. R. and Sere, K. *Superposition refinement of reactive systems*. Formal Aspects of Computing, Springer **8**(3), 324–346 (1996).

[BaWr90] Back, R. J. R. and Wright, J. v. *Refinement calculus, part I: Sequential nondeterministic programs*. In de Bakker et al. [dBdRR90], pp. 42–66.

[BBB+85] Bauer, F. L., Berghammer, R., Broy, M., Dosch, W., Geiselbrechtinger, F., Gnatz, R., Hangel, E., Hesse, W., Krieg-Brückner, B., Laut, A., Matzner, T. A., Möller, B., Nickl, F., Partsch, H., Pepper, P., Samelson, K., Wirsing, M., and Wössner, H. *The Munich Project CIP. Volume I: The wide spectrum language CIP-L*. No. 183 in Lecture Notes in Computer Science. Springer (1985).

[BBK95] Blom, J., Bol, R., and Kempe, L. *Automatic detection of feature interactions in temporal logic*. In Cheng and Ohta [ChOh95], pp. 1–19.

[BhHe97] Bharadwaj, R. and Heitmeyer, C. *Model checking complete requirements specifications using abstraction.* Technical Report NRL/MR/5540–97-7999, Naval Research Lab., Washington DC (10 Nov. 1997).

[BMS98] Bruns, G., Mataga, P., and Sutherland, I. *Features as service transformers.* In Kimbler and Bouma [KiBo98], pp. 85–97.

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

[Bre96] Bredereke, J. *Detection of feature interactions in Intelligent Networks by verification.* Software Concepts & Tools **17**(3), 121–139 (1996).

[Bre97] Bredereke, J. *Communication Systems Design With Estelle – On Style, Efficiency, and Analysis.* PhD thesis, Univ. of Kaiserslautern, Shaker Verlag, Aachen, Germany (Aug. 1997). ISBN 3-8265-2764-X.

[Bre98] Bredereke, J. *Avoiding feature interactions in the users' interface.* In Kimbler and Bouma [KiBo98], pp. 305–317.

[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.

[CHJB95] Cattrall, D., Howard, G., Jordan, D., and Buf, S. *An interaction-avoiding call processing model.* In Cheng and Ohta [ChOh95], pp. 85–96.

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

[DBBMS+95] Desharnais, J., Belkhiter, N., Ben Mohamed Sghaier, S., Tchier, F., Jaoua, A., Mili, A., and Zaguia, N. *Embedding a demonic semilattice in a relation algebra.* Theoretical Comput. Sci. **149**, 333–360 (1995). Also in: Research Report DIUL-RR-9302, Département d'Informatique, Université Laval, Canada, 1993.

[dBdRR90] de Bakker, J. W., de Roever, W.-P., and Rozenberg, G., editors. *REX Workshop on Stepwise Refinement of Distributed Systems – Models, Formalisms, Correctness*, no. 430 in Lecture Notes in Computer Science, Mook, The Netherlands (1990). Springer.

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

[DDH83] Darlington, J., Dzida, W., and Herda, S. *The role of exursions in interactive systems.* Int'l. Jour. of Man-Machine Studies **18**(2), 101–112 (Feb. 1983).

[DTK94] Desharnais, J., Tchier, F., and Khédri, R. *Demonic relational semantics of sequential programs*. Technical Report DIUL-RR-9406, Département d'Informatique, Université Laval, Québec, QC, G1K 7P4, Canada (Dec. 1994).

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

[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).

[FLdM95] Farooqui, K., Logrippo, L., and de Meer, J. *The ISO reference model for Open Distributed Processing: an introduction*. Comp. Networks and ISDN Syst. **27**(8), 1215–1229 (July 1995).

[FoWa74] Foley, J. D. and Wallace, V. L. *The art of natural graphic man-machine conversation*. Proc. of the IEEE **62**(4), 462–471 (Apr. 1974).

[FvDFH90] Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. *Computer Graphics: Principles and Practice*. Addison-Wesley, 2nd ed. (1990).

[GBGO98] Griffeth, N., Blumenthal, R., Gregoire, J.-C., and Ohta, T. *Feature interaction detection contest*. In Kimbler and Bouma [KiBo98], pp. 327–359. URL: http://www.tts.lth.se/FIW98/contest.html.

[Gib97] Gibson, J. P. *Feature requirements models: Understanding interactions*. In Dini et al. [DBL97], pp. 46–60.

[GrVe94] Griffeth, N. D. and Velthuijsen, H. *The negotiating agents approach to runtime feature interaction resolution*. In Bouma and Velthuijsen [BoVe94], pp. 217–235.

[HBGL95] Heitmeyer, C., Bull, A., Gasarch, C., and Labaw, B. *SCR*: A toolset for specifying and analyzing requirements*. In "Proc. Ninth Annual Conf. on Computer Assurance – COMPASS'95", pp. 109–122, Gaithersburg, MD, USA (June 1995).

[Heh90] Hehner, E. *A practical theory of programming*. Sci. Comput. Programming **13**(2–3), 133–158 (1990).

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

[ITU93a] ITU-T, Recommendation Q.1213. *Global Functional Plane for Intelligent Network CS-1* (Mar. 1993).

[ITU93b] ITU-T. *Q.12xx-Series Intelligent Network Recommendations* (1993).

[ITU97] ITU-T. *Q.122x-Series Intelligent Network Recommendations for Capability Set 2* (1997).

[Jac83] Jacob, R. J. K. *Executable specifications for a human-computer interface*. In Janda, A., editor, “Proc. ACM CHI'83 Human Factors in Computing Systems Conf.”, pp. 28–34, Evanston, USA (12–15 Dec. 1983). ACM, North-Holland.

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

[Kec98] Keck, D. O. *Requirements and a proposal for the prevention of a class of service interactions in Intelligent Networks*. In Margaria, T. et al., editors, “Services And Visualization : Towards User-Friendly Design ; ACoS'98”, no. 1385 in LNCS, pp. 90–105. Springer (1998).

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

[Lam94] Lamport, L. *The temporal logic of actions*. ACM Trans. Prog. Lang. Syst. **16**(3), 872–923 (May 1994).

[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).

[Mor87] Morris, J. M. *A theoretical basis for stepwise refinement and the 2 programming calculus*. Sci. Comput. Programming **9**(3) (1987).

[Mor88] Morgan, C. C. *The specification statement*. ACM Trans. Prog. Lang. Syst. **10**(3), 403–419 (1988).

[Mor90] Morgan, C. C. *Programming from Specifications*. Prentice-Hall (1990).

[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).

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

[Par69] Parnas, D. L. *On the use of transition diagrams in the design of a user interface for an interactive computer system*. In “Proc. 24th National ACM Conf.”, pp. 379–385 (1969).

[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).

[Par98] Parnas, D. L. *Some perspective on refinement*. In Sekerinski, E. and Sere, K., editors, "Program Development by Refinement – Case Studies Using the B Method", pp. xix–xxiv. Springer (1998).

[Pet98] Peters, D. K. *Deriving Real-Time Monitors from System Requirements Documentation*. PhD thesis, McMaster Univ., Hamilton, Canada (Dec. 1998). *To appear.*

[Pnu92] Pnueli, A. *System specification and refinement in temporal logic*. In Shyamasundar, R., editor, "12th Conf. on Foundations of Software Technology and Theoretical Computer Science", no. 652 in Lecture Notes in Computer Science, New Dehli, India (1992). Springer.

[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.

[Shn97] Shneiderman, B. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 3rd ed. (1997).

[Thi82] Thimbleby, H. *Character level ambiguity: consequences for user interface design*. Int'l. Jour. of Man-Machine Studies **16**(2), 211–225 (Feb. 1982).

[vdL94] van der Linden, R. *Using an architecture to help beat feature interaction*. In Bouma and Velthuijsen [BoVe94], pp. 24–35.

[vS92a] 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 (Telecommunications Research Institute of Ontario), Hamilton, Canada (Feb. 1992).

[vS92b] 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.

[WeGr97] Weiss, M. and Gray, T. *Experiences with a service environment for distributed multimedia applications*. In Dini et al. [DBL97], pp. 242–253.

[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].

[ZWO+95] Zibman, I., Woolf, C., O'Reilly, P., Strickland, L., Willis, D., and Visser, J. *Minimizing feature interactions: an architecture and processing model approach*. In Cheng and Ohta [ChOh95], pp. 65–83.