Interner Bericht

Automata-Theoretic Criteria for Feature Interactions in Telecommunications Systems

Jan Bredereke

Nr. 273/95

December 1995

Fachbereich Informatik
Universität Kaiserslautern

Postfach 3049
D-67653 Kaiserslautern
Germany

E-mail: `bredereke@informatik.uni-kl.de`

Abstract

The feature interaction problem in telecommunications systems increasingly obstructs the evolution of such systems. We develop formal detection criteria which render a necessary (but less than sufficient) condition for feature interactions. It can be checked mechanically and points out all potentially critical spots. These have to be analysed manually. The resulting resolution decisions are incorporated formally. Some prototype tool support is already available. A prerequisite for formal criteria is a formal definition of the problem. Since the notions of feature and feature interaction are often used in a rather fuzzy way, we attempt a formal definition first and discuss which aspects can be included in a formalization (and therefore in a detection method). This paper describes on-going work.

Contents

List of Definitions with Symbols

List of Figures

List of Tables

1 Introduction

Telephone switching systems are a classical example for long-lived and perpetually evolving software in the telecommunications domain. The first software controlled switching exchanges essentially still provided the plain old telephone service (POTS). Step by step, new features have been added since then which were supposed to offer added value to the customer (e.g. by call forwarding) and/or to the service provider (e.g. by improved accounting).[1]

Up to now a large number of features has been developed (several hundred, [Bo$^+$89]), especially by the American telephone companies. Therefore the probability is high that augmenting such a system by one more feature will influence another feature, especially in an undesired way. This is called a feature interaction. Both the notion of feature and the notion of feature interaction are used quite fuzzily and informally most of the time. This does not make a solution of the problem easier. The formalization in this report is intended as an attempt to improve this situation.

Since we do not know how to *avoid* feature interaction problems in telecommunications systems altogether, we need to *detect* and *resolve* them ([BoVe94]). Detection of feature interactions may be done in different ways, for example by simulation or by online behaviour analysis, but we will concentrate on an off-line *verification* approach here. Usually, a verification approach works like this (compare, e.g., the contributions at the Feature Interaction Workshop at Amsterdam [BoVe94], the work of Faci [Fac95], and the SCORE project [SCO95]): we write a high-level, property oriented description for each feature and for the basic system (e.g., in temporal logic), and we write a lower-level, constructive[2] description for them (e.g., in Estelle or SDL). Then we define an "implements" relation and prove mathematically that the combined system built from the constructive description of the basic system and the added features still implements all high-level properties. If we cannot prove this "implements" relation, we may get hints instead in which situations the properties do not hold anymore, depending on the actual verification method used.

This approach surely is promising, and it needs to be studied further. But it also has some limitations and prerequisites. If the proof is done by logical reasoning, the work is painstaking even with tool support, and the method is only applicable to small systems. Symbolic model checking, used, e.g., in the tools of Holzmann (see, e.g., [HoPe94, LiLi94]), has brought some progress with this respect, lately. Furthermore, one can only verify those properties that are explicitly specified. And in general, it is practically impossible to formalize *all* expectations at the system which the service provider or the customer may have.

[1]Example: the features defined in the ITU-T recommendations for Intelligent Networks (IN) [ITU93b].

[2]"Constructive" means "executable" or that an implementation can be derived directly.

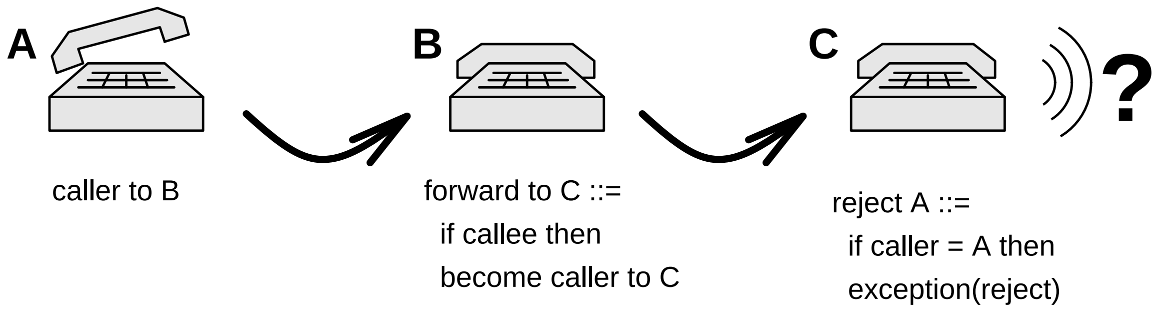

Figure 1: Feature interaction between call forwarding and terminating call screening.

Even worse, existing property descriptions may become “incomplete” or “inaccurate” by a later addition of a new feature: Imagine that you have expressed all relevant properties concerning a “caller” in a telephone system. Then, add a new call forwarding feature to this system. When a caller A calls a callee B, then B may forward this call to a user C (compare Figure 1). This way B may become a caller, too, in a sense. Who is the caller for C? Therefore, the notion of “caller” has become a little muddy, and our carefully written formulae on properties of a caller may not denote the meaning anymore which we wanted them to have.

In this report, we propose a different verification approach which may help finding feature interactions that could go unnoticed by other verification approaches. In Section 2, we propose formal definitions for the terms “feature” and “feature interaction”. In Section 3, we describe detection criteria for feature interactions; in Section 4, we sketch a way to resolve the detected interactions; in Section 5 we discuss briefly how our generic formalism can be mapped onto real formal description techniques (FDTs); in Section 6 we present the tools developed up to now and outline the practical applications done so far; and the report is concluded by a summary and an outlook.

2 Definition of “Feature Interaction”

2.1 What to Define and What to Leave Out

Before we tackle the feature interaction problem, we will give our definitions of the terms “feature” and “feature interaction”. As has been pointed out by Cameron and Velthuijsen [CaVe93], it is crucial to define the problem before trying to solve it. Our formal definitions surely do not cover all the aspects which are commonly associated with the informal terms, as no formal definition probably ever will be able to. But they state more clearly what the achievements are and which limitations have to be taken into account while they are used.

Cameron and Velthuijsen [CaVe93] describe two views on the problem. In the business view, “a feature is a tariffable unit”, and “a feature interaction occurs when the behaviour of one feature is altered by the use of another”. “A second kind of interaction occurs when the use of one feature should alter the behaviour of another, but does not.” In the implementor’s view, “a feature is any increment of functionality added to an existing system”. “Just as in the business view, a feature interaction occurs when one feature’s behaviour is altered by the use of a second.” Other authors provide similar descriptions, compare e.g. Kimbler and Velthuijsen [KiVe95, page 2] and Aho and Griffeth [AhGr95].

Even if there does not exist a clear (formal) concept of what a feature and a feature interaction are, the notion of “*behaviour* of a feature” is surely a central one. It is a necessary precondition for a feature interaction that this behaviour is changed. In this report, we disregard the case where an interaction occurs because a change in behaviour does *not* take place when a change in the intentions requires it to do so, since the notion of intended behaviour is still a topic of research ([CaVe93]). Maybe a property oriented approach (compare Section 1) will be able to handle it. In the remainder of this section, we will develop a formal definition which is based on the central notion of the change in behaviour.

When we have to decide if the behaviour is still the same as before, there are different aspects of the behaviour. First, there are the functional aspects. This concerns the sequences of possible states/events. But the non-functional aspects can become important, too. This concerns any real time or performance aspects. Also, there can be important properties on knowledge, e.g. security properties. Unfortunately, the formal treatment of non-functional aspects is much harder than that of functional aspects, and much less well understood [BCN95]. Therefore, we restrict our notion of feature interactions to functional aspects. This excludes, for example, the treatment of a case where one feature is impaired because another feature causes an overload on some shared resource.

We will not define what a service is. Commonly, it is some collection of features, but only if this collection is somehow self-contained. Since we will concentrate on functional aspects anyway, we have no need to take the pains with designing a formal definition of a service; we will stick to the task of defining a feature and its interactions.

For the formalization of the functional aspects of the behaviour, we advocate the use of automata [HoUl79]. The *computations* (execution sequences) of an automaton are a good and formal way to capture this kind of behaviour. There are many variants on the exact definitions; we will present a suitable one in the following.

2.2 Suitable System Architectures

We will decide first how we describe concurrency. One may choose to use a collection of explicitely concurrent communicating automata, or a single global automaton (comprising different local automata which, again, have some form of implicit concurrency and communication). If we take the first alternative, we still have to define formally what concurrency and communication is. Since a modelling of concurrency by interleaving is sufficient for our purposes, this question will inevitably lead to an answer that equals the second alternative. Therefore, and since the semantics of the standardized formal description technique Estelle ([ISO89]) is defined in a very similar way, we take the second alternative. As soon as we want to apply our results to practical system specifications, we have to transfer the results to practical specification languages; a similarity in the underlying semantics will facilitate this.

The computations of an automaton can be formalized by a tree of reachable states or by a set of linear sequences of reachable states. Since we are not interested in terminating computations, we take the simpler second alternative.

Existing telecommunications systems are huge and have a long history of development and uncounted modifications. In the effort of opening up these telephone systems to allow the addition of new features, even by third party providers, abstract models have been developed, e.g. the intelligent network (IN) conceptual model [ITU92, DuVi92] by the ITU-T. This model tries to formalize a basic telecommunications system on a very abstract level and includes only those aspects that are relevant for the addition of new features.

The basic call state machine (BCSM) in the IN conceptual model (INCM) is designed in a way that the control flow remains as much as possible in the BCSM. Among others, this results from the communication overhead involved in passing away the control. On the other hand, this system design results in a rather complex semantics of the *detection points* (DPs) where control may be given away. It turned out that such a complicated architecture obstructs the formal analysis of the system. Therefore, we use a simplified IN architecture. We do not use explicit detection points, we just specify a homogenous automaton, and extensions result in the addition of more transitions and states of the same kind as the old ones. A closer adaption to the IN architecture (or *of* the IN architecture?) is a topic for future research.

2.3 Basic Definitions

We start by defining our state space. Informally, the idea is that we have a fixed set of components which each have a fixed set of variables. This implies that we make two important simplifications:

- We disallow the dynamic creation and termination of components[3].

[3] In the FDT Estelle, one may do this with module instances, and in the FDT SDL, one may do this with processes.

- We disallow the dynamic allocation of local variables inside a component[4].

Therefore, we reduce the expressive power of our formalism. On the other hand, we trade in a much simpler formalism which seems us worth the sacrifice. Dynamic memory allocation makes it hard to talk about objects because they do not have static names. This complicates reasoning. And the dynamic creation and termination of components is just a special case of this, with the same problems. For the application area we have in mind, a set of pre-allocated module instances / processes will do just fine.

After we have demanded that the structure of the variable state inside a component be static, we also demand that it be *flat*, i.e. a straight tuple. (Example: no use of "records", just several individual variables.) From the point of usability, this is a reduction in readability, of course. But our formalism is only intended for the definition of a problem and for the description of a solution approach. If one wants to apply this approach, he needs to map it to his specific FDT. And then it is trivial to give a bijective mapping between a static hierarchical structure and a linear structure.

Definition 2.1 (local state) *A local state* is a (finite) tuple $s^l = (s_j)_{j=1}^m = (s_1, \ldots, s_m)$ of atomic states s_j, with $m \geq 1$.

Example 2.1.1 $s^l = (\text{waitForAck}, 42, \texttt{'abcdef'})$

Definition 2.2 (global state) *A global state* is a (finite) tuple $s^G = \left(s_i^l\right)_{i=1}^n = (s_1^l, \ldots, s_n^l)$ of local states s_i^l, with $n \geq 1$.

Example 2.2.1

$$\begin{aligned} s^G = \Big(& (\text{waitForAck}, 42, \texttt{'abcdef'}), \\ & (\text{connected}, 41, \texttt{'hijklm'}), \\ & (\text{idle}, 0)\Big) \end{aligned}$$

Definition 2.3 (global state space) *A global state space* S^G is a (possibly infinite) set of global states s^G which each have a fixed length n, and where each local state s_i^l, $1 \leq i \leq n$, has a fixed length m_i.

The restriction on the length n implies a static structure of components of the system, and the fixed length m_i implies a static structure of local variables inside a component.

[4]Estelle allows this.

Definition 2.4 (simple global automaton) *A simple global automaton* is a tuple $A^s = (s^G, T^s, s_0)$ with a global state space S^G, a (possibly infinite) set of simple transitions $T^s \subseteq S^G \times S^G$, and an initial state $s_0 \in S^G$.

This is our basic definition of a global automaton. Simple transitions have the disadvantage that their number can become infinite as soon as the domain of a variable is infinite, e.g., like for the natural numbers. Therefore, we allow to specify many simple transitions at the same time by a "transition". The number of these must be finite. The same concept to achieve a finite representation can be found in Estelle and SDL specifications, too. The most important example is the case when a transition only depends on the major state[5] of a module instance and not on the extended state space, i.e., there is a simple transition for every value combination of the extended state space.

Definition 2.5 (transition) *A transition t of a simple automaton $A^s =$ (S^G, T^s, s_0)* is a tuple $t = (S^e, \sigma, \lambda)$ with a set of enabled states $S^e \subseteq S^G$, a state transition relation $\sigma \subseteq S^G \times S^G$ such that for all $s_1 \in S^e$ holds $\exists s_2 \in S^G : s_1 \, \sigma \, s_2$, and a transition name λ.

Now we can define a global automaton which has a finite representation of its transitions.

Definition 2.6 (global automaton) *A global automaton* is a tuple $A =$ (S^G, T, s_0) where $A^s = (S^G, T^s, s_0)$ is a simple global automaton and T is a non-empty, finite set of transitions of A^s with different transition names λ^t for every transition $t \in T$ such that T^s equals the union

$$\bigcup_{(S^e_t, \sigma_t, \lambda_t) \in T} \sigma_t \cap (S^e_t \times S^G)$$

of all state transition relations σ_t.

Note that two transitions may have the same set of enabled states, and they may have the same state transition relation, but they must not have the same name.

Definition 2.7 (computation) *A computation c of a global automaton A* is a (possibly infinite) sequence of states $\langle s_i \rangle_{i=0} = \langle s_0, s_1, \ldots \rangle$ which starts always with the initial state s_0 of A and where for all $s_i, s_{i+1} \in c$, $i \geq 0$, holds that $(s_i, s_{i+1}) \in T^s$. If c is finite and of length n, there exists no state s such that $(s_n, s) \in T^s$.

[5]In this section, we do not distinguish a major state. Later in Definition 3.6 and in Theorem 3.4 we will use the convention that the first component of a local state is called the major state, and that there may be only a finite number of different atomic states for it.

The last restriction implies that we extend a computation as far as possible. (Other formalisms have been presented which count all the incomplete prefixes $\langle s_0, \ldots, s_i \rangle$ of a computation as a “computation”, too.) Here, a sequence of global states is only a computation if it is either infinite or there is no more simple transition by which it could be extended. Each global automaton defines a set of possible computations:

Definition 2.8 (set of possible computations) *The set of possible computations of an automaton A* is the set $C = \{c \mid c \text{ is a computation of } A\}$.

2.4 Adding to an Existing System

Now we come to the part of the formalism that will define what a feature is. For this, we describe how we add something to a global automaton. In our formalism, it is a prerequisite for the detection of interactions between features that the different features are specified separately. We assume (in the spirit of the IN conceptual model [ITU92, DuVi92]) that such a partitioning is possible, and we are not concerned in this report about optimization steps that could lead to a (re-)integration of features.[6] Since we model the entire system by computations, it must be possible in our formalism to associate each transition with a single feature[7]. For this, every transition must contain only functionality of a single feature. This implies that we have to use a *specific specification style* when we specify the system and its extensions. Two basic rules we adopt here are that we *modify only on a coarse-grained level*, and that we *only add* to the specification. This means for the behaviour part that we never modify an existing transition to achieve the behaviour of a new feature, but we only add an entirely new transition. Also, we do not delete any obsolete transition (see Section 4 for how they can be treated). The only way to extend a global automaton in our formalism is to add new possible computations.

We will design our formalism in such a way that it supports these stylistic rules for the state space part, too. In a first step, we only add to the state space, without modifying the way in which the computations evolve. A state extension function defines where something is added to a state.

Definition 2.9 (local state extension function)

A local state extension function ϕ is a function $\phi : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$ such that $m, n \in \mathbb{N}^+$ and ϕ is injective.

[6]Currently, the telecommunications community is much more concerned about the feature interactions problem than about execution speed. Due to the staggering development of computing hardware, they only came into the position to introduce so many new features in so short time.

[7]For orthogonality, we consider the basic system as just another feature, too.

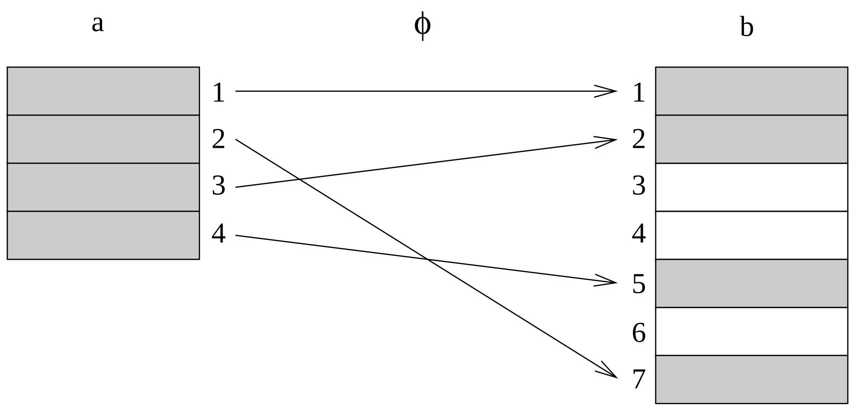

Figure 2: The local state extension function may rearrange the local state tuple.

Note that the injectiveness of the state extension function ϕ implies that the resulting state cannot shrink ($m \leq n$). Now we can augment a local state by something new:

Definition 2.10 (local state extension) *The local state $b = (b_i)_{i=1}^n$ is a local state extension of the local state $a = (a_j)_{j=1}^m$ with respect to the local state extension function $\phi : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$, written $b \succeq^\phi a$,* iff $\forall k : 1 \leq k \leq m : b_{\phi(k)} = a_k$.

Note that a local state is a local state extension of itself with respect to the identity function. Note further that we do not require that the state extension function maps the old parts of the local state onto the first m elements of the extended local state (i.e., $\forall k : 1 \leq k \leq m : \phi(k) = k$), and we do not even require that the state extension function ϕ be monotonic. This means that the state extension function may insert the extended parts in arbitrary places, and that the order of the old parts may be rearranged (compare Figure 2). Nevertheless, soon we will demand that this rearrangement is done the same way all over the lifetime of the system. Next, we carry the idea of the state extension from the local states on to the global states.

Definition 2.11 (global state extension function) *A global state extension function* is a tuple $\phi^G = (\phi_i)_{i=0}^m$ such that $\phi_1, \ldots, \phi_m$ are local state extension functions and ϕ_0 is a function $\phi_0 : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$ such that $m, n \in \mathbb{N}^+$ and ϕ_0 is injective.

Element ϕ_0 describes where new system components are introduced, and the other ϕ_i are the local state extension functions for all the hitherto existing components.

Definition 2.12 (state extension) *The global state $b = (b_i)_{i=1}^n$ is a state extension of the global state $a = (a_j)_{j=1}^m$ with respect to the global state ex-*

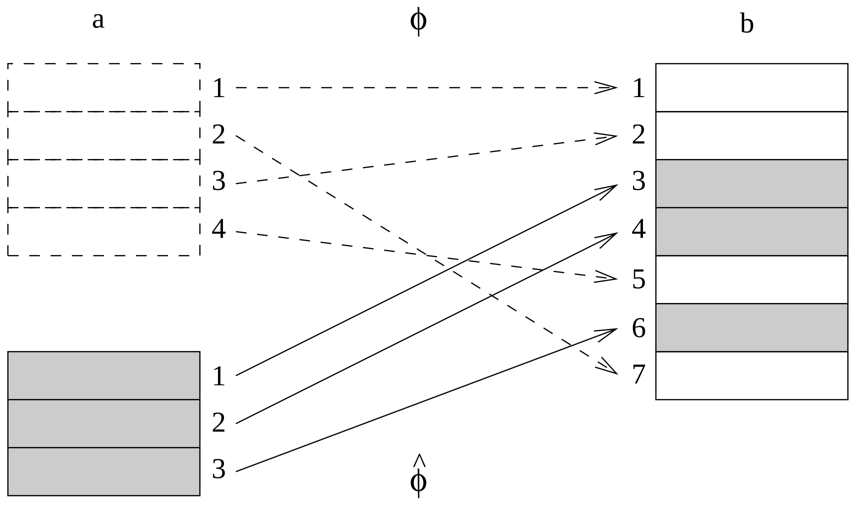

Figure 3: The complementary local state extension function $\hat{\phi}$ picks all the indices where an extension part was inserted.

tension function ϕ^G*, written* $b \succeq^{\phi^G} a$*,* iff $\phi_0 : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$ and $\forall k : 1 \leq k \leq m : b_{\phi_0(k)} \succeq^{\phi_k} a_k$.

Below we will need a function which picks all the indices of an extended local state where an extension part was inserted (compare Figure 3):

Definition 2.13 (complementary local state extension function)
The complementary local state extension function $\hat{\phi}$ *to a local state extension function* $\phi : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$*,* $m \leq n$*,* is the function $\hat{\phi} : \{1, \ldots, n - m\} \mapsto \{1, \ldots, n\}$ with

- $\forall j : 1 \leq j \leq n - m : \forall i : 1 \leq i \leq m : \hat{\phi}(j) \neq \phi(i)$, and
- $\hat{\phi}$ is strictly monotonic.

When we have extended a local state, we not only want to know *where* it was extended (compare the local state extension function above); we also want to know what *values* the new parts have:

Definition 2.14 (state extension part of a local state)
The state extension part $\Delta(s^l, \phi)$ *of a (previously extended) local state* $s^l = \left(s^l_i\right)_{i=1}^n$ *with respect to a local state extension function* $\phi : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$*,* $m \leq n$*,* is the tuple $\Delta(s^l, \phi) = \left(\Delta(s^l, \phi)_i\right)_{i=1}^{n-m}$ with $\Delta(s^l, \phi)_i = s^l_{\hat{\phi}(i)}$.

This can be generalized to global states:

Definition 2.15 (state extension part of a global state) *The state extension part $\Delta(s^G, \phi^G)$ of a (previously extended) global state $s^G = \left(s_i^G\right)_{i=1}^n$ with respect to a global state extension function $\phi^G = (\phi_i)_{i=0}^m$ with $\phi_0 : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$, $m \le n$,* is the tuple $\Delta(s^G, \phi^G) = \left(\Delta(s^G, \phi^G)_i\right)_{i=1}^n$ with

$$\Delta(s^G, \phi^G)_i = \begin{cases} \Delta(s_i^G, \phi_j) & \text{if } \exists j : 1 \le j \le m : \phi_0(j) = i \\ s_i^G & \text{otherwise} \end{cases}$$

Now we can define what a state-extended computation is.

Definition 2.16 (state-extended computation) *A computation d is a state-extended computation of a computation c with respect to the state extension function ϕ^G, written $d \succeq^{\phi^G} c$,* iff d and c have the same length $n \in \mathbb{N} \cup \{\infty\}$ and $\forall i : 0 \le i \le n : d_i \succeq^{\phi^G} c_i \wedge \Delta(d_i, \phi^G) = \Delta(d_0, \phi^G)$.

Note that the last conjunction requires that the state extension part remains the same all over the run of the computation. We want to achieve that the extension of the state space is redundant as long as it is not explicitely modified by a newly added transition (see below). We want to introduce new behaviour only by explicitely adding more transitions, and these new transitions should have variables of their own.

In the next section, we will need the notion of state extension for sets of possible computations, too:

Definition 2.17 (state-extended set of possible computations) *A set of possible computations C_b is a state-extended set of possible computations of a set of possible computations C_a with respect to the state extension function ϕ^G, written $C_b \succeq^{\phi^G} C_a$,* iff there exists a bijective function $\Upsilon^C : C_a \to C_b$ with $\Upsilon^C(c) \succeq^{\phi^G} c$ for all computations $c \in C_a$.

In Definition 2.16, as well as in all the other definitions, we took care that there is only a single global state extension function ϕ^G for the entire set of computations. As soon as we introduce extended automata, this will render a static and uniform kind of state extension.

Based on the notion of the state-extended computation, we can define the extension of a global automaton:

Definition 2.18 (extension of a global automaton) *A global automaton B is an extension of a global automaton A with respect to the global state extension function ϕ^G, written $B \succeq^{\phi^G} A$,* iff for every computation c of A there is a computation d of B with $d \succeq^{\phi^G} c$.

Note that the extended global automaton B may have more computations than A, and $C_B \succeq^{\phi^G} C_A$ does not necessarily hold. This extension does not only comprise the state space, but also the behaviour.

For the definition of a feature, we have to define first the state extension of a transition:

Definition 2.19 (state extension of a transition) *The transition $t_b = (S_b^e, \sigma_b, \lambda)$, $\sigma_b \subseteq S_b^G \times S_b^G$, is a state extension of the transition $t_a = (S_a^e, \sigma_a, \lambda)$, $\sigma_a \subseteq S_a^G \times S_a^G$, with respect to the state extension function ϕ^G, written $t_b \succeq^{\phi^G} t_a$,* iff

- there exists a bijective function $\Upsilon^S : S_a^G \to S_b^G$ with $\Upsilon^S(s) \succeq^{\phi^G} s$ and $s \in S_a^e \Leftrightarrow \Upsilon^S(s) \in S_b^e$ for all global states $s \in S_a^G$, and
- there exists a bijective function $\Upsilon^\sigma : \sigma_a \to \sigma_b$ with $\Upsilon^\sigma(\,(s_1, s_2)\,) = (\Upsilon^S(s_1), \Upsilon^S(s_2))$ for all simple transitions $(s_1, s_2) \in \sigma_a$.

Definition 2.20 (state extension of a set of transitions) *The set of transitions T_b is a state extension of the set of transitions T_a with respect to the state extension function ϕ^G, written $T_b \succeq^{\phi^G} T_a$,* iff there exists a bijective function $\Upsilon^T : T_a \to T_b$ with $\Upsilon^T(t) \succeq^{\phi^G} t$ for all transitions $t \in T_a$.

Since we want to associate every transition to a specific feature, we do this by structuring the transition name appropriately.

Definition 2.21 (transition name) *A transition name* λ is a pair (φ, γ) of a feature name φ and a local name γ.

Now, we have completed the basic definitions so we can define a feature itself, finally.

Definition 2.22 (feature) *A feature for a global automaton $A = (S, T, s_0)$* is a tuple $f = (\phi^G, T^e, d^0)$ where T^e is a set of transitions which have all the same feature name φ, and ϕ^G is a global state extension function for S such that there exists a global automaton $A' = (S', T', s_0')$ with

- $A' \succeq^{\phi^G} A$, and
- $d^0 = \Delta(s_0', \phi^G)$, and
- there exists a T'' with $T'' \succeq^{\phi^G} T$ such that $T' = T'' \cup T^e$, and
- no transition in T has the feature name φ.

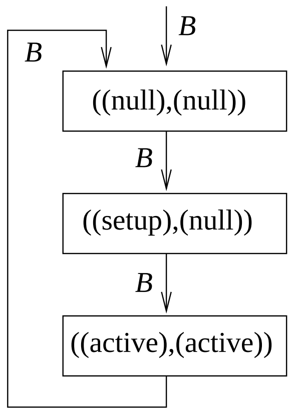

Figure 4: A global automaton with two local components.

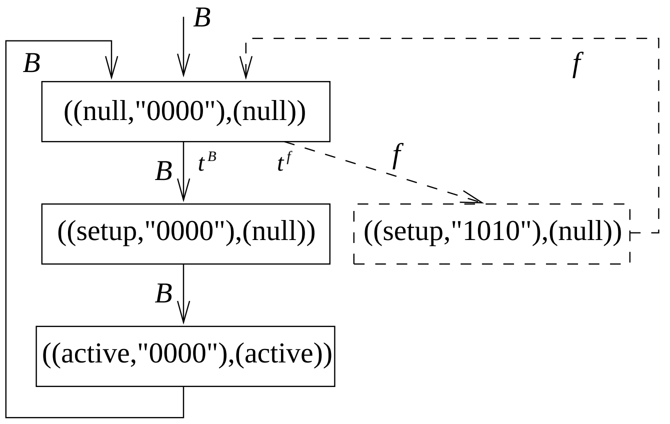

Figure 5: Adding the feature f of dialling numbers to the global automaton.

Note that we are allowed to add a feature f to a global automaton A only if A does not contain any transition with the feature name φ of the new feature f.

Example 2.22.1 Figure 4 shows an original automaton which only contains transitions of a simple basic system B; in Figure 5, a (simple) feature f has been added. The state space has been extended by a new local component for the dialled number (via ϕ^G), the initial state has got a value for this number (via d^0), and there are new transitions (T^e).

Definition 2.23 (global automaton with feature f) *A global automaton with feature f* is A' from Definition 2.22, written $A \oplus f$.

In the way it is defined above, a feature is an *increment* in the possible behaviour. It only allows for more computations, but it does not allow to remove any computation which was possible for the global automaton without the feature. This is a deliberate restriction which we designed into our formalism. This way, we still have all the computations of the other features and can find deviations (see Section 2.5). If the purpose of the new feature is to remove the other computation, then this is certainly an interaction, and it should be handled explicitely. In Section 4, we will discuss how this resolution may take place.

Definition 2.24 (simple transition belongs to feature)
A simple transition (s_1, s_2) *of a global automaton* A *belongs to the feature* $f = (\phi^G, T^e, d^0)$ *for* A iff there exists a transition $t = (S^e, \sigma, \lambda) \in T^e$ such that $s_1 \in S^e$ and $s_1 \, \sigma \, s_2$.

Note that a simple transition may belong to more than one feature. We will need this notion in several definitions below.

Definition 2.25 (feature mapping) *A feature mapping for a global automaton* $A = (S, T, s_0)$ is a function $\mathcal{F} : T \mapsto N$ where N is a set of feature names such that each transition t is mapped onto the feature name φ^t in its transition name λ^t.

We consider only global automata with at least one transition (see Definition 2.6) so that there is at least one feature name. By convention, the "basic system" of the global automaton which is always present carries the feature name B.

Definition 2.26 (feature in a global automaton)
A feature in a global automaton A is a feature $f = (\phi^G, T^e, d^0)$ such that there is a global automaton A' with $A = A' \oplus f$, and there is no transition in A' which has the same feature name as the transitions in f.

Definition 2.27 (global automaton without feature f)
The global automaton without feature f of a global automaton A is the global automaton $A \backslash f$ for which $A = (A \backslash f) \oplus f$, and there is no transition in $A \backslash f$ which has the same feature name as the transitions in f.

Note that there is always a feature mapping for a global automaton which maps all its transitions to a finite set of feature names. An automaton is determined entirely by its features, except for the order of its state tuples. (The proof of this is left to the reader.) Some other authors allow for different instances of the same feature, resulting in the possibility that a feature may interact with itself. We renounce this because it makes necessary a distinction between a feature and its instances. If this is really needed, it can be expressed by explicit repetition.

Definition 2.28 (set of possible computations of a feature)
The set C^f of possible computations of a feature f for a global automaton A is the set of all possible computations c of A for which exists a simple transition $(c_i, c_{i+1}) \in c$ which belongs to feature f.

Note that a computation may belong to more than one feature's set of possible computations.

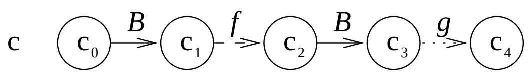

Figure 6: A computation c which is only possible if both features f and g are present.

2.5 Formal Definition of "Feature Interaction"

In Section 2.1, we stated that it is a necessary precondition for a feature interaction that the behaviour of a feature is changed. Using the formalism introduced up to now, we are now able to formalize this notion. A feature f interacts with a feature g if the set of possible computations of g depends on wether f is in the automaton or not (modulo the state space extension performed by f). Or more formally:

When we have a feature f and a feature g in the global automaton A, and $f \neq g$, and it holds for the set C^g_A of possible computations of feature g in A, and for the set $C^g_{A\backslash f}$ of possible computations of feature g in A without feature f that

$$\neg\exists\phi^G : C^g_A \succeq^{\phi^G} C^g_{A\backslash f}$$

then f interacts with g.

We will illustrate this by an example: think of a computation c which is a possible computation of feature f as well as of feature g,[8] i.e., $c \in C^f$ and $c \in C^g$, and where a simple transition belongs to feature f (Def. 2.24) exclusively (and therefore belongs to neither g nor B), and where a second simple transition belongs to feature g exclusively. Compare Figure 6. Then this computation c is only possible if both features f and g are present in the global automaton. In the example in Figure 6, a system made up only out of features B and f would generate only the computation $\langle c_0, c_1, c_2, c_3\rangle$. The addition of feature g to this system renders the computation $\langle c_0, c_1, c_2, c_3, c_4\rangle$, shown in Figure 6, thus modifying a computation which is associated with feature f, and therefore changing the behaviour of f. On the other hand, if the system is made up only out of features B and g, there would be only the computation $\langle c_0, c_1\rangle$, and the addition of feature f would modify it with the same result as before. Therefore, both features mutually interact.

But there is also another way in which the behaviour of feature g can be changed by feature f, even without changing the set of possible computations of g. There may be a computation c^g in the global automaton $A\backslash f$ which contains a simple transition (c^g_j, c^g_{j+1}) that belongs to feature g (and not to B). From a certain point in this computation on, let us call it c^g_i, it becomes inevitable that the automaton will eventually execute the simple transition (c^g_j, c^g_{j+1}), because there is no more

[8]with $f \neq g$ and $f \neq B$ and $g \neq B$

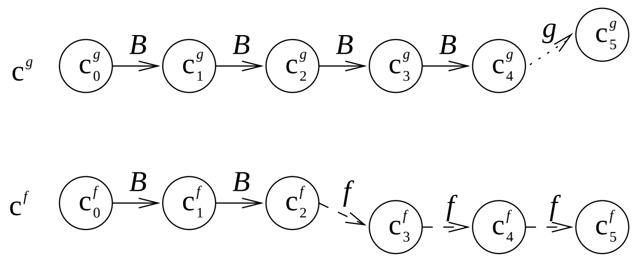

Figure 7: Feature f may divert the execution away from computation c^g of feature g.

non-determinism between c_i^g and c_{j+1}^g in the automaton $A\backslash f$. Then, we add the feature f to $A\backslash f$. This may introduce a computation c^f which does not contain any simple transition belonging to g, but which has a common prefix with c^g up to the global state $c_i^g = c_i^f$, and which continues differently at c_{i+1}^f. (Compare Figure 7.) Now, there is suddenly a non-deterministic choice in the automaton at the global state c_i^g, and the simple transition (c_j^g, c_{j+1}^g) is not necessarily executed. Imagine, e.g., a payphone where a customer has inserted some coins, but where he suddenly doesn't get connected through because of some interaction with a new feature (and there is no refund). The customer righteously expects that there is no such alternative in this case. In Figure 7, there are two such computations c^g and c^f, and feature f interacts with feature g in the way described. (But not vice versa, therefore this relation is not symmetric.)

All of the above cases are subsumed formally by Definition 2.29.

Definition 2.29 (feature f interacts with feature g)
A feature f in the global automaton A interacts with a feature g in A iff $f \neq g$ and there exists a computation $c^f \in C^f$ and a computation $c^g \in C^g$ and an index $i \in \mathbb{N}$ such that the prefixes of c^f and c^g up to the i-th state are equal, i.e, $\langle c_0^f, \ldots, c_i^f \rangle = \langle c_0^g, \ldots, c_i^g \rangle$, and the simple transition (c_i^f, c_{i+1}^f) belongs to feature f but neither to feature g nor to B, and there exists a simple transition (c_j^g, c_{j+1}^g) which belongs to feature g but neither to feature f nor to B.

Note that c^g may be equal to c^f, covering the first case described in the informal part above.

Every useful feature which is added on top of the basic system will of course modify its behaviour. This is the intention behind adding the feature. Therefore, for practical purposes we do not count such a feature interaction in the following definition.

Definition 2.30 (feature interaction among features)
A feature interaction among the features of a global automaton A occurs iff

there exist two features f, g in A such that f interacts with g and where both are not the basic system B.

3 Automatic Detection of Potential Feature Interactions

In Definition 2.30, we have defined what a feature interaction among the features of a global automaton is. Since computations are of infinite length in general, the criterion in Definition 2.29 cannot be used constructively (and efficiently). Therefore we have to find criteria for feature interactions which

1. can be computed efficiently,
2. are still necessarily fulfilled for feature interactions (wide enough), and
3. do include as few as possible potential interactions (specific as possible).

From Theorem 3.2 on (see below), we will present a sequence of criteria which all fulfill the first two items and which become increasingly better at the third item.

3.1 Necessary Conditions for Feature Interactions

Under the assumption of infinite computations[9], there may be some feature interaction only if there is a reachable state s in which there is a non-deterministic choice between at least two transitions belonging to different features.

Theorem 3.1 Be A a global automaton such that for all features h, $h \neq B$, holds that the set of all possible computations $C_{A\setminus h}$ contains only infinite computations.

A feature interaction among the features of A may occur only if there exist a feature $f = (\phi_f^G, T_f^e, d_f^0)$ in A, $f \neq B$, a transition $t_f = (S_f^e, \sigma_f, \lambda_f) \in T_f^e$, a feature $g = (\phi_g^G, T_g^e, d_g^0)$ in A with $g \neq f$, a transition $t_g = (S_g^e, \sigma_g, \lambda_g) \in T_g^e$ and a global state s such that $s \in c$ for some computation $c \in C$ and $s \in S_f^e$ and $s \in S_g^e$.

Note that g may equal B in Theorem 3.1.

[9]The assumption that the basic system for itself is free of deadlocks can be justified by the applications in mind, which are reactive systems. A first feature interaction may invalidate the above assumption for a second, but we can always detect the first one and after a resolution also the second one.

Proof: a feature interaction among the features of A implies that a feature f in A interacts with a feature g in A (Definition 2.30). So it remains to prove that this plus the premise that every computation is infinite implies the existence of the items enumerated in the end of Theorem 3.1. The global state s of Theorem 3.1 is identical with $c_i^f = c_i^g$ of Definition 2.29; the transition t_f of Theorem 3.1 is provided through the existence of the simple transition (c_i^f, c_{i+1}^f) of Definition 2.29. Now we have to distinguish two cases, depending on if (c_i^f, c_{i+1}^f) belongs to some other feature h, $h \neq f$, too, or not.

no: If we look at $A \backslash f$, we won't find (c_i^f, c_{i+1}^f) (modulo state space extension) there, since (c_i^f, c_{i+1}^f) belongs exclusively to f. We assume without restriction of generality that there is no simple transition in $\langle c_0^f, \ldots, c_i^f \rangle$ which belongs to f. Since there are no finite computations in $C_{A\backslash f}$, and since it must contain at least one computation because of feature B, there must be another continuation simple transition (c_i^f, c_{i+1}^g) with $c_{i+1}^g \neq c_{i+1}^f$, and therefore a corresponding transition $t_g = (S_g^e, \sigma_g, \lambda_g)$ exists with $s = c_i^g \in S_g^e$, and for feature g holds that $g \neq f$. (g may be equal to B.)

yes: For (c_i^f, c_{i+1}^f), there must be a corresponding transition $t_h = (S_h^e, \sigma_h, \lambda_h)$ with $s = c_i^f \in S_h^e$, and for feature h holds that $g \neq f$. $\Box$

The "yes" case was so easy to prove because there may be non-determinism between two transitions of different features even without a feature interaction, when those two transitions do exactly the same. But this is a pathological case, since the new feature f does nothing that was not done before by other features. Theorem 3.1 detects these cases too, and that seems to be rather an advantage since they are probably also some case of design error.

The condition of non-determinism is necessary for feature interactions, but (in general) less than sufficient for another reason, too: we did not count the interactions with the basic feature B. If a feature f interacts with B, then there may or may not be an interaction with another feature g, as shown in Figure 7. Therefore, if some feature f interacts with B, we are warned and have to check manually if there are transitions of g later in the affected computations. (We will give more specific criteria on this in the following subsections.)

Furthermore, the criterion in Theorem 3.1 is still not efficiently applicable because of the condition that s must be reachable. This would imply a complete state space search. Therefore we weaken the condition further and drop that condition:

Theorem 3.2 Be A a global automaton such that for all features h, $h \neq B$, holds that the set of all possible computations $C_{A\backslash h}$ contains only infinite computations.

A feature interaction among the features of A may occur only if there exist a feature $f = (\phi_f^G, T_f^e, d_f^0)$ in A, $f \neq B$, a transition $t_f = (S_f^e, \sigma_f, \lambda_f) \in T_f^e$, a

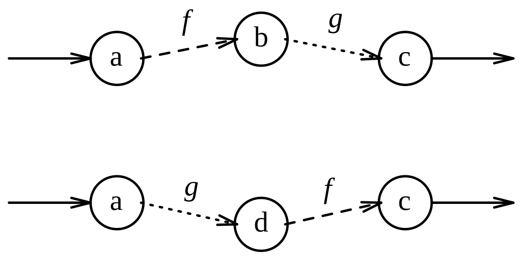

Figure 8: The firing order of these transitions is irrelevant.

feature $g = (\phi_g^G, T_g^e, d_g^0)$ in A with $g \neq f$, a transition $t_g = (S_g^e, \sigma_g, \lambda_g) \in T_g^e$ and a global state s such that $s \in S_f^e$ and $s \in S_g^e$.

Informally, Theorem 3.2 says that we can find any feature interaction if we compare all pairs of transitions from different features and check if their enabling sets overlap. If no such pair exists, then no feature interaction can occur. The proof is a straightforward extension of the above one.

A simple example for a situation as described in Theorem 3.2 may be found in Figure 5. The global state s is $\big((\text{null}, \text{“0000”}), (\text{null})\big)$. Due to our very abstract modelling, you find in this state a non-deterministic choice between the normal setup transition t^B and the newly added transition t^f which aborts the setup in case of a wrong dialled number. This means that the new feature f of dialling numbers may possibly interfere with the basic system B. It was impossible that a call could fail in our very simple basic system.

3.2 Excluding Concurrent Transitions

The above criteria warn us of potential feature interactions. But even if the computations of a feature are affected by another feature, this may be harmless in some cases. E.g., think of two transitions belonging to different features which are enabled in the same states and where the resulting state when both have fired does not depend on the order of firing. (Compare Figure 8.)

Definition 3.1 (independent transitions) *Two transitions $t_f = (S_f^e, \sigma_f, \lambda_f)$ and $t_g = (S_g^e, \sigma_g, \lambda_g)$ of a global automaton A are independent* iff

- for all enabled states $s_1 \in S_f^e$ and for all simple transitions $(s_1, s_2) \in \sigma_f$ holds that $s_1 \in S_g^e \Leftrightarrow s_2 \in S_g^e$,
- for all enabled states $s_1 \in S_g^e$ and for all simple transitions $(s_1, s_2) \in \sigma_g$ holds that $s_1 \in S_f^e \Leftrightarrow s_2 \in S_f^e$, and
- for $\mathcal{T} = (S_f^e \cap S_g^e) \times S^G$ and $\overline{\sigma_f} = \sigma_f \cap \mathcal{T}$ and $\overline{\sigma_g} = \sigma_g \cap \mathcal{T}$ holds that

$$\overline{\sigma_f} \circ \overline{\sigma_g} = \overline{\sigma_g} \circ \overline{\sigma_f}$$

Informally, this means that two transitions are independent if and only if firing one of them does not disable or enable the other one, and if both can fire then the firing order is irrelevant.

Definition 3.2 (harmless potential feature interaction)
A potential feature interaction as determined by the criterion of Theorem 3.2 is harmless iff the interacting transitions t_f and t_g are independent.

This declares non-determinism as harmless if it, e.g., results from our modelling of concurrency of the local automata by interleaving. The condition of Definition 3.2 is usually hard to decide for two given transitions. Thus, we will reformulate it based on the local state components modified by one transition and the local state components used by the other transition. As soon as we start to use a specific (constructive) formal description technique, these two notions are quite straightforwardly applied to specific language constructs. For example, an assignment to a local variable will lead to a modification of this local state component, and the use of the value of a local variable will be a use of this local state component.

The *output modification pattern* O^t (Definition A.1 on page 38) describes which local state components can possibly be modified by transition t. We print the actual formal definition of this notion and of the following two in Appendix A.1 since they are long but not essential to get the idea of what a state space dependence is.

The *output dependence pattern* D^t (Definition A.2 on page 39) describes for each output local state component (k, l) on which input local state components (i, j) it depends.

The *input pattern* I^t (Definition A.3 on page 40) describes which local state components (i, j) are used at all by transition t. This comprises both the enabling conditions and the computation of the output. Output dependences on the input in the fashion of the identity function are not counted.

The output modification pattern and the input pattern allow us to define the notion of a *state space dependence.*

Definition 3.3 (state space dependence) *A transition t_2 of a global automaton A has a state space dependence on a transition t_1 of A,* iff n is the tuple length of the global states and

$$\exists\, i \in \{1, \ldots, n\} : O_i^{t_1} \cap I_i^{t_2} \neq \emptyset$$

Example 3.3.1 We use the same setting as in Examples A.1.1 to A.3.1 of Appendix A.1.

$$\begin{aligned} O_1^t \cap I_1^t &= \{\} \\ O_2^t \cap I_2^t &= \{1\} \end{aligned}$$

Because of the second row, t has a state space dependence on itself.

Now we can write a criterion which implies a harmless potential feature interaction and which should be easier to check for any specific formal description technique.[10]

Theorem 3.3 A feature interaction as determined by the criterion of Theorem 3.2 is harmless if the interacting transitions t_f and t_g mutually have no state space dependence.

The proof is left to the reader.

3.3 Excluding Interactions with the Basic System

Theorem 3.2 from Section 3.1 is a criterion that warns us of all possible feature interactions. As already mentioned there, it also warns us if a feature f interacts with the basic system B, because there may be situations like the one in Figure 7 where this influences another feature g indirectly. But since a feature which does not interact with the basic system at all is rather useless, we have to differentiate further cases when f interacts with B.

To this end, we use our knowledge of the typical structure of a telecommunications system (compare, e.g., [ITU93b, ITU93a]). Call processing is described there by one extended finite state automaton (EFSA) (if we take a global view) or by two EFSAs (in a distributed view). These typically have a *null state* where no call is established and which is reached again after the processing of one call has been completed. Usually, this is also the initial state. This allows us to reduce the analysis of the computations c^f and c^g in Definition 2.29 to the end of the prefix where the null state has been reached the next time.

Definition 3.4 (returned-to state) *A global state s^G of a global automaton A is a returned-to state,* iff there exists a computation $c \in C$ and there exist two indices $i, j \in \mathbb{N}$ such that $s^G = c_i = c_j \wedge i < j$. The above indices i are called return indices of the returned-to state.

[10]This criterion is slightly weaker, the condition given in Theorem 3.3 is sufficient but not necessary for being harmless.

Definition 3.5 (global null state) *A global state s^G of a global automaton A is a global null state,* iff it is a returned-to state and there exists a return index i for it such that there is no other returned-to state which has a smaller return index.

Informally, this definition means that a global null state is the first state after a possible system initialization phase. This is marked by the fact that the system can return to this state after it has done some work.[11] Note that there could be more than one global null state.

In order to reflect the use of EFSA in our formal modelling, we adopt the convention that the first component of the local state tuple describes the *major state.* Any sensible specification of a system will not only have a finite, but also a small domain of values for each major state.

Definition 3.6 (major state) *The major state of a local state $s^l = (s_1, \dots, s_m)$* is the first component s_1.

This allows us to do an abstraction step and reduce a global automaton to a simpler one with the same number of local components, but where these local components only have a major state and no extended state.

Definition 3.7 (abstraction of a global state) *The abstraction function "$\downarrow$" renders for any global state $s^G = \big((s_{1,1}, \dots, s_{1,m_1}), \dots, (s_{n,1}, \dots, s_{n,m_n})\big)$* the abstracted global state $s^{G\downarrow} = \big((s_{1,1}), \dots, (s_{n,1})\big)$.

Example 3.7.1

A global state:

$$s^G = \big((\text{waitForAck}, 42, \texttt{'abcdef'}), (\text{connected}, 41, \texttt{'hijklm'}), (\text{idle}, 0)\big)$$

Its abstraction:

$$s^{G\downarrow} = \big((\text{waitForAck}), (\text{connected}), (\text{idle})\big)$$

The transitions of the abstracted global automaton are derived from the full automaton: if and only if there is a transition from a major state to another major

[11] It is debatable if this is the only sensible definition of a global null state. The important part is the fact that this state can be reached again. The rest of the definition has been chosen to be as simple as possible, it could be extended if this would be needed.

It may be difficult to achieve such a reset for some systems: if a call forwarding number has been set, it has to be reset first, and if accounting is modelled, the bill has to be mailed so that the account can be reset. Generally, all data base operations that don't have a clear local null state should be excluded from the modelling in order to make this theorem applicable. See also Definition 3.11 on page 24 ("null-resetting"). The modelling and analysis of long-lasting data-dependences is a topic for future research.

state for any value combination of the extended state part of the other local components, there is a transition in the abstracted global automaton. Accordingly, for any computation of the full automaton there will be a corresponding computation of the abstracted one, and maybe some more because we disregarded the dependence of transitions on the extended state part.

Definition 3.8 (abstraction of a transition) *The abstraction function "$\downarrow$" renders for any transition $t = (S^e, \sigma, \lambda)$* the abstracted transition $t^\downarrow = (S^{e\downarrow}, \sigma^\downarrow, \lambda)$, where

- $S^{e\downarrow} = \{s^{e\downarrow} \mid s^e \in S^e\}$, and
- $\sigma^\downarrow = \{(s_1^\downarrow, s_2^\downarrow) \mid s_1 \sigma s_2 \}$

Definition 3.9 (abstraction of a global automaton) *The abstraction function "$\downarrow$" renders for any global automaton $A = (S^G, T, s_0)$* the abstracted global automaton $A^\downarrow = (S^{G\downarrow}, T^\downarrow, s_0^\downarrow)$, where

- $S^{G\downarrow} = \{s^{G\downarrow} \mid s^G \in S^G\}$, and
- $T^\downarrow = \{t^\downarrow \mid t \in T\}$

This abstraction step to major states and a limitation of the analysis up to the next global null state allow us to do an exhaustive reachability analysis without the tractability problems that a state space explosion imposes.

We start a reachability analysis at the state s where a transition of a feature f shows non-determinism with a transition of the basic system B, and we follow all transitions from this state. If we cannot find any transition belonging to some other feature than B or f before we reach the global null state again, it is impossible that the detected non-determinism signifies a feature interaction among the features of the global automaton (Definition 2.30). If we can find such a transition, there may be a harmful feature interaction and we have to analyse this situation manually. The manual analysis is supported with the information where the non-determinism showed up, which was the offending computation (prefix) and which were the possibly interacting transitions, these in turn provide information on the features which are concerned.

Definition 3.10 (feature f interacts with feature g cross-reset) *A feature f in the global automaton A interacts with a feature g in A cross-reset* iff feature f interacts with feature g as described in Definition 2.29 on page 15 and for the computations c^f, c^g and the index i from this definition exists an index $j > i$ such that c_j^g is a global null state and there is no simple transition between c_0^g and c_j^g that belongs to feature g but neither to f nor to B.

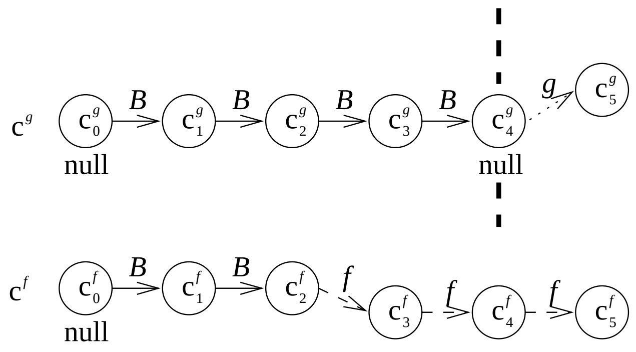

Figure 9: A cross-reset and therefore not harmful feature interaction due to a global null state inbetween.

Compare Figure 9. We will assume that such cross-reset feature interactions are never harmful. If a telephone system returns to a global null state, all state components are reset to their original values, and therefore "the system loses all knowledge on its history". If we additionally assume that the humans using the system at least lose all interest in this history as soon as they have reset the system, we can conclude that the cross-reset feature interactions can never be harmful.

Theorem 3.4 Be $A = (S^G, T, s_0)$ a global automaton such that for all features h, $h \neq B$, holds that the set of all possible computations $C_{A \setminus h}$ contains only infinite computations. Furthermore, every computation $c \in C_A$ shall contain a global null state and for every computation this global null state shall be reached again a second time.[12]

A feature interaction among the features of A which is not cross-reset may occur only if there exist a feature $f \neq B$ in A, a transition $t_f \in T^e_f$, a feature g in A with $g \neq f$, a transition $t_g \in T^e_g$ and a global state s such that $s \in S^e_f$ and $s \in S^e_g$, and additionally:

- $g \neq B$ or
- (if $g = B$:) for the global automaton $A' = (S^G, T, s)$[13], there exists a computation which contains a simple transition (s_m, s_{m+1}) that belongs to some other feature than B or f and no state in $\langle s, \ldots, s_m \rangle$ is a global null state of the original global automaton A.

The proof is based on Theorem 3.2, of which this theorem is an extension, and is left to the reader. Figure 10 shows a global automaton for which the condition

[12]This implies that the set of all possible computations C_A of A itself contains only cyclic and therefore infinite computations.

[13]i.e., A' is the same as A but starts at s instead of s_0

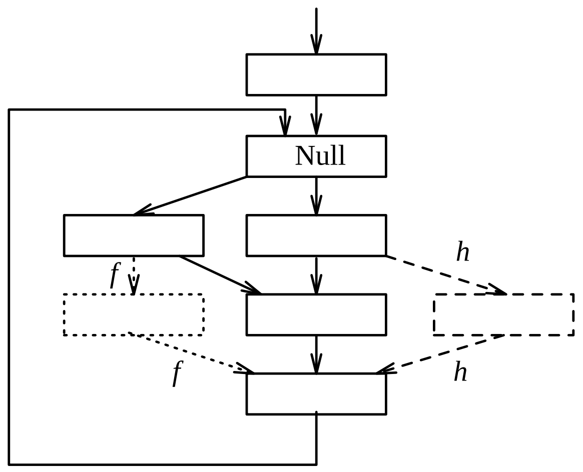

Figure 10: A global automaton with only a cross-reset feature interaction.

in the second paragraph of Theorem 3.4 is always false which means that there is only a cross-reset feature interaction.

Since an implementation of the implied algorithm still may have to cope with a large state space, for a certain class of systems we can abstract away all state components except for the major states. In order to do this, we have to add that the domain of all major states be finite, and we need that the investigated global automaton is "very consequent on forgetting earlier sessions", as in the following definition:

Definition 3.11 (null-resetting global automaton) *A global automaton A is null-resetting* iff for all global states s_1, s_2 holds that if their abstractions are equal, i.e. $s_1^{\downarrow} = s_2^{\downarrow}$, and s_1 is a global null state, then also s_2 is a global null state.

Informally, this means that all extended state space components are always reset as soon as all the local major states return to a null state. For such null-resetting automata, we can reduce the complexity of the state space search considerably through a modification of Theorem 3.4:

Theorem 3.5 Be $A = (S^G, T, s_0)$ a global automaton such that for all features h, $h \neq B$, holds that the set of all possible computations $C_{A \setminus h}$ contains only infinite computations. Furthermore, every computation $c \in C_A$ shall contain a global null state and for every computation this global null state shall be reached again a second time; and be A null-resetting.

A feature interaction among the features of A which is not cross-reset may occur only if there exist a feature $f \neq B$ in A, a transition $t_f \in T_f^e$, a feature g in A with $g \neq f$, a transition $t_g \in T_g^e$ and a global state s such that $s \in S_f^e$ and $s \in S_g^e$, and additionally:

- $g \neq B$ or
- (if $g = B$:) for the abstracted automaton $A'^{\downarrow}$ of the global automaton $A' = (S^G, T, s)$, there exists a computation which contains a simple transition $(s^{\downarrow}_m, s^{\downarrow}_{m+1})$ that belongs to some other feature than B or f and no state in $\langle s^{\downarrow}, \ldots, s^{\downarrow}_m \rangle$ is a an abstraction of a global null state of the original global automaton A.

The proof is trivial. Since all major states have finite domains, and the local states only comprise the major state, the search implied by the quantification operators is now always finite. If these domains are additionally small, as in usual specifications, the search is even not costly.

Note that the null-resetting property of A is needed: otherwise we might stop our search too early which looks for a transition that belongs to some other feature than B or f. We might stop at a global state of A' which is an abstraction of a global null state of A but where this global null state would not be reached by A because some part of the extended state space has not been reset yet. E.g., a call forwarding number might not have been reset.

This need for the null-resetting property has consequences for a suitable specification style that supports the detection of feature interactions. Any transition which leads into a null state should explicitely re-initialize the state space. Usually, one just doesn't care for state components which contain some old data which will never be used again. But here, we have to reset them to some constant values. In a practical FDT, this style rule means that a transition which leads into a null state must assign some constant values to all variables that are defined. Such a transition also terminates any indirect data dependences.

3.4 Excluding Independently Inserted Features

We may use another criterion that reduces the class of potential feature interactions further which have to be analysed manually. For this, we partially take into account the extended state space, again. A common situation in a local call processing FSA is shown in Figure 11. Two short automata extensions/features are inserted into the mostly sequential control flow of the basic local automaton. Examples for such a feature may be "abbreviated dialling" which translates a single digit into a full directory number, or "reverse charging" which changes some charging control variables. The first extension can never disable the second because it returns control back to nearly the same point where it took it away. This return situation may be detected easily during our exploration of the major state space, and be treated accordingly.

Definition 3.12 (major state preserving)
A transition $t = (S^e, \sigma, \lambda)$ of a global automaton $A = (S^G, T, s_0)$ is major

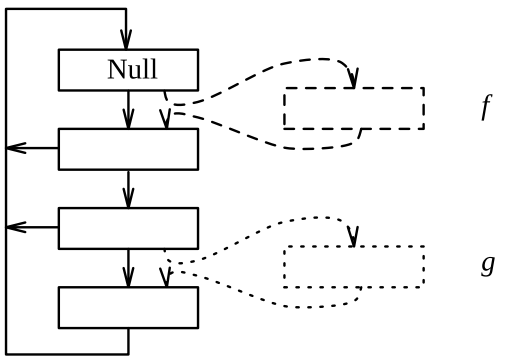

Figure 11: A typical situation with two new features.

> *state preserving* iff for all (abstracted) simple transitions $(s_1^{\downarrow}, s_2^{\downarrow}) \in \sigma^{\downarrow}$ with $s_1^{\downarrow} \in S^{e\downarrow}$ holds that $s_1^{\downarrow} = s_2^{\downarrow}$ and/or the simple transition $(s_1^{\downarrow}, s_2^{\downarrow})$ belongs also to the basic system B.

Informally, this means that transition t either does not modify the major states at all or that there exists a transition t_B of the basic system such that t_B modifies the major states in exactly the same way as t. Note that this definition does *not* comprise the case depicted in Figure 11. There, more than one transition is inserted before control is returned to the basic system. Such patterns are only sensibly defined for a local atomaton (see Section 3.5).

Of course, a situation where two short automata extensions/features are inserted into the control flow is still a feature interaction in the sense of Definition 2.29, since there are computations which are possible only if both features are present in the automaton. But often those features act on different parts of the state space so that each feature still "does the same" on the part of the state space it uses as without the other feature. This leads us to an extension of the notion of state space dependence. We define an indirect state space dependence. And if two features interact as in the scenarios above, but without such an indirect state space dependence, we call this a *minor feature interaction* (see Definition 3.14 below). We suppose that we should scrutinize manually minor feature interactions only after the more interesting cases have been handled, since the cases without an even indirect state space dependence probably present no harmful feature interaction.

An example of an indirect state space dependence is the case where the effects of a transition t_B associated with the basic system B depend on a state component which was modified by the first transition t_f, and where in turn a second transition t_h depends on these effects of t_B (compare Figure 12, left side). To resolve this situation, we need to do some simple data flow analysis.

But we have to watch out for another pattern of dependence, too (compare Figure 12, right side): in the beginning of a computation, transition t_f modifies some

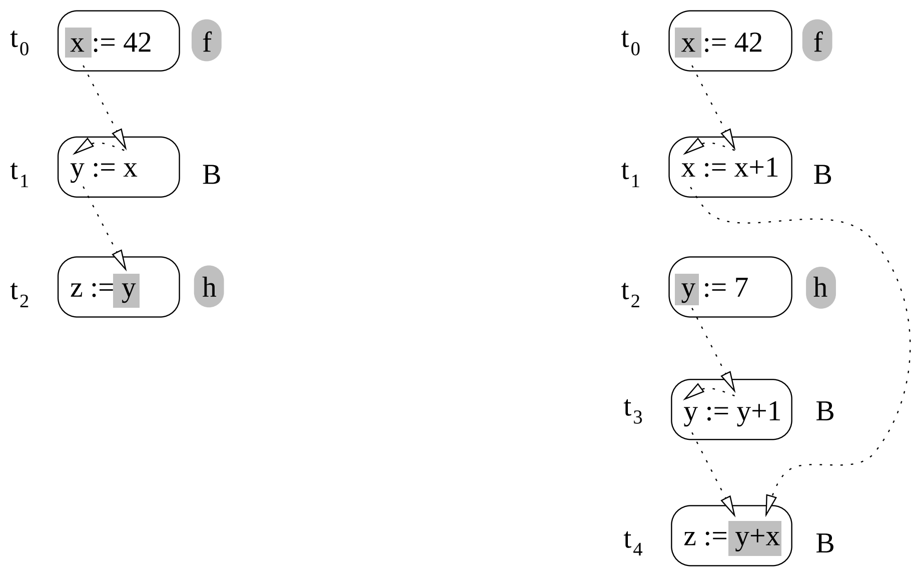

Figure 12: Two examples of indirect data dependences.

part of the state space (let us call it x), later, transition t_h modifies another, different part (y), and even later, a transition t_B of the basic system depends on both x and y. An example is the case where the first feature f changes the clock rate by which the charge is incremented, and where the second feature g changes the customer who is charged. The dependence shows up only when the analysis reaches the transition of the basic system which acutally charges a specific customer. The following definition comprises both cases.

Definition 3.13 (indirect state space dependence)

A sequence $\langle t_1, \ldots, t_m\rangle$ of transitions contains an indirect state space dependence on a transition t_0 iff t_0 belongs to a feature f, $f \neq B$, and there exists a transition t_j, $1 \leq j \leq m$, which does neither belong to feature f nor to the basic feature B such that

- for the a transition[14] $t = (S^e, \sigma, \lambda)$ with $S^e = S^e_{t_1}$ and $\sigma = \sigma_{t_1} \circ \ldots \circ \sigma_{t_j}$ and a new λ holds that t has a (direct) state space dependence of t_0 (Def. 3.3), or
- there exists another transition t_k, $j < k \leq m$ in the sequence such that for the the transitions $t = (S^e, \sigma, \lambda)$ and $t' = (S^{e\prime}, \sigma', \lambda')$ with $S^e = S^e_{t_1}$ and $S^{e\prime} = S^e_{t_{j+1}}$ and $\sigma = \sigma_{t_1} \circ \ldots \circ \sigma_{t_k}$ and $\sigma' = \sigma_{t_{j+1}} \circ \ldots \circ \sigma_{t_k}$ and a new λ and a new λ' holds that t has a (direct) state space dependence of t_0 and t' has a (direct) state space dependence of t_j.

[14]This transition t *does not* need to belong to the global automaton!

Figure 12 shows examples of indirect data dependences for both cases.

As mentioned before, such state space dependences are quite easy to determine for the language constructs of any specific (constructive) formal description technique. This extends naturally for indirect state space dependences, we only have to perform some kind of sequential concatenation of the transition relations. E.g. for the FDT Estelle, we mainly have to concatenate the Pascal-style transition body code.

Based on the definition of an indirect state space dependence, we are able to define the notion of a minor feature interaction formally, which was introduced already above.

Definition 3.14 (minor feature interaction) *A feature interaction as defined in Definition 2.29 on page 15 is minor* iff $c^f = c^g$ and $j > i$ and for the computation c_g holds additionally that there exists a sequence of transitions $\langle t_i, \ldots \rangle$ such that (c^g_k, c^g_{k+1}) is one of the simple transitions of t_k[15] for all $k \geq i$, and the sequence $\langle t_{i+1}, \ldots \rangle$ contains no indirect state space dependence on transition t_i.

Now, we can strengthen Theorem 3.5 for the major state preserving case.

Theorem 3.6 Be $A = (S^G, T, s_0)$ a global automaton such that for all features h, $h \neq B$, holds that the set of all possible computations $C_{A\backslash h}$ contains only infinite computations. Furthermore, every computation $c \in C_A$ shall contain a global null state and for every computation this global null state shall be reached again a second time; and be A null-resetting.

A feature interaction among the features of A which is not cross-reset and not minor may occur only if there exist a feature $f \neq B$ in A, a transition $t_f \in T^e_f$, a feature g in A with $g \neq f$, a transition $t_g \in T^e_g$ and a global state s such that $s \in S^e_f$ and $s \in S^e_g$, and additionally:

- $g \neq B$ or
- (if $g = B$:)
  - t_f is major state preserving, and
  - for the abstracted automaton $A'^{\downarrow}$ of the global automaton $A' = (S^G, T, s)$, there exists a computation $\langle s^{\downarrow}_1, \ldots, s^{\downarrow}_n, \ldots \rangle$ with $s^{\downarrow}_1 = s^{\downarrow}$ and s_n being a global null state of the original global automaton A and no state $s^{\downarrow}_i$ with $1 \leq i < n$ is such an abstraction of a global null state, and

[15]or, more formally: $t_k = (S^e_k, \sigma_k, \lambda_k)$ and $c^g_k \in S^e_k \wedge c^g_k \; \sigma_k \; c^g_{k+1}$

– there exists a sequence of transitions $\langle t_1, \ldots, t_{n-1} \rangle$ such that $(s_i^{\downarrow}, s_{i+1}^{\downarrow})$ is one of the simple transitions of $t_i^{\downarrow}$[16] for all $1 \leq i < n$, and this sequence of transitions contains an indirect state space dependence on feature f (Def. 3.13).

The proof follows directly from Theorem 3.5. Note that we did not require the sequence of transitions $\langle t_1, \ldots, t_{n-1} \rangle$ to be executable by A, but only the sequence $\langle t_1^{\downarrow}, \ldots, t_{n-1}^{\downarrow} \rangle$ be executable by its abstracted counterpart $A^{\downarrow}$. This makes Theorem 3.6 weaker than it could be, but it saves us from exploring the full state space.

3.5 Analysis on a Local Level

Since we describe distributed and concurrent systems, the specified systems consist of local components (modules/processes) which operate with a certain degree of independence. (Except for descriptions on the global functional plane.) Many transitions have effects only on one local component. This is supported by most of the specific formal description techniques. Their syntax of transitions allows only local effects except for some special communication commands. This can be exploited by performing individual checks for feature interactions on a local level first. Due to the more restricted state space and the usually smaller degree of concurrency, this allows a deeper and more thorough automatic analysis without a state space explosion.

All detection criteria from the preceding subsections may as well be applied only to one local component. This can already discover those feature interactions which happen only locally. For example, an interaction may be discovered (and resolved) which results from an unspecified order of originating call screening and abbreviated dialling, in an appropriate local setting. We can determine if a transition has only local effects by the output modification pattern introduced in Section 3.2. In Sections 3.3 and 3.4, we proposed some (major state) data flow analysis after the detection of non-determinism between two transitions of different features. This analysis may be restricted to a single local component as long as only transitions with local effects are involved. If we reach a transition with non-local effects we have three options: we can study the consequences manually; we can apply the following analysis globally, as described in the preceding subsections; or we can continue with another local analysis. These secondary level local analyses start with any transition that depends on the state component in question, as determined by the input pattern defined in Section 3.2.

A local analysis has a further advantage: in Section 3.4 we discussed major state preserving transitions. On a global level, we could include only those transitions into the definition which either didn't modify the major states at all or for which

[16]or, more formally: $t_i^{\downarrow} = (S_i^{e\downarrow}, \sigma_i^{\downarrow}, \lambda_i)$ and $s_i^{\downarrow} \in S_i^{e\downarrow} \wedge s_i^{\downarrow}\ \sigma_i^{\downarrow}\ s_{i+1}^{\downarrow}$

existed a “similar” transition of the basic system. This did not comprise the case depicted in Figure 11. On a local level, we can sensibly extend the notion of major state preserving to entire groups of transitions. We replace the single transition of above by a sub-automaton consisting of several transitions and states, and we only require that the entry and exit of this sub-automaton exposes the properties described above.[17]

4 Resolution of Feature Interactions

To resolve detected feature interactions, generally high-level design decisions on the policy of the service provider or on the needs of the users have to be made (compare, e.g., [Sri95]). And since the detected interactions are the typical cases nobody had thought of before, it is quite probable that a satisfactory answer cannot be deduced from any existing behavioural description (compare Section 1). Even if some solution can be deduced, we do not know if it also really satisfies the implicit and unexpressed requirements of the users and the provider.

But even if an automatic resolution is impossible in many cases, several manual resolution approaches are used [Cai92, CGL$^+$93]: one can define fixed priorities for the set of features, one can define precedences among features, and one can introduce a dedicated feature manager[18]. A completely different resolution approach is the negotiating agents approach based on artificial intelligence [GrVe93, GrVe94, Vel93].

In our approach, we employ two different methods to resolve the interactions: precedences and dedicated “resolution features”.

In the simple case, we have detected direct non-determinism between two transitions associated with different features. This is also the case when a new feature is “plugged” into the old system since, up to now, we do not have introduced a way to disable the old transition. We simply get this non-deterministic choice instead.

So, after the detection methods have been applied, and after we are sure that it is the right thing to override the old transition, we specify corresponding *precedences* between the transitions. Each transition of a feature is given the same precedence, therefore we only have to decide which feature takes precedence over which. This is done by setting up a precedence matrix for features. The matrix does not need to contain an entry for every pair of features, it is sufficient to enter the

[17]Some more details have to be considered, e.g., there must not be a deadlock, and the notions of input pattern and output modification pattern have to be extended suitably.

[18]The latter can be used when the basic call process and the features are strictly separated entities and can communicate only by exchanging messages. The feature manager monitors this communication channel and allows a more complex interaction handling in this architecture than the preceding two approaches.

Table 1: Example of a precedence matrix from [BrGo94a]

	BCS	SCRT	CF	SCRT&CF	SAFE	LUQE
BCS					$\succ$	
SCRT	$\succ$				$\succ$	
CF	$\succ$	$\succ$			$\succ$	
SCRT&CF	$\succ$	$\succ$	$\succ$		$\succ$	
SAFE						
LUQE					$\succ$	

explicit design decisions we actually made. An example for such a matrix can be found in Table 1, it is taken from a case study in an earlier technical report of us [BrGo94a].

Here is a sketch of how the precedences are incorporated into our formalism. We define a global automaton with priorities $A^p = (S^G, T^p, s_0, p)$ for the hitherto existing global automaton $A = (S^G, T, s_0)$, where p is a partial order on the set of feature names. Since each transition name λ is already associated unambigiously to a feature name φ by Definition 2.21, this renders a partial order on the set of transitions T. This allows us to construct the new global automaton A^p which is identical to A except that we remove any simple transition from T^{s^p} where all the corresponding transitions t were overridden by other transitions with a higher priority.

Of course, there may be other cases where precedences are not sufficient. Then, we need to specify additional behaviour which resolves the conflict. We specify this behaviour formally like any other new feature, and we add it to the system if and only if both of the interacting features are added. This approach for the integration of the resolving behaviour allows us to investigate even interactions which may happen between this newly introduced behaviour and a third feature.

More details on the resolution of feature interactions may be found in our technical report [BrGo94a]. There, we apply our ideas to the formal description technique Estelle.

It has to be stressed that the analysis of feature interactions as discussed in Sections 2 and 3 requires that there are *no precedences* in the global automaton. Nevertheless, we would like to apply our approch in an iterative manner: adding a feature, analysing for FI, resolving the FI, adding again, analysing again, ... If we "switch off" the precedences for the second round of analysis, we will be told all the potential feature interactions that we already know from the first round, and which we have resolved since long.

Therefore, we need to maintain a matrix of pairs [19] of features which already have

[19]This approach does not take care of feature interactions which only happen among three or more features. But these seem to be so rare that no example is known up to now.

been analysed. This "checked"-matrix can be integrated into the precedence matrix. The existence of an entry simply indicates that an analysis has been performed already. We only need an additional symbol that marks those pairs for which no precedence is needed, but which have been analysed. And we need the convention that we only enter entries where an analysis has taken place, and don't enter them only because we know that the precedence relation is transitive. This is necessary because the "checked" relation is not transitive. Nevertheless, it seems sensible to integrate both matrices into a single one because of the strong semantic dependence of the "checked"-matrix on the precedence matrix.

5 Using Existing FDTs

Up to now, we introduced a generic formalism for the specification of telecommunications systems, and we included only those language aspects which were necessary for our discussion. To make our ideas applicable, we need a full-fledged formal description technique which offers the appropriate expressive power that allows a system developer to specify real systems. Our ideas have to be adapted for such a FDT. But we designed our generic formalism in such a way that this adaption is supported well. E.g., the FDT Estelle is rather rich in language constructs, but its semantic model ([ISO89]) is very similar to our global automaton. Therefore it is possible to find syntactic criteria in Estelle for possible feature interactions that correspond to our definitions in the generic formalism. We have already investigated the notion of non-determinism between Estelle transitions in depth [The95]. The richness of Estelle provided a rather large number of special cases to consider, but this only justified our decision to investigate appropriate feature interaction criteria in a simpler formalism first and to apply them to real FDTs later.

An application to the FDT SDL should pose no fundamental problems as well since the semantic models of SDL and Estelle are very similar. Concerning the FDT LOTOS, our simple global automaton A^s is basically already a labelled transition system which is the semantic model of LOTOS.

6 Tools and Case Studies

A number of tools for the FDT Estelle is in use now:

- a preprocessor which allows to extract a specification consisting of exactly the desired features from a common definition,
- a tool to compute priority values from a precedence matrix and to generate the appropriate Estelle code for the value definitions,

- a tool to detect non-determinism between Estelle transitions of different features (Confine, [ThBr95, The96]),

- a tool to detect and automatically eliminate Estelle transitions which have become non-executable because they are completely overlapped by new, higher-priorized Estelle transitions (Confine again), and

- a toolset to automatically generate executable code from an Estelle specification, and to animate the execution (Pet/Dingo from NIST, [SiSt93]).

A first case study [BrGo94a][20] used a simple global service specification of a telephone switching system. Our detection tool Confine did not only find the already known feature interactions but also two interferences which escaped us while we specified this simple example. A manual analysis led to the conclusion that both cases are harmless if an implementation is sufficiently fast. Nevertheless, we achieved a deeper understanding of some underlying problems. Furthermore, Confine detected and removed automatically the inactive transitions which had become obsolete through the resolution procedure for the known feature interactions. Currently, we are working on a second case study which is based on a simplified version of the IN conceptual model [ITU92], specified in Estelle again, which takes into account the distribution aspect, too [Ill95, Jer96]. Parallely, we work on incorporating more criteria into our detection tool [Bar96].

7 Summary and Future Work

The feature interaction problem in telecommunications systems currently obstructs the evolution of such systems more and more. We presented an approach for the offline detection of feature interactions which is different from other verification approaches. It does not rely on a high-level, property oriented description against which a lower-level, constructive description is checked. We observed that the behaviour of a feature is a central notion, and that feature interactions are indicated by a change in this behaviour. We formalized the notion of change in a definition of feature interactions. From this, we derived a necessary (but less than sufficient) condition for feature interactions. It can be checked mechanically and points out all potentially critical spots. These have to be analysed manually. We proposed further criteria to exclude some cases where the interactions are harmless, and we made use of our knowledge of typical telecommunications systems to define criteria for more cases in which there cannot be any negative influence even if the basic criterion indicates a possible problem. This led to a restricted reachability analysis.

[20] An overview of this study is more widely available [BrGo94b].

Since the notions of feature and feature interaction, as they are widely used, had been too fuzzy to have hope for a resolution of the problem, we had to start out with a formal definition, and we discussed which aspects have been included in the formalization (and therefore in the detection method). E.g., we restricted ourselves to the functional aspects of a system. This allowed us to use the computations (possible execution sequences) of a global automaton to catch the notion of behaviour formally. We formalized a feature as an increment in the possible behaviour of a global automaton. A specific specification style was a prerequisite to construct an unambiguous association of the state transitions to the different features.

The result of our analysis is a list of pairs of transitions from different features which may possibly interact. This list may be condensed to a list of pairs of features which may possibly interact. Such a condensed list is of value for the management of detected possible feature interactions: for each entry, we need to find an expert who knows both features and can determine how the interaction is resolved best. This is an improvement to the current situation, where we need an expert who knows *all* features in the system if we want to add one new feature. And the larger the system becomes, the harder it is to find such an overall expert.

Next, we sketched an approach for the resolution of detected feature interactions, which we have detailed in [BrGo94a]. We put up a precedence matrix for features and thereby specify a partial order on them. This allows us to disable unwanted transitions. There is also the possibility to specify new behaviour to resolve an interaction. Furthermore, we discussed how our generic formalism can be mapped onto real FDTs.

Some of the detection criteria are already supported by an automated tool. A first case study based on the FDT Estelle used a simple global service specification of a telephone switching system. Our detection tool did also find two (relatively harmless) interferences which escaped us while we specified this simple example. Currently, we are working on a second case study which takes into account the distribution aspect, too.

This report describes on-going work. We are working on even more detailed detection criteria. They would reduce the manual work of, e.g., dismissing certain classes of harmless interactions. Parallely, we work on incorporating more criteria into our detection tool, and we will continue to perform case studies. The application of our method to other FDTs than Estelle (e.g., SDL) should be straightforward.

References

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

Foundations of Software Engineering" (1995).

[Bar96] Barthel, D. *Implementation of criteria for the detection of feature interactions.* Master thesis (in German), Univ. of Kaiserslautern, Dept. of Comp. Sci. (1996). To appear.

[BCN95] Bucci, G., Campanai, M., and Nesi, P. *Tools for specifying real-time systems.* Real-Time Systems Journal **8**, 117–172 (1995).

[Bo$^+$89] Bowen, T. F. et al.. *The feature interaction problem in telecommunication systems.* In "Seventh IEEE International Conference on Software Engineering for Telecommunication Systems" (July 1989).

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

[Bre95] Bredereke, J. *Formal criteria for feature interactions in telecommunications systems.* In Nørgaard, J., editor, "IFIP International Working Conference on Intelligent Networks, Proceedings", pp. 83–97, Copenhagen, Denmark (28–31 Aug. 1995). IFIP TC6.

[BrGo94a] Bredereke, J. and Gotzhein, R. *A case study on specification, detection and resolution of IN feature interactions with Estelle.* Tech. Rep. 245/94, Univ. of Kaiserslautern, Dept. of Comp. Sci. (May 1994).

[BrGo94b] Bredereke, J. and Gotzhein, R. *Specification, detection and resolution of IN feature interactions with Estelle.* In Hogrefe and Leue [HoLe94], pp. 366–368.

[Cai92] Cain, M. *Managing run-time interactions between call-processing features.* IEEE Commun. Mag. **30**(2), 44–50 (Feb. 1992).

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

[CGL$^+$93] 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.* IEEE Commun. Mag. pp. 64–69 (Mar. 1993).

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

[Fac95] Faci, M. *Detecting Feature Interactions in Telecommunications Systems Designs.* PhD thesis, University of Ottawa, Dept. of Comp. Sce. (1995).

[GoBr95] Gotzhein, R. and Bredereke, J., editors. *5. GI/ITG Workshop on Formal Description Techniques for Distributed Systems*, Univ. of Kaiserslautern, Dept. of Comp. Sci. (22–23 June 1995). URL http://www.informatik.uni-kl.de/aggotz/fachgespraech95.

[GrVe93] Griffeth, N. D. and Velthuijsen, H. *Reasoning about goals to resolve conflicts*. In "Proceedings of the International Conference on Intelligent and Cooperative Information Systems", Rotterdam (May 1993).

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

[HoLe94] Hogrefe, D. and Leue, S., editors. *Seventh International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols — FORTE '94, Proceedings*, Berne, Switzerland (4–7 Oct. 1994).

[HoPe94] Holzmann, G. J. and Peled, D. *An improvement in formal verification*. In Hogrefe and Leue [HoLe94], pp. 177–191.

[HoUl79] Hopcroft, J. E. and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley (1979).

[Ill95] Illerich, J. *Design of a telephone switching system in Estelle*. Projektarbeit (in German), Univ. of Kaiserslautern, Dept. of Comp. Sci. (Oct. 1995).

[ISO89] ISO/TC 97/SC 21, ISO 9074. *Information Processing Systems — Open Systems Interconnection — Estelle: A Formal Description Technique Based on an Extended State Transition Model* (1989).

[ITU92] ITU-T, Recommendation Q.1201. *Principles of Intelligent Network Architecture* (Oct. 1992).

[ITU93a] ITU-T, Recommendation Q.931. *DSS 1 — ISDN User-Network Interface for Basic Call Control* (Mar. 1993).

[ITU93b] ITU-T. *Q12xx-Series Intelligent Network Recommendations* (1993).

[Jer96] Jerusalem, D. *Extension of a telephone switching system in Estelle*. Projektarbeit (in German), Univ. of Kaiserslautern, Dept. of Comp. Sci. (1996). To appear.

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

[LiLi94] Lin, F. J. and Lin, Y.-J. *A building block approach to detecting and resolving feature interactions*. In Bouma and Velthuijsen [BoVe94], pp. 86–119.

[SCO95] SCORE-Technology Transfer. *Service Creation in an Object-Oriented Reuse Environment*. Deliverable D409 – R2017/SCO/WP4/DS/P/029/b1, RACE Project 2017 (SCORE) (4 Jan. 1995).

[SiSt93] Sijelmassi, R. and Strausser, B. *The PET and DINGO tools for deriving distributed implementations from Estelle*. Comp. Networks and ISDN Syst. **25**(7), 841–851 (Feb. 1993).

[Sri95] Srinivasan, S. *Impact of enhanced feature interactions*. IEEE Commun. Mag. **33**(1), 100–102 (Jan. 1995).

[ThBr95] Thees, J. and Bredereke, J. *A tool for the analysis of feature interactions in IN*. In Gotzhein and Bredereke [GoBr95], pp. 199–208. URL http://www.informatik.uni-kl.de/aggotz/fachgespraech95/thees.ps.gz (in German).

[The95] Thees, J. *Design and implementation of a tool for the analysis of feature interactions in Estelle specifications*. Master thesis (in German), Univ. of Kaiserslautern, Dept. of Comp. Sci. (Apr. 1995). URL http://www.informatik.uni-kl.de/aggotz/thees/diplomarbeit.ps.gz.

[The96] Thees, J. *Confine – A Tool for the Analysis of Estelle Specifications (Tutorial)*. Univ. of Kaiserslautern, Dept. of Comp. Sci. (in German) (1996).

[Vel93] Velthuijsen, H. *Distributed artificial intelligence for runtime feature-interaction resolution*. IEEE Comp. **26**(8), 48–55 (Aug. 1993).

A Auxiliary Definitions

This appendix contains some definitions which are necessary for a strict formalization in the main part but for which the idea should be clear anyway. So the actual, long formalization would only distract the casual reader from more important aspects.

A.1 For "State Space Dependence"

The definition of a state space dependence (Def. 3.3 on page 19) depended on three auxiliary notions which were left out there:

The *output modification pattern* shall describe which local state components can possibly be modified by a transition.

Definition A.1 (output modification pattern)
The output modification pattern O^t of a transition $t = (S^e, \sigma, \lambda)$ of a global automaton $A = (S^G, T, s_0)$, $t \in T$, is a tuple like a global state (Def. 2.2), but the local states are replaced by subsets of index ranges. For all i, j with $1 \leq i \leq n$ and $1 \leq j \leq m_i$ holds that $j \in O^t_i$ iff there exists a simple transition $(s_1, s_2) \in \sigma$ with $s_1 \in S^e$ such that $s_{1_{i,j}} \neq s_{2_{i,j}}$ (i.e., the component j of the first local state $s^l_{1_i}$ differs from the component j of the second local state $s^l_{2_i}$).

Example A.1.1

We illustrate this definition by a very simple automaton
$A = (S^G,\ \{t\},\ s_0)$
with a global state space
$S^G = \{(\clubsuit), (\diamondsuit)\} \times (\{\spadesuit, \heartsuit\} \times \{\spadesuit, \heartsuit\})$
an initial state
$s_0 = ((\clubsuit),\ (\spadesuit, \spadesuit))$
and a (single) transition
$t = (S^G, T^s, (B, \zeta))$
which has got a set of simple transitions
$T^s : ((x), (y, z)) \to ((x), (\heartsuit, y)) \quad \text{for } x \in \{\clubsuit, \diamondsuit\},\ y, z \in \{\spadesuit, \heartsuit\}$

For clarification of the above notation, we repeat the definitions of the global state space S^G and the set of simple transitions T^s in long form:

$$\begin{aligned}
S^G = \{\ & ((\clubsuit), (\spadesuit, \spadesuit)),\ ((\clubsuit), (\spadesuit, \heartsuit)),\ ((\clubsuit), (\heartsuit, \spadesuit)),\ ((\clubsuit), (\heartsuit, \heartsuit)),\\
& ((\diamondsuit), (\spadesuit, \spadesuit)),\ ((\diamondsuit), (\spadesuit, \heartsuit)),\ ((\diamondsuit), (\heartsuit, \spadesuit)),\ ((\diamondsuit), (\heartsuit, \heartsuit)))\ \}\\
T^s = \{\ & (((\clubsuit), (\spadesuit, \spadesuit)),\ ((\clubsuit), (\heartsuit, \spadesuit))),\\
& (((\clubsuit), (\spadesuit, \heartsuit)),\ ((\clubsuit), (\heartsuit, \spadesuit))),\\
& (((\clubsuit), (\heartsuit, \spadesuit)),\ ((\clubsuit), (\heartsuit, \heartsuit))),
\end{aligned}$$

$$
\begin{array}{l}
(((\clubsuit),\ (\heartsuit,\heartsuit)),\ \ ((\clubsuit),\ (\heartsuit,\heartsuit))),\\
(((\diamondsuit),\ (\spadesuit,\spadesuit)),\ \ ((\diamondsuit),\ (\heartsuit,\spadesuit))),\\
(((\diamondsuit),\ (\spadesuit,\heartsuit)),\ \ ((\diamondsuit),\ (\heartsuit,\spadesuit))),\\
(((\diamondsuit),\ (\heartsuit,\spadesuit)),\ \ ((\diamondsuit),\ (\heartsuit,\heartsuit))),\\
(((\diamondsuit),\ (\heartsuit,\heartsuit)),\ \ ((\diamondsuit),\ (\heartsuit,\heartsuit)))\ \}
\end{array}
$$

In this simple example automaton, the resulting output modification pattern of the above defined transition t is:

$$O^t = (\{\},\ \{1,2\})$$

$\square$

The *output dependence pattern* shall describe for each output local state component (k,l) on which input local state components (i,j) it depends.

Definition A.2 (output dependence pattern) *The output dependence pattern D^t of a transition $t = (S^e, \sigma, \lambda)$ of a global automaton $A = (S^G, T, s_0)$, $t \in T$,* is structured like a global state (Def. 2.2), but the local state components are replaced by subsets of pairs of indices. For all i, j and k, l with $1 \le i \le n$ and $1 \le j \le m_i$ and $1 \le k \le n$ and $1 \le l \le m_k$ holds that $(i,j) \in D^t_{k,l}$ iff there exist two global states $s_1, s_1' \in S^e$ such that: s_1' is equal to s_1 except for the component j of the local state i and there exists a global state $s_2 \in S^G$ with $s_1 \sigma s_2$ such that for all[21] global states $s_2' \in S^G$ with $s_1' \sigma s_2'$ holds that $s'_{2_{k,l}} \neq s_{2_{k,l}}$.

Example A.2.1 We reuse the global automaton A and the transition t of the previous Example A.1.1. The resulting output dependence pattern of transition t is:

$$D^t = \Big(\big(\{(1,1)\}\big),\ \big(\{\},\{(2,1)\}\big)\Big)$$

$\square$

Note that there are cases where Definition A.2 may behave differently as expected: e.g., if the enabling condition of a transition is such that two local state space components (i_1, j_1) and (i_2, j_2) have to be equal for the transition to fire, then there cannot be any state space dependence on these components. But this does not matter since both components will be part of the input pattern defined below anyway.

[21]This universal quantification with the following negation of the equality is necessary because of the possible non-determinism of the transition.

The *input pattern* shall describe which local state components (i, j) are used at all by a transition. This comprises both the enabling conditions and the computation of the output. Output dependences on the input in the fashion of the identity function are not counted.

Definition A.3 (input pattern)
The input pattern I^t of a transition $t = (S^e, \sigma, \lambda)$ of a global automaton $A = (S^G, T, s_0)$, $t \in T$, is a tuple like a global state (Def. 2.2), but the local states are replaced by subsets of index ranges. For all i, j with $1 \leq i \leq n$ and $1 \leq j \leq m_i$ holds that $j \in I_i^t$ iff:

- there exist two global states $s_1, s_2 \in S^G$ where $s_1 \in S^e$ and $s_2 \notin S^e$ which are equal except for the component j of the local state i, which is not equal,

  and/or
- there exists a local state component (k, l) such that
  - $(i, j) \in D_{k,l}^t$ and
  - $(k, l) \neq (i, j)$ or there exists a simple transition $(s_1, s_2) \in \sigma$ with $s_1 \in S^e$ such that $s_{1_{i,j}} \neq s_{2_{i,j}}$.

Example A.3.1 We reuse the global automaton A and the transition t of the previous Examples A.1.1 and A.2.1. The resulting input pattern of transition t is:

$$I^t = (\{\},\ \{1\})$$