Requirements for Evolving Systems: A Telecommunications Perspective

Pamela Zave
AT&T Laboratories—Research
Florham Park, New Jersey, USA
pamela@research.att.com
http://www.research.att.com/info/pamela

Abstract

In many software application domains, constant evolution is the dominant problem, shaping both software design and the software process. Telecommunication software is the prototypical example of such an application domain. This paper examines how requirements engineering, formal description techniques, and formal methods should be adapted to work well in these application domains.

1 Introduction

In many software application domains, constant evolution is the dominant problem, shaping both software design and the software process. Telecommunication software is the prototypical example of such an application domain.

This paper examines the early phases of software engineering for application domains dominated by change. What is the best form of software engineering for these domains? Is it different from software engineering for other domains? I shall attempt to answer these questions by drawing on the history of software development in telecommunications, and the challenges its developers face today.

This paper has a two-dimensional structure, displayed in Table 1. Section 2 covers the state of practice in telecommunications, including both what people do and what they need. Section 3 summarizes some relevant results from research in formal methods. Section 4 combines the two in suggestions for requirements analysis, formal description techniques, and formal methods for validation and verification.

The subsections of the paper follow two classic themes. The first subsection of each section is concerned with questions of modularity: Why and how should formal descriptions be decomposed into modules? What are the consequences? The second subsection of each section is concerned with questions of level of abstraction: Which phenomena are described by a formal description, and which are left out? How can the phenomena relevant to software development be arranged in orderly layers? Ultimately I attempt to weave all these subjects and themes into a coherent whole.

2 The State of Practice in Telecommunications

The heart of *telecommunications* is real-time, person-to-person communication at a distance (by electronic means). Considering all its inter-operating parts together, the telecommunication network is world-wide and approximately 125 years old. Despite its venerable age, it is now evolving even more rapidly than usual. The influence of the Internet is bringing packet switching to compete with circuit switching. It is extending personal communication to include media other than voice, such as text, images, and video. It is also extending personal communication to include new modes of interaction such as mail and browsing.

2.1 Feature-Oriented Description

The behavior of telecommunication software is almost always described in terms of *features.* A *feature* of a software system is an optional or incremental unit of functionality. A *feature-oriented description* consists of a base description and feature modules, each of which describes a separate feature. The set of possible system behaviors is determined by applying a feature-composition operator to the base description and these modules.

The big attraction of feature-oriented description is *behavioral modularity,* which makes it possible to change the system's behavior easily. With *perfect* behavioral modularity, it would be possible to make *any* desired change to the behavior of a system by composing a new feature module with the existing system description; it would never be necessary to change existing modules.

Even though perfect modularity will never be achieved, it must be approximated to a significant degree. For ex-

	modularity	**level of abstraction**
2. The State of Practice in Telecommunications	2.1. Feature-Oriented Description	2.2. Network-Independent Description
3. Formal Methods for System Description	3.1. Global Proof Obligations	3.2. Specification versus Architectural Description
4. Prescriptions for Evolving Systems	4.1. Feature Engineering	4.2. Component Architectures

Table 1. Sections and themes in this paper.

ample, it is extremely common to create an exception to an existing feature by adding a new feature rather than by changing the existing feature.

Such an exception is an example of a *feature interaction.* A *feature interaction* is some way in which a feature or features modify or influence another feature in describing the system's behavior set. Formally this influence can take many forms, depending on the nature of the feature-description language and composition operator. A group of logical assertions, composed by conjunction, can affect each other's meanings rather differently than a group of finite-state machines, composed by event synchronization.

In general, features might interact by causing their composition to be incomplete, inconsistent, nondeterministic, or unimplementable in some specific sense.[1] Or the presence of a feature might simply change the meaning of another feature with which it interacts.

Feature-oriented description emphasizes individual features and makes them explicit. It also de-emphasizes feature interactions, and makes them implicit in the effects of the feature-composition operator.

Two points about feature interactions are frequently misunderstood, despite extensive research on this subject [6, 7, 8, 9, 17]. Since these misunderstandings make it impossible to talk about feature interaction clearly, let alone formally, it is important to emphasize these points:

- While many feature interactions are undesirable, many others are desirable or necessary. *Not all feature interactions are bad!*
- Feature interactions are an inevitable by-product of behavioral modularity.

These points are exemplified by "busy treatments" in telephony, which are features for handling busy situations. Suppose that we have a feature-description language in which a busy treatment is specified by providing an action, an enabling condition, and a priority. Further suppose that a special feature-composition operator ensures that, in any busy situation, the single action applied will be that of the highest-priority enabled busy treatment.

In a busy situation where two busy treatments B_1 and B_2 are both enabled, with B_2 having higher priority, these features will interact: the action of B_1 will not be applied, even though its stand-alone description says that it should be applied. This feature interaction is intentional and desirable. It is a by-product of the behavioral modularity that allows us to add busy treatments to the system without changing existing busy treatments. Without the special composition operator, when B_2 is added to the system, the enabling condition E_1 of B_1 must be changed to $E_1 \wedge \neg E_2$.

Most feature-oriented descriptions are still informal—the feature modules are written in natural language, and the feature-composition operator is concatenation of the text. Any desired change to system behavior is easy to make: just describe the change in natural language, whatever it is. On the other hand, the description is neither comprehensible nor analyzable overall. It rarely defines the system's behavior in a complete, consistent, and unambiguous manner, especially since a feature-oriented style is an invitation to ignore feature interactions altogether. This situation affects all segments of the telecommunication industry, and is the primary motivation for the industry's interest in formal methods.

The telecommunication industry needs feature-oriented descriptions that are also formal. Telecommunication engineers need formal methods that help them manage feature interactions, rather than ignore them until they must be dealt with in an *ad hoc* manner.

This has proven to be difficult, as behavioral modularity and formality do not combine easily. The history of research on description of telecommunication systems [6, 7, 8, 9, 17] records many types of feature composition, each illustrated with a handful of features, but few that seem applicable to telecommunication systems of realistic size, with hundreds or even thousands of features.

So far, consciousness of the feature-interaction problem is largely confined to people with experience in circuit-switched telephony. The IP community interested in telecommunications [3, 18] tends to view the present and

[1] For example, in TLA [1] the result of feature composition could fail to be *machine-closed.* The specification would be unimplementable because it requires the system to control the environment's choices.

future in terms of highly complex, yet stand-alone, services. Due to the immaturity of IP telecommunications, not many people have grappled with the realities that there is no such thing as a stand-alone telecommunication service, and that once people start using a service, it becomes a legacy whose evolution must be managed.

2.2 Network-Independent Description

Even before packet switching became part of the picture, transmission technology for telecommunications was evolving rapidly. Features were programmed into switches, so that replacement of a switch entailed replacement of its feature code. This was an untenable situation, and motivated development of the Intelligent Network (IN) architecture [10, 11, 16]. The IN architecture separates feature code from switches, so that switches can be upgraded without altering feature code; it has had a tremendous influence on the telecommunication industry.

Behavioral description of telecommunication services is concerned with such concepts as customer requirements, endpoint device user interfaces, and telecommunication customs. Networking is concerned with such concepts as transmission technology, resource allocation, and distribution of data. The history of telecommunications shows clearly that these concerns must be separated. There should be a clean, robust interface between them, so that each can continual evolving independently. Each is so complex, and evolves so fast, that close coupling between them creates insuperable problems.

The recent development of IP networks also supports this conclusion. Current transmission technology is not stable, and a mass conversion to optical networks is forseeable. Resource allocation for quality of service is an extremely contentious issue. There is much research on the distribution, replication, and synchronization of data. Thus it is more important than ever to describe the behavior observed by users in ways that do not constrain or rely on network architecture.

Although the IN architecture purports to define an interface between service behavior and networking, it is too limited even for traditional telephony services [22], let alone the multimedia, multimodal services of the future. Thus, IN is not the solution to the pressing problem of network-independent description.

3 Formal Methods for System Description

3.1 Global Proof Obligations

Among the major artifacts of software engineering [27] are:

- *Domain knowledge*, which provides presumed facts about the environment.
- *Requirements*, which indicate what the stakeholders need from the system, described in terms of its desired effect on the environment.

Software engineering also creates one or more descriptions of the behavior of the required system. Some relationships among these descriptions are illustrated by Figure 1.

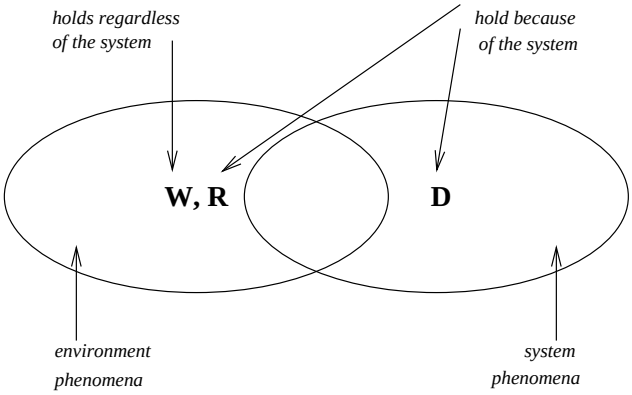

Figure 1. Relationships among formal descriptions. W ("world") is domain knowledge, R is requirements, and D is any description of system behavior.

A recent reference model of the proof obligations for formal methods of software development ([12], based on [13, 15, 27]) proposes three proof obligations. The first proof obligation, loosely paraphrased, says that the domain knowledge must be consistent or satisfiable. Since the real world cannot actually be inconsistent, this is a check on the formalization of the system's environment, rather than the environment itself.

The second proof obligation, *very* loosely paraphrased, says that for every possible behavior of the environment, there must be a behavior of the system that is consistent with the system description.[2]

The third proof obligation says that a description of the system, in conjunction with the domain knowledge, must imply the satisfaction of the requirements. This is the most important proof obligation, and the one that receives the most attention.

These proof obligations are relevant to modularity because they tell us what global relationships must hold

[2] The aspects of the proof obligation omitted by this paraphrase concern logical quantification and control of events.

among descriptions, regardless of any modular decomposition used in the descriptions.

For example, in telecommunications **D** is sure to be a feature-oriented description. Yet **W** and **R** contain at least some global constraints that apply to the composition of all features. **W** must include the characteristics of telecommunication devices, which provide the user interface to all features. **R** should include principles of telecommunication privacy, predictable billing, feature integration, and etiquette upon which all users can rely at all times.

3.2 Specification versus Architectural Description

There are two ways of writing a system description **D**. The two ways are equivalent in the sense that the three proof obligations can be stated formally for either of them [12]. They differ significantly in the phenomena they describe, however, and in their practical advantages and disadvantages.

As explained in Section 3.1, domain knowledge and requirements are descriptions of environment phenomena. Obviously a system description must be a description of system phenomena. The interface between system and environment consists of *shared phenomena* that are visible to both. A *specification* is a description of the system's behavior, written strictly in terms of shared phenomena. In other words, a specification mentions no system phenomena that are hidden from the environment (Figure 2).

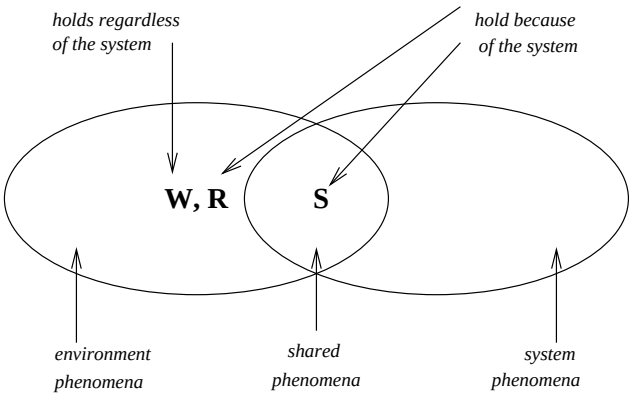

Figure 2. Relationships among formal descriptions. S is a specification.

The major advantage of a specification is that it tells requirements engineers and stakeholders a minimum about the system, and implementors a minimum about the environment. It separates their concerns; it can act as a contract or a firewall between them. On each side of the wall engineers retain maximum freedom to do their job without consulting the other side, and are distracted by a minimum of information that is not significant to them.

The major disadvantage of specifications is that they are difficult to write. For realistically complex systems, if not for academically simple ones, the strict constraint on the phenomena that can be used is a severe problem.

Consider the following trivial example. A telecommunication system receives an input event from a user, and responds with (let us say) one output event. The input and output events are shared phenomena, being visible to users, and therefore legitimate specification phenomena.

The specification must describe which output event is stimulated by the input event. This choice, however, is extremely complex—it is determined by the composition of all the features in the system. We might organize the feature composition by means of a pipeline, where the input event e_i stimulates the first feature. The first feature responds with event e_1, reflecting its view of the situation; e_1 then stimulates the second feature, the second feature responds with event e_2, and so on. The last feature responds with event e_n, which now reflects the joint view of all the features, and this becomes the output event e_o.

Unfortunately, this useful description is not a legitimate specification, because the events e_1 through e_n are not shared phenomena. Rather, they are artifacts of our structuring technique.

Many specification languages provide information hiding to address this problem. Then specifiers can use extra phenomena such as events, yet declare them to be optional in an implementation. However, the information-hiding approach imposes a heavy burden on formal methods: to prove that a true implementation is behaviorally equivalent to the pseudo-implementation in the specification.

The second way of writing a system description employs two other familiar artifacts of software engineering [12]:

- A *programming platform* provides the basis for programming a system to satisfy the requirements.
- A *program* implements the requirements on the programming platform.

These descriptions are not confined to system phenomena that are shared with the environment. Because a programming platform is often referred to as an *architecture*, their combination can be termed an *architectural description.* Their relationships are illustrated by Figure 3.

The major advantage of architectural descriptions is that they are easier to write and implement than specifications. Writers of architectural descriptions can invent any extra phenomena that they find convenient, for organizational or other purposes. And an architectural description is inherently executable.

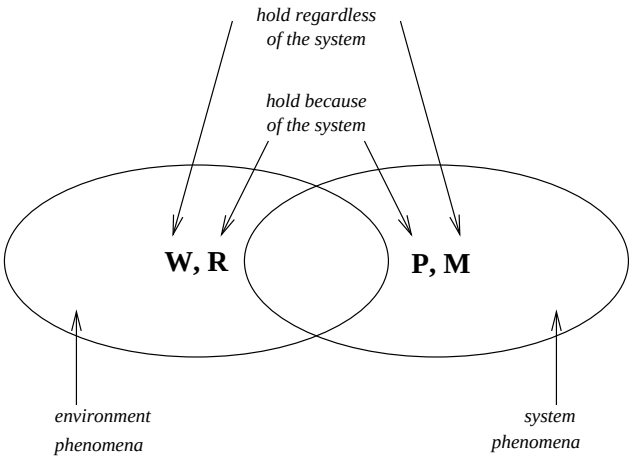

Figure 3. Relationships among formal descriptions. P is a program or programs, and M ("machine") is a programming platform.

The major disadvantage of architectural descriptions is that they do not have the implementation-independence of specifications. The disadvantage is particularly serious if the programming platform is low-level and general-purpose: it will be difficult to program for and reason about.

It is possible to alleviate this disadvantage by describing a more abstract, idealized, domain-specific programming platform. Not only will such a platform be easier to program for and reason about, but it can also separate implementation concerns into orderly layers.

Continuing with the example introduced earlier in this section, a programming platform for telecommunications might support the simple feature-composition mechanism proposed above by letting features run as concurrent processes, and turning the internal events into some appropriate kind of asynchronous interprocess communication. Such a platform would be feature-oriented and distributable. It could also be made network-independent by abstracting away from lower-level networking concerns.

Even in a distributable platform, many lower-level networking concerns can be ignored by eliminating the concept of location. Then there will be no formal representation of which node of a network is running a process or storing some data, let alone where the network nodes are located geographically. Because resource consumption and performance in networks are heavily dependent on location, it is impossible to determine them from a location-free description. Quite separately from the description of system behavior, engineers can study the consequences of various location alternatives on transmission technology, resource consumption, performance, and data consistency.

4 Prescriptions for Evolving Systems

4.1 Feature Engineering

Verification, as an approach to establishing the adequacy of a system description, is only as good as the requirements provided. Requirements for evolving systems present significant difficulties, which can only be overcome with special techniques.

One difficulty is obvious: the behavior of a feature-oriented system changes, sometimes radically, with the addition of every feature. An assertion that is true of today's system but not true of tomorrow's has little value. The gravity of this situation cannot be overestimated. Taking into account all the features of the public switched telephone network (PSTN) that I have seen proposed or implemented, I have not been able to think of a *single* interesting assertion that would be true of a system incorporating all of them.

Of course, the proper attitude toward this difficulty is equally obvious. The PSTN has been shaped by 125 years of technological improvements, most of which were not anticipated when earlier decisions were being made. As a result, it has no inherent behavioral principles. We must use our contemporary knowledge to find behavioral abstractions that will stand the test of time, and use them to define behavioral principles supporting such properties as privacy, predictable billing, feature integration, and telecommunication etiquette. Then we must find ways of encapsulating the legacy features, so that their ability to compromise these principles is contained, and eventually withers away.

A closely related difficulty is that of interoperation: Just as principled features must interoperate with legacy features, a principled system must interoperate with other telecommunication systems over which it has no control. Once again, it is critical to distinguish between what can be proven of a known system, and what can be proven of a known system interacting with an unknown (or partially known) system.

Even the best general principles will constrain a system's behavior only loosely, providing a structure within which feature designers can offer whatever they want. The second requirements difficulty concerns specific feature behavior. To write complete requirements, it would be necessary to state formally and explicitly exactly how all features interact. This is something people tend to do poorly [20], and it is exactly the chore that feature-oriented description was invented to avoid.

A possible solution to this difficulty is a formal method for feature-oriented descriptions that we might call *feature engineering.* Feature engineering is a process of engineering features and feature interactions for maximum user satisfaction and system integrity. Feature engineering works from partial requirements, and develops the missing system

requirements from the bottom up.

Figure 4 is a simple picture of feature engineering. It assumes that a base description and old features already exist, and that the goal is to add new features to the system.

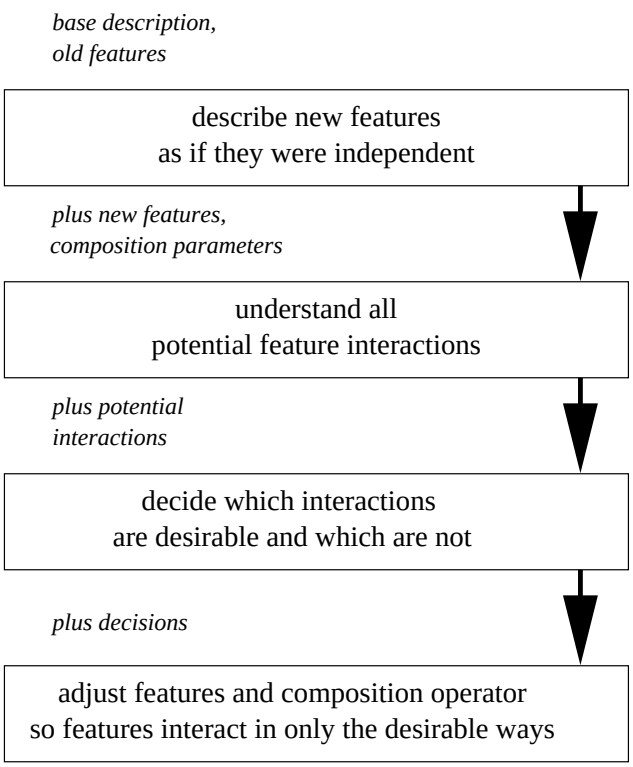

Figure 4. Feature engineering.

In the first step of the process, new feature requirements are added to **R**, and new features are described. To as great an extent as possible, each feature is described as if it were independent of all others. People will want to do this anyway, as there is a strong human tendency to ignore feature interactions. They will be able to do it only if the description technique has sufficient behavioral modularity.

In the second step of the process, engineers must come to understand all potential feature interactions. Feature composition must be structured well enough so that "all potential feature interactions" is a meaningful concept. There must also be automated analysis to detect their presence in a feature set.[3]

For example, consider the following two PSTN features. Call Blocking (CB) is subscribed to by subscriber y, who has configured it to reject all calls from x. Unconditional Call Forwarding (UCF) is also subscribed to by subscriber y, who has configured it to forward all his calls to z.

[3]Note that the feature-composition operator might be parameterized, in which case parameter values might also be fed into the analysis.

Formal description and structured feature composition should reveal that these features have several possible interactions. If CB is given precedence, then all calls from x to y will be blocked, and UCF will not forward all calls to y, but rather only those calls to y that do not come from x. If UCF is given precedence, then all calls to y while UCF is active will be redirected to z, and CB will not reject all calls from x to y, but rather only those calls from x to y placed while UCF is not active.

In the third step of the process, engineers classify potential interactions as bad or good. This step is primarily manual. As engineers gain experience and insight, they might discover general principles that govern some of their choices. Such principles can feed into automated assistance for this step, or even be elevated to the status of requirements.

Concerning the PSTN example, most engineers would probably conclude that the first feature interaction is good and the second is bad. This is by no means certain, however, as there are many factors to take into account [24].

In the fourth step of the process, feature descriptions are adjusted so that all of the good feature interactions are present and none of the bad ones are. If the description technique has sufficient behavioral modularity, only the composition parameters or descriptions of new features need be changed, and all the old features can remain untouched. If all its steps are sound, the result of feature engineering is that the requirements governing how features interact have been elicited, and the final behavioral description of the system is guaranteed to satisfy them.

4.2 Component Architectures

To summarize the conclusions so far, telecommunication systems need behavioral descriptions that are formal, feature-oriented, and network-independent. In addition to whatever general-purpose formal methods these descriptions support, they must support feature engineering by offering structured feature composition and a useful degree of behavioral modularity.

The best way to achieve all these goals is with formal system descriptions that are both feature-oriented and architectural. More specifically, they should be based on a *component architecture*, which is an architecture in which new functions can be added freely by adding component programs. As Figure 5 shows, feature modules and component programs can be the same things.

The first reason for this recommendation is that, within the research community studying feature interaction, there is a definite trend toward architectural descriptions [2, 5, 10, 11, 16, 14, 19, 21, 23, 25, 26]. They are by far the most successful in delivering behavioral modularity. This is not surprising; as the example in Section 3.2 shows, one of the

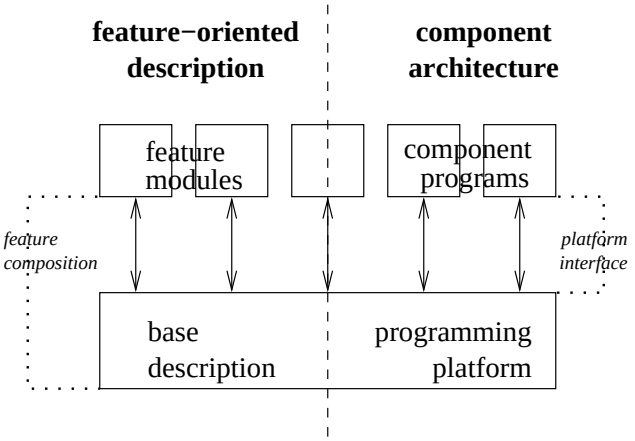

Figure 5. The same system description can be viewed as both feature-oriented and architectural.

helpful structures that can be created with a free choice of internal system phenomena is behavioral modularity.

A second reason for the recommendation is that the architectural structure that defines and constrains feature composition (component interaction) also provides a basis for enumerating, classifying, and detecting potential feature interactions. This is essential for feature engineering.

Despite its positive qualities, architectural description should not be recommended without at least considering its drawbacks, in particular the lack of separation between behavioral and implementation concerns.

There are many reasons why this drawback has little significance in the context of evolving systems, particularly telecommunication systems. To begin with, it is clear from Section 3.2 that implementation-dependence need not extend all the way down to network-dependence.

It is also important to consider the process of developing and managing an evolving system. One of the best ways to achieve timely evolution is to maintain an open system—one to which many parties contribute. Contributors to an open system are likely to be doing both requirements engineering and implementation, so they don't need a separation of those concerns. They do need a well-defined platform on which to program, one that elucidates the semantics of each component, helps different components cooperate within the overall system structure, and protects the integrity of the system from errant components.

These observations apply directly to telecommunications. For many years, providers of telecommunication services have been frustrated with the slow pace of feature development in vendors' proprietary equipment. Because they want to speed up the process by developing their own features or getting third parties to do it, they have been demanding from the vendors of telecommunication equipment that their systems be open.

5 Conclusion

All of these ideas are embodied in the Distributed Feature Composition architecture [14, 21, 23, 25, 26], which is a component architecture for the description of telecommunication services, designed for generality, feature modularity, structured feature composition, and analysis of feature interactions. An IP implementation of DFC [4] provides an existence proof for the plausibility of these ideas.

I have attempted to justify the conclusions of this paper with well-established observations about the telecommunication industry and telecommunication software. The intention is to provide plenty of background information—enough so that others can decide whether the recommendations apply to software development for other application domains dominated by change.

References

[1] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems* XV(1):73-132, January 1993.

[2] Pansy K. Au and Joanne M. Atlee. Evaluation of a state-based model of feature interactions. In [9], pages 153-167.

[3] Greg Bond and Eric Cheung, editors. *Proceedings of the IP Telecom Services Workshop 2000.* Atlanta, Georgia, September 2000.

[4] Greg Bond, Eric Cheung, Andrew Forrest, Michael Jackson, Hal Purdy, Chris Ramming, and Pamela Zave. DFC as the basis for ECLIPSE, an IP communications software platform. In [3].

[5] Kenneth H. Braithwaite and Joanne M. Atlee. Towards automated detection of feature interactions. In [6], pages 36-57.

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

[7] M. Calder and E. Magill, editors, *Feature Interactions in Telecommunications and Software Systems VI.*, IOS Press, Amsterdam, 2000.

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

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

[10] José M. Duran and John Visser. International standards for intelligent networks. *IEEE Communications* XXX(2):34-42, February 1992.

[11] James J. Garrahan, Peter A. Russo, Kenichi Kitami, and Roberto Kung. Intelligent Network overview. *IEEE Communications* XXXI(3):30-36, March 1993.

[12] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software* XVII(3):37-43, May/June 2000.

[13] Michael Jackson and Pamela Zave. Deriving specifications from requirements: An example. In *Proceedings of the Seventeenth International Conference on Software Engineering*, pages 15-24. ACM Press, ISBN 0-89791-708-1, 1995.

[14] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering* XXIV(10):831-847, October 1998.

[15] Michael Jackson and Pamela Zave. Domain descriptions. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 56-64. IEEE Computer Society Press, ISBN 0-8186-3120-1, 1992.

[16] Jalel Kamoun and Luigi Logrippo. Goal-oriented feature interaction detection in the Intelligent Network model. In [17], pages 172-186.

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

[18] Henning Schulzrinne, editor. *Proceedings of the Second IP-Telephony Workshop.* Columbia University, New York, New York, April 2001.

[19] Greg Utas. A pattern language of feature interaction. In [17], pages 98-114.

[20] Hugo Velthuijsen. Issues of non-monotonicity in feature-interaction detection. In [8], pages 31-42.

[21] Pamela Zave. An architecture for three challenging features. In *Proceedings of the Second IP Telephony Workshop*, Columbia University, New York, New York, April 2001.

[22] Pamela Zave. 'Calls considered harmful' and other observations: A tutorial on telephony. In Tiziana Margaria, Bernhard Steffen, Roland Rückert, and Joachim Posegga, editors, *Services and Visualization: Towards User-Friendly Design*, pages 8-27. Lecture Notes in Computer Science 1385, Springer-Verlag, 1998.

[23] Pamela Zave. An experiment in feature engineering. In *Essays by the Members of the IFIP Working Group on Programming Methodology*, Springer-Verlag, to appear.

[24] Pamela Zave. Secrets of call forwarding: A specification case study. In *Formal Description Techniques VIII (Proceedings of the Eighth International IFIP Conference on Formal Description Techniques for Distributed Systems and Communications Protocols)*, pages 153-168. Chapman & Hall, ISBN 0-412-73270-X, 1996.

[25] Pamela Zave and Michael Jackson. DFC modifications I (Version 2): Routing extensions. AT&T Laboratories Technical Report, January 2000.

[26] Pamela Zave and Michael Jackson. DFC modifications II: Protocol extensions. AT&T Laboratories Technical Report, November 1999.

[27] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology* VI(1):1-30, January 1997.