Foundations for Features

Glenn Bruns
Bell Labs, Lucent Technologies

Abstract

The feature interaction problem arose in the field of telecommunications but is now recognized as a general problem of software engineering. Capturing the idea of feature interaction in a simple and precise way has proven difficult. This paper briefly surveys existing notions of feature and feature interaction.

1 Introduction

The problem of feature interaction is proving to be a broad and basic problem of software engineering – not just a curiosity of telecommunication systems. The word "feature" once meant a call processing feature in a telecommunications system, but now is often used to mean a unit of change in system development. Despite its increasing relevance, the field of feature interaction research seems to lack a clear set of core technical problems. Indeed, there seems to be little consensus on some of the most basic questions one could ask about features and feature interaction:

- what is a feature?
- how are features composed?
- what is a feature interaction?
- are feature interactions peculiar to telecommunication systems, distributed systems, reactive systems, ...?
- is feature interaction a property of specifications, or implementations, or a relationship between specifications and implementations?

The feature interaction problem is fascinating because it is real and easy to explain, yet has been hard to pin down in a satisfactory way. Many papers that propose solutions to the feature interaction problem do not contain a clear statement of the problem. Others define the notion but link it to details of a software development approach. If a simple, shared conception of the problem existed, the field would be more focused, more accessible, and more likely to attract the attention of others.

The modest goal of this paper is not to provide definitive notions of feature and feature interaction, but only to survey some existing conceptions of feature interaction. We hope that by focusing on basic concepts we can avoid the problem of defining feature interaction in terms of the methods for dealing with them. (See [16] for a survey of reasoning methods for feature interactions.)

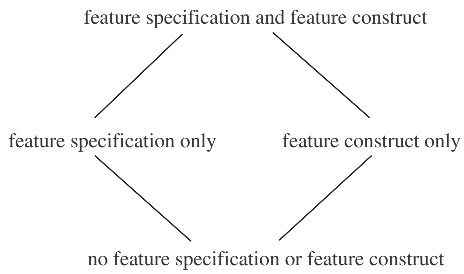

Figure 1: Assumptions that can be used when defining feature interactions.

2 Features and Feature Interaction

To define "feature" and "feature interaction", we need some basic ingredients. For example, it is common to assume the existence of a "base system" on which features are developed. Two other assumptions are also frequently made:

- features are formally specified
- a feature construct exists in the design or programming language

The term *feature construct* is understood here to mean a syntactic or semantic entity that represents a single feature at the program or design language level.

These two assumptions play key roles in many ideas of feature and feature interaction. For example, one notion of feature interaction is as the logical inconsistency of feature specifications. Another notion treats feature interaction as a confluence property when applying feature constructs to a base system.

The four parts of this section reflect the cases one gets by looking at whether feature specifications or feature constructs are present (see Figure 1). The subsections are ordered so that ideas using fewer assumptions come first. In each section we first provide a framework for features and feature composition, and then list some ideas on the nature of feature interaction. In most cases pointers to the literature are provided.

2.1 No feature specification or feature construct

In this case, programs implement features, but there is no feature construct in the programming language. Therefore we assume only that features have names, and that these names will appear in programs that implement them. We can formalize this as follows. Let P and Q stand for programs, and assume that a program can be interpreted as a *labelled transition system* (LTS), which consists of a set of states (including an initial state) and a transition relation on states. This is a very general model of program behavior. We write $[\![P]\!]$ for the LTS obtained as the interpretation of program P.

To capture that a program implements a feature, assume that transitions of the LTS can be labelled with feature names. We write $s \xrightarrow{f} s'$ if there is a transition from a state s to a state s' labelled with feature name f. Such a transition represents an action performed to support feature f. We write $\mathcal{L}(P)$ for the feature names found anywhere in $[\![P]\!]$. A feature named f is regarded as being implemented in P if $f \in \mathcal{L}(P)$. In this way we can say that a program implements a feature without having a feature construct.

We also need to be able to describe behavioral properties of programs. Let ϕ and ψ stand for formulas of some behavioral specification language. This means that ϕ and ψ can be interpreted as sets of LTSs. Assume that the language is closed under conjunction (written here as $\phi \wedge \psi$). We write $P \models \phi$ if program P satisfies the behavioral property ϕ.

Now some concepts of feature interaction can be presented. One concept is that two features interact if one of the features can disable the other. The property disables(f, g) holds of an LTS if either: 1) the LTS has reachable states s, s', s'' such that $s \xrightarrow{f} s'$ and $s \xrightarrow{g} s''$ but there is no state t such that $s'' \xrightarrow{f} t$, or 2) the symmetrical case in which f and g are swapped.

Idea 1 *Suppose P is a program with feature names f, g both in $\mathcal{L}(P)$. Then f and g interact in P if $P \models$ disables(f, g).*

This idea appears in [7], where it is called "missed trigger interaction".

One feature can also *enable* another. We write enables(f, g) if at some reachable state one of f, g but not the other is enabled, and by following a transition of the enabled feature one can immediately reach a state in which the other feature is enabled.

Idea 2 *Suppose P is a program with feature names f, g both in $\mathcal{L}(P)$. Then f and g interact in P if $P \models$ enables(f, g).*

This idea also appears in [7], where it is called "sequential action interaction".

Generally, there may be many properties of an LTS that we want to avoid, such as deadlock, livelock, and non-termination. The ideas above are easily generalized to any behavioral property.

Idea 3 *Suppose P is a program with feature names f, g both in $\mathcal{L}(P)$, and ϕ is a behavioral property. Then f and g ϕ-interact in P if $P \models \phi$.*

For example, in [22, 7], feature interaction is taken to be non-determinism on input events. In [7] this form of interaction is called "shared trigger interaction". A problem with the idea as written here is that ϕ may hold of P for no reason having to do with f and g. The condition could be strengthened in various ways; for example, by requiring that ϕ involve transitions of either f, or g, or both.

The idea above can be used not only to define feature interaction, but also a notion of a single feature being "bad": a feature f is bad in P if P satisfies some undesirable property.

2.2 Feature specification but no feature construct

Here again we have no feature construct, but now each feature has a formal specification. If ϕ is the specification of a feature, then the idea is that a program P implements the feature if $P \models \phi$. Since there is no feature construct, we let a feature's specification stand for the feature itself. So rather than saying "the feature specified by ϕ and the feature specified by ψ interact if ...", we instead say "the features ϕ and ψ interact if ...".

Some concepts of interaction take place purely at the specification level. One idea is that features interact if their specifications are logically inconsistent.

Idea 4 *Features ϕ_1 and ϕ_2 interact if $\phi_1 \wedge \phi_2$ is inconsistent.*

This idea can be found in [4, 13]. In [13] it is called a "Type I" interaction. The intuition is that if a property is inconsistent, then it is impossible to implement a system having the property. For reactive systems, it is perhaps more appropriate to use *realizability* [20], which is a stronger notion than consistency, reflecting that a reactive system can enforce a property regardless of the inputs it receives. Thus, a variant of the above is:

Idea 5 *Features ϕ_1 and ϕ_2 interact if $\phi_1 \wedge \phi_2$ is unrealizeable.*

This idea appears in [10]. Work on the use of Alternating-time Temporal Logic (ATL) to specify features [8] is related to realizability in that ATL allows one to describe properties that can be satisfied by a system regardless of the environment.

A variant of these ideas is to say that a feature interaction occurs if the conjunction of two feature specifications implies some undesirable property, such as non-determinism.

Idea 6 *Let ψ be a property. Then features ϕ_1 and ϕ_2 ψ-interact if $\phi_1 \wedge \phi_2$ implies ψ.*

The idea of features being inconsistent can be weakened by making it relative to a base system. We write $P \leq P'$ if P' is a *refinement* of P, where we assume only that $\leq$ is a preorder. Thus, to implement a feature ϕ on a base program P, one would find a program P' such that $P \leq P'$ and $P' \models \phi$. Then one can say that two features interact with respect to a program if the program cannot be refined so that both features are satisfied.

Idea 7 *Features ϕ_1 and ϕ_2* interact *in P if there is no P' such that $P \leq P'$ and $P' \models \phi_1 \wedge \phi_2$.*

Logical inconsistency is an obvious notion of conflict between features. Other notions of conflict can be subtle. For example, in reactive systems one can consider conflict on input and output events (see [15, 13]). Suppose two features seek to output different messages to the same output port – should this circumstance constitute conflict? The answer would seem to depend upon the "semantics" of the output events. For example, conflict may be present if the output events cause a binary setting to toggle, but absent if the events cause a count to be incremented.

Another subtle issue arises in saying what it means for a program to implement two features specified by ϕ and ψ. If a program P implements a feature specified by ϕ whenever $P \models \phi$, then it seems right to say that P implements both ϕ and ψ just when $P \models \phi \wedge \psi$. But there are other valid approaches. For example, if features are implemented sequentially to a base program, we may expect, or even require, that one feature has priority over another. Thus we may expect logically not that ϕ and ψ both hold, but that they both hold only in the absence of conflict, and that in the presence of conflict ϕ but not ψ holds. This is roughly the idea behind the use of "theory update" in composing formulas that represent feature specifications [14].

As in the previous section, some of the ideas here could be applied to single features. For example, we can ask of a single feature specification whether it is inconsistent, unrealizeable, or implies a undesirable property ψ. In [3], POTS (Plain Old Telephone Service) is specified, and then the specification is checked to see that deadlock is not possible.

2.3 Feature construct but no feature specification

Now we consider that features are captured in a feature construct, but have no formal specifications. For example, a feature-based programming notation might be used, in which features are implemented directly from informal requirements. There are many notions of feature construct in the literature; some examples are [5, 6, 13, 11, 19].

We write f for a feature, and $P + f$ for the composition of program P and feature f. The idea is that a program implements freature f if it has the form $P + f_1 + \cdots + f_n$, where f is identical to some f_i. We write $P \equiv Q$ to mean that programs P and Q are behaviorally equivalent. For example, this could mean that LTSs $[\![P]\!]$ and $[\![Q]\!]$ are observation equivalent [18].

A variation on the idea of "interaction as conflict" is to regard two features as interacting if their composition on a base system is not well-formed. In this view feature composition is a partial operation.

Idea 8 *Feature f and g* interact *in P if $(P + f) + g$ is not well-formed.*

This idea appears in [15], where a form of synchronous concurrent composition is used to apply features to a base system.

Another basic idea is that two features interact for a program if the order of application of the features matters.

Idea 9 *Features f and g* interact *in P if $(P + f) + g \not\equiv (P + g) + f$.*

This idea appears in [17, 6].

A variant is to use a notion of program equivalence based on some logical language: two programs are equivalent if they satisfy the same formulas. We write L here for a logical language. For simplicity we assume that L has negation (with the expected property that, for all programs, $P \models \phi$ iff $P \not\models \neg\phi$).

Idea 10 *Features f and g* interact *in P if there exists a ϕ in L such that $(P + f) + g \models \phi$ but $(P + g) + f \not\models \phi$.*

This idea appears in [19].

A general notion of feature interaction is that applying both features to a program will lead to a program with some undesirable property.

Idea 11 *Let ψ be a property. Then features f and g* interact *in P if $(P+f)+g \models \psi$ or $(P+g)+f \models \psi$.*

This idea appears in [5], where feature interaction is defined as a kind of non-determinism.

To take the algebraic approach further, one could define a feature construct and a binary composition operator $+$ on features such that every program can be written as a composition of features. We expect $+$ to be associative but not necessarily commutative, so that features plus composition forms a semigroup. Furthermore, the semantics of features on their own could be defined, and a behavioral equivalence on features (written $f \equiv g$) developed. Interaction of features could then be taken to mean simply non-commutativity.

Idea 12 *Features f and g* interact *if $f+g \not\equiv g+f$.*

In this framework the equivalence $\equiv$ should be defined so that it is a congruence: $f \equiv g$ implies $f+h \equiv g+h$ and $h+f \equiv h+g$, for all h. Algebraic approaches to feature interaction are described in [12, 21].

An obvious idea not listed above is that two particular features interact if an expected property of their specific combination does not hold. In other words, two fixed features f and g interact with respect to a base program P if $(P+f)+g$ does not satisfy some expected property ϕ (that depends on f and g). This approach is used in [1]. We mention this approach only in passing because it does not represent a generic concept of feature interaction.

2.4 Feature specification and feature construct

Finally we consider the case where a feature construct is used and features are specified. We write $f : \phi$ if ϕ is the specification of feature f, and then say that f implements ϕ if $P+f \models \phi$ for all programs P. Sometimes (as in [19]) the feature f will include a specification of the class of programs to which it can be applied.

In this setting two features can be said to interact if adding a feature causes an existing feature's specification to be violated.

Idea 13 *Let $f_1 : \phi_1$ and f_2 be features, and P be a program. Then f_1 and f_2 interact in P if $(P+f_1)+f_2 \not\models \phi_1$.*

This idea appears in [19].

A closely related idea if that two features interact if one feature's specification cannot be met in the presence of the other.

Idea 14 *Let f_1 and $f_2 : \phi_2$ be features, and P be a program. Then f_1 and f_2 interact in P if $(P+f_1)+f_2 \not\models \phi_2$.*

This also appears in [19].

Another idea focuses on the specification of the base program. Two features interact if the specification of the base program cannot be met in the presence of both features.

Idea 15 *Let f_1 and f_2 be features, and $P : \phi$ be a program. Then f_1 and f_2 interact in P if $(P+f_1)+f_2 \not\models \phi$, or $(P+f_1)+f_2 \not\models \phi$.*

3 Prospects

To get a sense of where the field of feature interaction is going, it is helpful to understand the problems the field is trying to solve. Some work is focused on specific application domains, such as the interactions of call-processing features in telecommunication systems. However, from a broad perspective, feature interaction addresses a fundamental problem that arises in a common approach to software development, in which:

- informal requirements list new features to be added to an existing system
- development teams – one for each feature – work independently to implement a feature on a private version of the base program
- implementing a feature involves modifying multiple files of the base system
- the final product is obtained by merging the code from each team; this is often performed semi-automatically with the help of a version-control system

A feature interaction can be said to occur when the new version does not support the features as expected. Other problems faced in this approach are that it can be hard to trace requirements to code, and hard to understand the effect of semi-automatic program merging.

Some progress on solving these problems may be possible by providing support for features in a general-purpose programming language. In a "features as modules" approach, each team would develop their feature as a self-contained piece of code with well-defined interfaces, and then the collection of modules would be applied to the base system, perhaps by explicitly ordering the features with potential interactions in mind. A module would be "higher-order", in the sense that it would describe modifications to the behavior of the existing code. In aspect-oriented programming, *advice* is much like this concept of feature module. Examples of work on feature interaction and aspects are [9, 2].

However, a "features as modules" approach may introduce new problems. For example, would systems described as a large collection of features be comprehensible? Would it be practical to define a single total order on all program features? How could one form packages of features with well-defined interfaces? Finally, how can introducing a feature construct by itself help in finding feature interactions? Perhaps ideas from programming languages can help solve problems of feature structure and scope, while ideas from the feature interaction world can help solve problems of "advice interaction".

Acknowledgements

Michael Benedikt, Alan Jeffrey, and Nils Klarlund provided helpful comments on a draft version of this paper. Research supported in part by a grant from the National Science Foundation: CyberTrust 0430175.

References

[1] R. Accorsi, C. Areces, W. Bouma, and M. de Rijke. Features as constraints. In *Proceedings of FIW'00*, pages 210–225. IOS Press, 2000.

[2] Lynne Blair and Jianxiong Pang. Aspect-oriented solutions to feature interaction concerns using AspectJ. In *Feature Interactions in Telecommunications Systems VII*, pages 87–104. IOS Press, 2003.

[3] J. Blom, B. Jonsson, and L. Kempe. Using temporal logic for modular specification of telephone services. In L. G. Bouma and Hugo Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, pages 197–216, Amsterdam, The Netherlands, May 1994. IOS Press.

[4] Johan Blom, Bengt Jonsson, and Lars Kempe. Using temporal logic for modular specification of telephone services. In W. Bouma and H. Velthuijsen, editors, *Feature Interactions in Telecommunications Systems*, 1994.

[5] Jan Bredereke. Automata-theoretic criteria for feature interactions in telecommunications systems. Technical Report 273/95, Dept. of Comp. Sci., Univ of Kaiserslautern, December 1995.

[6] G. Bruns, P. Mataga, and I. Sutherland. Features as service transformers. In *Feature Interactions in Telecommunications Systems V*. IOS Press, 1999.

[7] M. Calder, M. Kolberg, E. Magill, D. Marples, and S. Reiff-Marganiec. Hybrid solutions to the feature interaction problem. In *Feature Interactions in Telecommunications Systems VII*, pages 187–205. IOS Press, 2003.

[8] F. Cassez, M. D. Ryan, and P.-Y. Schobbens. Proving feature non-interaction with alternating-time temporal logic. In *Language Constructs for Describing Features*, pages 85–104. Springer Verlag, 2001.

[9] Rémi Douence and Pascal Fradet and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*. Springer-Verlag, Lecture Notes in Computer Science 2487.

[10] Amy P. Felty and Kedar S. Namjoshi. Feature specification and automatic conflict detection. *ACM Transactions on Software Engineering and Methodology*, 12(1):3–27, January 2003.

[11] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Proceedings of FSE '01*, 2001.

[12] Christophe Gaston, Marc Aiguier, and Pascale Le Gall. Algebraic treatment of feature-oriented systems. In *Language Constructs for Describing Features*. Springer Verlag, 2001.

[13] Robert J. Hall. Feature combination and interaction detection vis foreground/background models. *Computer Networks*, 32(4):449–469, 2000.

[14] H. Harris and M. D. Ryan. Theoretical foundations of updating systems. In *Proceedings of FSE 2003*. IEEE Computer Society Press, 2003.

[15] Jonathan D. Hay and Joanne M. Atlee. Composing features and resolving interactions. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 110–119. ACM Press, 2000.

[16] Dirk O. Keck and Paul J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Trans. Softw. Eng.*, 24(10):779–796, 1998.

[17] Cornel Klein, Christian Prehofer, and Bernhard Rumpe. Feature specification and refinement with state transition diagrams. In *Feature Interactions in Telecommunications Systems IV*. IOS Press, 1997.

[18] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[19] Malte Plath and Mark Ryan. Feature integration using a feature construct. *Sci. Comput. Program.*, 41(1):53–84, 2001.

[20] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 179–190. ACM Press, 1989.

[21] Christian Prehofer. From inheritance to feature interaction or composing monads. Technical Report TUM-I9715, Technische Univerität München, 1997.

[22] B Stepien and L Logrippo. Feature interaction detection by using backward reasoning with LOTOS. In S.T. Vuong and S.T. Chanson, editors, *Protocol Specification, Testing and Verification XIV*, pages 71–86. Chapman & Hall, 1995.