THE FEATURE INTERACTION PROBLEM
IN TELECOMMUNICATION SYSTEMS

T. F. Bowen  F. S. Dworak  C. H. Chow
N. D. Griffeth  G. E. Herman  Y.-J. Lin

Bellcore
Morristown NJ 07962
March 17, 1989

1 INTRODUCTION

Since the deployment of Stored Program Control (SPC) switching offices approximately 25 years ago, the development of software intensive telecommunications features has required development times which are, according to some telephone administrations, an order of magnitude greater than allowable for competitive offerings. The development of telecommunications features is difficult for many of the same reasons that software development in the large is difficult. Like other large scale software, telecommunications systems are large, imprecisely specified, and have long lives during which they undergo extensive evolution. Telecommunications services also have unique requirements, such as high availability and real-time response, which complicate the software architecture. All these factors contribute to making telecommunication software productivity approximately one-fourth of that for other systems[20]. In the future, the need for more rapid feature introduction requires separation of applications from the system, allowing multiple vendors to concurrently develop features. This paper explores what we believe is one major contributor to the existing lengthy development intervals, and a key obstacle to concurrent development of features by multiple vendors: the feature interaction problem.

Features are packages of incrementally added functionality providing services to subscribers or the telephone administration. The number of features accumulated during the life of a mature switching office is currently in the hundreds[4], and includes features visible to subscribers, such as call forwarding, and features visible only to the telephone administration, such as emergency priority line service. Feature interaction occurs when one feature interferes with the
operation of another feature. The feature interaction problem dominates the development period with efforts to evaluate and resolve potential feature interactions, which must increase exponentially with the number of features. The underlying problem is the inability to structure telecommunications systems so that new features can be developed without considering all existing features. To illustrate the magnitude of the problem, consider that if only 20 seconds are required to resolve the interactions for each possible combination of 20 features, then more than two years of effort (assuming 8 hour days) is consumed resolving feature interactions. Obviously, feature developers don't consider all possible interactions, but use insight and experience to reduce the problem space.

In other domains, for example database systems, applications are well isolated from each other, allowing addition of new applications without considering existing applications. The ideal solution would allow features to be developed independently and ensures correct interactions with other features. But this solution may not be achievable because features seem inherently interrelated. The need for this kind of solution is magnified by the potential for feature development by multiple vendors.

Current research, as surveyed by this paper, suggests two approaches to the feature interaction problem. One approach tries to solve the fundamental problem by restructuring systems as features are added to avoid exponential complexity growth. The other approach doesn't reduce the system's complexity but provides tools, most of which exist currently in only prototype form, to shield developers from the complexity. A third, largely unexplored approach is to restructure the problem itself, so that the entities being added are not features, but non-interacting applications. The facet of the problem that none of these approach adequately handles is termed dynamic feature interaction and covers interactions which obey the consistency requirements of the system model, but still result in undesirable system behavior.

As a framework for studying feature interactions we first discuss their effects on customers and developers and then consider tools and methods that exist or are emerging and that reduce their impact on the development process.

2 FEATURE INTERACTION EFFECTS

2.1 Effects on Subscribers

Feature interactions have both positive and negative effects on subscribers. Synergy between features sometimes allows subscribers to obtain service beyond the scope of either feature individually, or may cause the system to provide inadequate service. For example, the negative interaction between call-waiting and call-forwarding-busy-line prevents calls from being forwarded, while a more useful interaction allows both features to execute concurrently and completes a call to the first party answering it. Similarly, if a customer with call-again-on-busy
calls a customer that has call-waiting, a proper interaction is to restrict the call-again-on-busy feature, while the negative interaction restricts call-waiting. Feature interaction can result in customer confusion, for example, if customer A calls customer B whose calls are forwarded to customer C, and C has a feature that returns the last call, the returned call might either reach A, who had been trying to call B, not C; or reach B, who had not called C at all! The least confusing resolution of this interaction may be restricting the return last call feature for forwarded calls.

2.2 Effects on Developers

Feature interactions are a problem because their resolution lengths the development process. Development begins after general feasibility, marketing, regulatory and other analysis determines that a feature should be provided. Development typically consists of iterations of specification, design, implementation, and testing, with specification and testing intervals accounting for the largest share of the development interval. The feature interaction problem increases the effort in each development phase.

During specifications, feature interactions lead to two problems:

1. While it is usually easy to specify the logical behavior of isolated features, specifying their interactions with other features is difficult, since the other features and their interactions must be understood and complicated human factors issues must be considered.

2. The sheer number of interactions to be specified increases exponentially with the number of features in the system. Exponential complexity growth is insidious because it implies that as features are successfully added, the difficulty in adding more features increases.

The complexity introduced by feature interactions in the specification phase is inherited by the design, implementation, and testing phases, since all of the specified interactions must be accounted for. Additionally, the design and implementation phases may add feature interactions not present in the specification since the sharing of components and software by features creates another channel for interaction. While testing does not add new interactions, in current practice testing compensates for incomplete specifications, design, and implementation by searching for interactions overlooked at each phase.

The exponential growth in complexity introduced by feature interactions may account for the observation that sophisticated tools and techniques haven’t produced major telecommunication software productivity increases[26]. Actually, productivity may be increasing, but the problem itself keeps getting harder, so the increase is masked. This hypothesis is consistent with the anecdotal report of telecommunication feature developers[19].
3 FEATURE INTERACTION IN THE SPECIFICATION ENVIRONMENT

Introduction of new features into switching systems begins with specification of what the features do. Since resolution of interactions discovered at this time will multiply the specification size, it may be tempting to postpone resolution. But, since the later an error is found, the more expensive it is to correct[6], we believe interactions should be resolved during specifications. Also, interactions are easier to discover in an abstract specification than in an implementation, containing many details irrelevant to the abstraction. And if multiple vendors develop features independently, the only way to find interactions before testing is to use the specifications.

To find the feature interactions, the specifier must examine the specifications of all of the existing features. The specifications must meet certain standards.

- Specifications must be clear and understandable.
- Specifications must be complete and consistent.
- The specification must agree with the customer's concept of the system.
- Specifications begin large and grow larger with every change.
- The relationship of specifications to design and implementation must be clear.

Current specifications rarely meet these standards. In fact, specifications are often an afterthought, produced after the system has been implemented.

Current research into formal specification languages for switching systems, prototypes for consistency-checking tools, and prototype tools for generating specifications from informal descriptions is unproven. Several problems remain to be addressed, such as testing specifications efficiently for interesting properties and managing very large specifications. We describe below the current state of research in this area, and some problems that still need to be addressed.

Specifications written in natural language are easy to understand but hard to test for completeness or consistency. Since many interactions show up as inconsistencies, this interferes with discovery of interactions. The ambiguity of natural language specifications can also impede discovery of divergence from customer's wishes.

The alternative is to use an artificial specification language — formal or informal. Formal specification languages can theoretically be tested for completeness and consistency, and for interactions as well. Also, some artificial languages are executable or automatically transformable to an executable form. Such languages can be used for rapid prototyping, to discover interactions that cannot be found by examining the written specification. Furthermore, artificial languages tend to be more concise than natural languages.
Three artificial specifications languages developed specifically for real-time systems are RSL[3, 1], SDL[32, 11], and PAISLey[39, 40]. In addition, concurrent logic programming languages are promising alternatives.

RSL was motivated by the need for automated consistency checking of the voluminous requirements specifications for the Ballistic Missile Defense system. The automated tools provided for RSL include interactive facilities for displaying flow graphs and static checking for several kinds of errors. RSL specifications (as well as specifications in other languages) can also be processed by the Requirements Language Processor (RLP)[13, 14] developed at GTE laboratories, where it has been used to detect errors in a small switching system specification. It tests for incorrect static structure of a specification, redundancies, and inconsistent or undefined state transitions. It would detect some kinds of interactions, but it does not test for dynamic problems such as deadlock or livelock.

SDL – the CCITT Specification and Description Language – is widely used for specifying telecommunications systems. Many tools have been developed for use with SDL, including interfaces to display SDL graphics[17, 22], an editor for defining SDL graphics[5], automatic translators of SDL code into executable code[2, 18], and consistency checking tools[37].

PAISLey (Process-oriented, Applicative, and Interpretive Specification Language) is a formal language that follows the operational approach to software development, which places emphasis on constructing an operating model of the system functioning in its environment. Since the specifications are executable, rapid prototyping is immediate in PAISLey.

Logic programming, a paradigm proposed by Kowalski[24] based upon Robinson’s resolution principle[31], also appears suitable for specifying features. To describe the behavior of features, a language should express parallel programming concepts such as processes, communications, and synchronization. Concurrent logic programming languages, a class of extended logic programming languages including PARLOG[8], Concurrent Prolog[33], and GHC[38], have been proposed as general parallel programming languages to provide such expressive powers. The potential use of concurrent logic programming languages in the specification and implementation of telecommunication switching systems has been demonstrated by Elshiewy[16] using PARLOG-RT, a variant of PARLOG. Similar to PAISLey, rapid prototyping is also easy with PARLOG-RT.

Formal specifications require so much precision that they may be difficult to produce. The SAFE system[30], (Specification Acquisition From Experts) addresses this problem by synthesizing a formal specification from an informal description. A related system for telecommunications services is the SAL system, developed at NTT ECL in Japan, which automates the generation of specifications in SDL from individual service specifications. SAL automatically combines individual service descriptions, detects inconsistencies between the individual services, and produces a complete set of SDL process descriptions.

Automatic verification that specifications are consistent is a promising ap-
proach. But detecting certain kinds of inconsistency may be intractable[28] or undecidable[34]. There are interactions whose detection appears to require examination of all possible executions of the system. We call these “dynamic interactions”. To make this idea concrete, suppose that customer A has automatic “retry on busy”, which continues calling a busy line until it is free, and customer B has automatic “return call if busy”, which remembers a call that arrives when the line is busy and returns it as soon as the line is free. If A calls B, an infinite cycle of calls could be initiated, in which B tries to return A’s call but A is retrying B, who remains busy trying to call A.

Some dynamic interactions can be identified in a simulation or prototype of the system. But this approach does nothing to reduce the magnitude of the problem. A system with a large number of features will not be verifiable by exhaustive methods.

Finally, existing code and specifications (or lack of them) must be managed. In the worst case, specifications are never written out, but exist only in the code. A developer trying to modify the system will find it difficult to extract the specifications from the code. But tools are emerging[21] to help extract the specification from the code and record the extracted specifications. Another problem is locating relevant information in complete (and therefore very long) specifications. Database technology, which requires structured information, is currently used for this[17]; hypertext technology[10], which allows arbitrary links between unstructured documents, may also prove useful.

4 FEATURE INTERACTION IN THE DESIGN ENVIRONMENT

The first problem in the design stage is that incremental introduction of new features tends to subvert the original decomposition of the system. The second problem is that new interactions may be introduced by resource sharing and other design decisions.

Software engineers have long recognized the importance of obtaining a good decomposition. A crucial idea in choosing a decomposition is information hiding[29]. Information hiding refers to the choice of information that one module presents to or hides from another module. This criteria has been used widely in existing switching systems, such as the AXE system[27], 5ESS[12], DMS-100[15], and others. But there is no canonical choice for what information to hide, and a good choice for one system may be a bad choice for another. For example, some operating systems simplify access to files by hiding details of how the files are stored. This practice inhibits the implementation of good database managers, which need to know these details[35] The decision to hide details of how files are stored is a good technique if none of the applications manipulate massive quantities of data, and bad otherwise. Furthermore, what starts as a
good decision may become a bad decision as the system evolves.

So information hiding applies best when the changes to a system can be anticipated. Massive restructuring of a deployed system to add a new feature that was not anticipated in a way that realizes good functional design is much more difficult.

Achieving and maintaining a good decomposition may be assisted by tools that automate the conversion of specification into design. These tools follow software engineering rules to achieve good functional decomposition. An automated software design assistant[23] uses quantitative measures to determine the degree to which a decomposition of a system into modules satisfies the criteria of structured software design.

A second approach to this would apply program transformations to restructure the system. The program transformation system developed by the Munich CIP project (Computer-aided Intuition-guided Programming) and the transformational activities connected with the SETL project both address methods of source-to-source program transformation. The goal of the CIP project is to verifiably transform specifications to programs, but this work can be used in system modification also. One way to use it would be to re-specify and re-implement the system. Since the implementation is largely automatic, much work can be saved. A second way is to apply transformation rules directly to existing source code. Contributions of the SETL project include work on program optimization by transformation and a verification/manipulation system that allows programs to be changed and annotated with assumptions or augmented with new assumptions, if necessary to verify the program.

Program transformation systems can also be used to automate conversion from specification to design. CIP, the SAFE system, PSI, and PAISley all include mechanisms for this purpose. The HOPE language[7] can be used both to formulate the programs to be transformed and also to write the metaprograms that transform them. The DEDALUS system[25] was created to derive LISP programs automatically from high-level input-output specifications.

Even if the specification environment allows efficient handling of specification level interactions and the design environment provides for restructuring the system, new interactions arise during design because of the need to share certain components such as trunks, processors, and service circuits. Component sharing creates several interaction problems. First, and simplest, multiple usage of a resource class complicates calculation of the number of resource instances required. Next, resource combinations may result in situations violating system specifications. For example, it may appear perfectly reasonable for inter-office call forwarding to use a normal outgoing trunk. However, if the incoming call is interoffice, the resulting trunk to trunk connection may violate the transmission requirements of tandem connections. Redefinition of the conditions under which resources are used may require changes to other systems components. For example the fault tolerance system depends on this knowledge to detect erroneous conditions. Finally, deadlocks may arise from dynamic resource competition.
A final problem in the design environment is the complexity that is inherited from the specification environment, and increased by the design itself. This again emphasizes the need for an architectural solution to the feature interaction problem to prevent exponential growth.

5 FEATURE INTERACTION IN THE IMPLEMENTATION ENVIRONMENT

Feature interactions increase the implementation effort because they increase the amount of software that must be written, and lead to problems in software reuse.

The increased amount of software is not a serious problem, since the conversion of a complete design, based on a complete specification is not difficult and can be accomplished by relatively un-experienced programs. Implementation is further aided by tools such as automated code generators which reduce the human effort required. Obviously, if the specifications and design are incomplete, then during implementation specification and design decisions must be made, so it can only be preformed by highly skilled programmers.

A more serious implementation level problem results from software reuse. Given a system which contains many software functions, it is likely that the feature being added requires new functions that are nearly identical to existing functions and that can be obtained by applying seemingly minor, localized changes to the existing functions. However, these changes occasionally ripple through the system and cause other functions to fail catastrophically. Tools for automated detection of ripple effects[9] are emerging and will become an important part of the implementation environment.

In theory, systems evolution is iterative, and proposed changes to external behavior are formally specified and analyzed in the specification and design environments before being implemented. In practice, formal specification and design is often skipped. Implementors make changes to the software with no regard to whether the underlying design is properly decomposed or whether the total specification remains consistent. While the practice of making changes to the implementation without reference to the specification is dangerous, ripple effect analysis, coupled with hyper-text linkages between software, design, and specifications, may provide tools for analyzing the impact of implementation level changes on the design and specification.
6 FEATURE INTERACTIONS IN THE TESTING ENVIRONMENT

The feature interaction problem leads to lengthy testing intervals because all of the interactions introduced during specification, design, and implementation must be tested. Exhaustive testing is impractical; experienced testers use heuristics to limit the number of cases actually tested, but occasionally important cases are omitted. Theoretical work on probabilistic selection of test cases[36] is valuable because it reduces dependence on experienced testers. Tools that automatically produce test cases from specifications are valuable since they rigorously enumerate all possible tests, however, in general the problem is not in coming up with enough tests, it is in limiting the number of tests executed.

In current practice the completeness of specifications, design, and implementation is not ensured, and the feature interaction problem leads to additional testing effort since testing is used to compensate for incompleteness in the other phases. Tests are devised to search out overlooked interactions. These tests normally require a highly experienced tester who can use insight to suggest likely problem areas for testing. Testing effort would benefit from techniques improving the completeness of all other phases.

7 CONCLUSIONS

The primary difficulties presented by the feature interaction problem appear to be the size of the problem, because the interactions grow exponentially, and dynamic interactions, which are not well understood.

The problem raised by dynamic interactions especially requires more study. These interactions have not been addressed by current research. Will this class of interactions become more significant as more features are added to switching systems? Can specifications be considered complete if dynamic interactions are not addressed? Are there methods that can be used to detect and resolve them?

The feature interaction growth rate is addressed by a number of tools and techniques, some of which have been described here. But these approaches give a developer more leverage to apply to the problem without really solving it. We contend that the size and growth rate of the feature interaction problem requires more than tools that work faster than people. Instead, we need to restructure the problem to eliminate the need to consider every interaction. We cannot say exactly what approach will work, or even is most promising, but we can suggest approaches that have proved useful for other kinds of systems.

The success of operating and database systems is achieved by presenting an interface to the user that hides interactions between their applications. In the case of an operating system, the interactions occur when two applications attempt to access the same physical resource. In the case of a database system, the interactions occur when two transactions attempt to access the same data.
This suggests that we must first determine what conflicts cause the interactions in a switching system and then define an interface that enables us to hide those interactions from a feature designer.

Another angle of attack on the problem requires restricting the assumptions that a feature designer can make about the behavior of other features. That this is necessary follows almost immediately from the nature of the feature interaction problem: new features are being added to a telecommunications system, more will be added in the future, but their nature is not known. So any assumption that a feature designer makes restricts the possible behaviors of other features. On the other hand, it is always necessary to make some assumptions to accomplish anything. So the goal is to find a minimal set of assumptions, one that does not restrict the potential functionality of features, but that does eliminate the potential for negative interactions.

A third possibility is to alter the external model of the switching system’s behavior. The external model is a black box to the user. If any decisions are made about resource allocation, the switching system must make them in a reasonable way. Feature programming might be easier if the user and network could communicate with each other to make control decisions, as opposed to the current approach of embedding control decisions in feature software.

8 ACKNOWLEDGEMENTS

We are grateful to our many colleagues at Bellcore who have spent time and effort helping us to understand this problem better. Rich Asmuth, Lisa Blitzer, Milo Bloom, Pete Brady, Howard Bussey, Wendy Chow, Neil Haller, Gary Hayward, Tom Koh, Lee Moffitt, Margaret Nilson, T. A. Pham, and Brian Whittle have been especially helpful.

References


