A FEATURE MANAGER APPROACH TO THE ANALYSIS OF COMPONENT-INTERACTIONS

Lynne Blair, Trevor Jones

Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K.
{lb@comp.lancs.ac.uk, jonest@lancaster.ac.uk}

Stephan Reiff-Marganiec

Department of Computing Science, University of Stirling, Stirling, FK9 4LA, Scotland.
{srm@cs.stir.ac.uk}

Abstract: Feature interactions have been the subject of extensive research over recent years within the telecommunications industry. We consider one approach developed within this domain, a feature manager approach, which we claim, has wider-applicability. We analyse this approach in a component-based context, showing how the technique can be used to manage interactions occurring between components. Through a component-based example, we introduce the various steps of the technique. In addressing performance issues, clearly significant for a run-time technique, we conclude that the results are promising and, with the benefit of an on-the-fly method, are scalable.

Keywords: Components, component-interactions, features, feature-manager.

1. Introduction

Distributed systems are currently undergoing a period of fundamental change caused by the emergence of multimedia, mobile and ubiquitous computing and other key areas such as internet and GRID-based technologies. To help manage complexity inherent in such systems, effective methods of structuring systems need to be available. *Object-oriented* development methodologies are one such mechanism, but suffer from a number of significant problems with respect to evolution and adaptation, such as the well-known fragile base class problem (Szyperski, 1988).

More recently, *component-based* architectures have emerged as an attractive structuring mechanism with three key advantages over object-oriented methods. Firstly, they rely on composition, rather than inheritance, and thus side-step the fragile base-class problem. Secondly, they are often language independent and typically coarse-grained, thus encouraging re-use. Finally, they are denoted by explicit (external) contracts including required/provided interfaces, therefore encouraging third-party development.

Consequently, component-based methodologies focus on the provision of methods to specify individual components together with their composition (i.e. an architecture). By allowing new components to be added, and existing software to be packaged as components, an incremental development model for evolutionary and dynamic architectures can be obtained.

Previously there has been considerable success in developing formal methods to support object-oriented distributed systems, for example through the FMOODS community. In terms of providing a formal representation of component architectures, architectural description languages (ADLs), such as Rapide, Piccola and Wright, have been investigated (Allen and Garlan, 1997) (Luckham and Vera, 1995). Typically, architectures are verified in terms of type compatibility between (required and provided) interfaces. In addition, checks may be carried out on the validity of architectures against certain style rules (Medvidovic and Taylor, 2000) (Shaw and Clements, 1996).

Such techniques are likely to remove many intra- and inter-component problems at design-time/ compile-time, but do not cater for the more subtle problems associated with unwanted interactions between components at run-time. Importantly, two key questions are not addressed:

- When we compose an architecture, how can we be confident that components work well together at run-time and that there are no unwanted or subtle interactions?
- Also, when we adapt an architecture (at run-time), how can we be confident that replacements or updates behave as expected, especially in tandem with other components?

This paper aims to address these questions, and proposes a run-time technique originally developed for detecting and resolving interactions between features of telecommunications systems. So, whilst the technique itself is not new, the study of its wider-applicability to component-based systems is entirely new. The example we present is also novel, and shows how the feature manager approach can be used to manage interactions occurring in a component-based system.

2. Features and Feature Interactions

Telecommunications features, and feature-interactions, have been extensively researched over the last 10 years. A telephony *feature* refers to a unit of additional functionality, added to an existing system and usually perceived as having a self-contained functional role. Examples of telephony features include call forwarding on busy, call waiting, voice mail, etc.

Problems arise when several features do not work correctly, or consistently, in the context of each other. This is termed the *feature interaction problem*, i.e. “the behaviour of one feature is affected by the

behaviour of another feature or another instance of the same feature” (Kimbler and Velthuisen, 1995).

There are numerous well-known cases of (telephony) feature interactions, e.g. see (Dini et al., 1997), (Calder and Magill, 2000) and (Kimbler and Bouma, 1998). However, the interaction problem is not restricted to the telecommunications domain. More wide-ranging interaction problems have also been identified in (Hall, 2000) (in email systems), in (FIREworks) (in a variety of miscellaneous examples) and in (Blair and Pang, 2000) and (Blair et al., 2001) (in multimedia, mobile and internet services, and component-based middleware respectively).

Many different techniques have been proposed for the detection, and subsequent resolution, of feature interactions. Essentially *off-line* and *on-line* techniques are distinguished, the former are understood to apply at design time, the latter at run-time. In most off-line techniques, the base system and features are specified in a formal language whilst the properties that the combined system should exhibit are (typically) specified through temporal logic. The level of success depends on the accuracy and level of abstraction of the specified properties since an inaccurate (or not precise enough) property specification will lead to interactions remaining undetected (Du Bousquet, 1999). Furthermore, off-line techniques rely on a-priori knowledge of the behaviour of the individual features (i.e. design time information).

In contrast, adaptive on-line techniques address this latter issue. Such approaches have been developed from a more pragmatic perspective and have evolved over time to become increasingly (dynamically) adaptive. Commonly, a *feature manager* performs run-time detection and resolution of interactions, whilst its adaptation strategy is typically powered by a knowledge database, such as predefined interaction tables, state transition rules, abstract data types or user agent rules.

3. A Feature Manager Approach to Detecting and Resolving Interactions

3.1 A Description of the Approach

The technique presented in this paper builds on the (transaction-based) feature manager approach of (Marples and Magill, 1998) for run-time detection, enhanced with the automatic (rule-based) resolution technique of (Calder et al., 2000) and (Reiff-Marganiec, 2000). The feature manager’s primary role is to mediate between the features and the base system. Specifically, within the telecommunications domain, a feature manager intercepts incoming events from the underlying network and passes them to the base system and all additional features. The base system and all features

are embedded within their own individual cocoon (Marples and Magill, 1998) that offers a consistent interface to the feature manager. Note that the features and the base system are treated the same throughout this technique, and hence need not be distinguished. If an event triggers a response in the features, the response(s) are sent back to the feature manager and collected. If the feature manager receives more than one response to a particular event, we have the potential for interaction between features. In such an instance, the feature manager takes on the task of exploring the *space* of possible resolutions, using the list of (2 or more) responses, as follows:

- Initially, a *behaviour tree* is built representing all possible *solutions* to the interaction problem. In constructing the tree, the feature manager first stores the current state of the system and initiates copies of all the features. The first response is then fed back to the copies of all features, the responses are collected and are then processed. Part of this processing may involve sending and receiving further events. All possible message orderings are considered and all subsets of the set of features are considered. On termination of the process we have a tree containing all possible solutions.
- *Pruning* of the tree then takes place by using pre-defined rules to remove ‘bad’ or ‘incorrect’ solutions. Note that formulating these rules will, in general, require domain specific knowledge.
- Finally, *extraction* of the best resolution occurs, again using pre-defined rules. Whilst the formulation of extraction rules may be helped by domain specific knowledge, such knowledge is not necessary. For example, extraction rules may ensure that the largest number of features is satisfied, or may set priorities based on some architectural design.

Note that the three phases (construct, prune and extract) are all applied at run-time and that a tree is always constructed when a trigger event occurs. One might assume that the solutions found could be stored and reused. However, the system is assumed to be reconfigurable at run-time and we also cannot assume that a component always reacts in the same way (there might be internal decisions taking place) and hence we only know what behaviour is requested when we probe the components. A more detailed description of these processes can be found in (Calder et al., 1999) and (Reiff-Marganiec, 2001).

3.2 Applying the Approach to Component-Based Systems

An advantage of both the detection and resolution methods above is that they do not require knowledge of the internal behaviour of features, but are based on observable behaviour, i.e. events sent by the features. This enables

the technique to be used, for example, in closed, proprietary or legacy systems, where internal knowledge may be unavailable.

Significantly, this reliance only on externally observable behaviour makes both techniques transferable to other domains, such as component-based systems, where source code or other internal knowledge may not be available. Reasons for this include:

- Components are often language independent and may be shipped in binary-only form.
- They are typically denoted by explicit (external) contracts including required/ provided interfaces, therefore encouraging third-party development (and hence the potential withholding of internal information for commercial or competitive reasons).

In mapping this approach to component-based systems, the events in the techniques described above can be mapped directly to events generated on the interfaces of components. Furthermore, the feature (or component) manager can be interpreted as a (management) component in the component-based system. Finally, it is noticeable that the cocoons used around features have similarities with the use of containers in, for example, the Enterprise Java Beans component architecture (Roman, 1999), though we have yet to explore this similarity further. However, we do rely on the following assumptions:

- For the detection stage, we rely on the fact that we can create copies of all the associated processes, or take a snapshot of the system, so that we can revert back to an earlier state (e.g. through the Java *cloning* mechanism).
- For the resolution stage, we also assume that we are able to empirically generate rules for both the pruning and extraction stages. Inevitably, these rules will be domain-specific and will likely require the knowledge of a domain-expert for their generation. However, the detection process can assist in the finding of rules since it identifies all potential interactions.

4. A Component-Based Example

4.1 Overview

As an illustrative example, we examine a component-based system based on multiple clients trying to access multiple servers (that offer different services). Note that the primary purpose of this example is, not in the details of the example itself, but in allowing us to confirm our perceived wider-

applicability of the detection and resolution technique, and to facilitate analysis and evaluation of the approach.

In our system, we have a number of clients accessing a number of services, each held on a different server. Some services are secure and require passwords whilst others allow open access. Some services require access control lists to be checked whilst others are unrestricted. Finally, each service, due to high demand, is replicated and load balancing occurs between two servers offering identical services. Table 1 shows each service's status.

Table 1. Status of each service

Service	Security: secure/ open	Access Control: restricted/ unrestricted	Replication: functionally equivalent
1, 2	secure	restricted	1 & 2
3, 4	secure	unrestricted	3 & 4
5, 6	open	restricted	5 & 6
7, 8	open	unrestricted	7 & 8

We assume that the security, access control and load balancing concerns are implemented as individual management components (rather than being embedded in the services) thus allowing the system to be easily configured, or re-configured, as desired. So, in our system the 'feature manager' corresponds to a component manager whilst the 'features' correspond to the servers and the three managers (for security, access control and load-balancing management, denoted SM, ACM and LBM respectively). Furthermore, each 'feature' is still embedded in its own individual cocoon; this provides a consistent interface to the component manager and also handles the creation and termination of copies of the features (c.f. Java cloning above).

4.2 Components of the System

In this section, we provide automata descriptions of all components in the system. These automata consist of a start state, denoted by a double circle, other states and transitions. Transitions are labelled with events that follow a LOTOS-like notation: *event_name* followed optionally by *!output_value* or *?input_value*, with internal events denoted by *i(...)*. Note that for simplicity in the figures, we omit the input type. Also, since the originator and recipient of events are important for our analysis, we add *!* and *?* to events without data to clarify whether they are output or input events respectively. Note that input events correspond to methods with a public published interface that can be called (remotely) by other components, whereas output events correspond to the calling of a remote method.

4.2.1 The Client Component

The client component starts by issuing a *request* for a service, outputting both the name of the service required and client's own id. This event is followed by one of four possible events:

- The client may *timeout* (an internal event) and return to the initial state.
- The client may receive a *service_denied* event and return to the initial state.
- The client may receive a *service_granted* event, in which case the service either provides a response or the client times out (e.g. if the service is down).
- The service may first require a *password* to be supplied; this is then followed by a *timeout*, a *service_denied* or a *service_granted*, as above.

Figure 1. A model of the client

4.2.2 The Server Component

The server component is intentionally simple since we are interested only in the existence of responses to service requests, not in exactly how they are generated. Consequently, we model the server by assuming that when requests for a given service are received, these are added to a queue along with the id of the client. Once processed, a response to the request at the head of the queue is sent back to the appropriate client.

Figure 2. A model of a server component

4.2.3 The Security Manager

The security manager intercepts any *request_service* events and, before passing them on to the appropriate server, checks the security status of the requested service. If the status is *open* access (no security checks required) then the service is *granted* and the *request* forwarded to the server. If the status is *secure*, then a *password* must be supplied before progressing (or the security manager *times out*). If the password is *valid* then the service is *granted*, else it is *denied*.

Figure 3. A model of the security manager

4.2.4 The Access Control Manager

Like the security manager, the access control manager also intercepts *request_service* events. This time, if the service has an *unrestricted* access control policy, the service is *granted* and the *request* forwarded to the server. If access to the service is *restricted*, then the client's *id* is checked against the access control lists and, if *valid*, access is *granted*, and otherwise *denied*.

Figure 4. A model of the access control manager

4.2.5 The Load-Balancing Manager

In order to perform simple load-balancing across the replicated services, we assume that a count exists of the number of requests currently being processed per service (i.e. before a response is given). For this example, we assume that $cnt(s)$ corresponds to $size(queue, s)$, i.e. the number of requests on

the queue associated with server s (figure 2 above). We also assume that the function $alt(s)$ returns the identifier of the alternative server offering an equivalent (replicated) service, e.g. $alt(3)=4$ and $alt(4)=3$ (table 1 above).

On intercepting a service request, the load-balancing manager checks the queue size, $cnt(s)$, and compares it against a threshold value for s , $threshold(s)$, and a maximum value for s , $max(s)$. We also assume that we know the alternative server's queue size, $cnt(alt(s))$, and its threshold value, $threshold(alt(s))$. The action taken (for server s) depends on the size of queues associated with both s and $alt(s)$, see table below.

Table 2. Actions of the load balancing manager for server s

	$cnt(s) < threshold(s)$	$threshold(s) \leq cnt(s) < max(s)$	$cnt(s) \geq max(s)$
$cnt(alt(s)) < threshold(alt(s))$			request service on server $alt(s)$
$threshold(alt(s)) \leq cnt(alt(s)) < max(alt(s))$		request service on server s	
$cnt(alt(s)) \geq max(alt(s))$			

These constraints appear as guards in the automata model of the load-balancing manager below:

Figure 5. A model of the load-balancing manager

4.2.6 No Manager Operation

Finally, we permit the system to run with no managers. In this case, the request for a service should automatically be relayed through to the server.

Figure 6. A model of no manager operation

4.3 The Component Manager

One mechanism for analysing interactions would be to take the above automata and compose them to examine the overall behaviour of the system.

However, we do *not* choose this approach since we cannot assume the knowledge of such internal behaviour in a component-based world. Instead, we have provided these automata as an indication of what behaviour such components may have, and as a guide for explaining the analysis below. As will be seen, using the feature manager approach, we focus solely on how components respond to particular trigger events, i.e. what externally observable behaviour results from a given trigger.

4.3.1 An Initial Intuitive Analysis

Before explaining the tasks undertaken by the feature manager, we first present a brief intuitive analysis of what interactions we can expect to arise. Note that the potential for an interaction exists when more than one component responds, thus the approach does not need to know about all potential interactions in advance in order to detect them.

The security manager, if security criteria are met, can grant a client access to a given service. Likewise the access control manager, providing the access control criteria are met. If both managers are in place, we expect both sets of criteria to be met, not just one, before access to the service is granted.

Furthermore, the load-balancing manager effectively acts like a call-forwarding-on-busy feature: if one server is overloaded, the client is routed to a replica server (provided the second server is not itself overloaded). This leads to the possibility of security and access control checks being skipped.

Finally, we have provided behaviour for operation in the absence of other managers (no manager operation). Mixing this behaviour with one of the other managers will inevitably lead to undesirable interactions.

4.3.2 Constructing a Behaviour Tree

The first task of the component manager is to construct a *behaviour tree* (see section 3.1) and, in so doing, explore all potential interactions and all possible solutions. In our example, we start with the trigger event *request_service !s !id*, and broadcast this to all combinations of components in our system and collect any responses. The construction considers the behaviour between two stable states, where a stable state corresponds in this example to the initial state of the user. Stable states are left by the user issuing *request_service !s !id* and are entered again when the user can issue a new *request_service !s !id*.

As can be seen from the automata models above, all components except the server can respond to the initial trigger event. For example, the security manager might issue a *passwdreq!* or a *service_granted!* event. It is the output events that are of interest, since they may trigger further behaviour in other components, so these responses go back to the feature manager and are

collected. The following table shows the responses from the first stage in the generation of the behaviour tree:

Table 3. Start of the generation of the behaviour tree

trigger event	initial responses (with abbreviation of originating component)
<i>request_service !s !id</i> (C)	<i>service_granted!</i> (SM, ACM), <i>service_denied!</i> (ACM, LBM), <i>request!s</i> (LBM, NMO), <i>request!alt(s)</i> (LBM), <i>passwdreq!</i> (SM)

As can be seen, several responses are generated to the initial trigger event, immediately alerting us to a potential interaction. Importantly, the component manager computes all combinations of events to allow for different combinations of components being active in our system. In such a way, the component manager will be aware, at run-time, of all possible choices and will be able to decide on an appropriate resolution. For our example, calculating all combinations of events clearly generates a very big tree. Hence, we show only one part (figure 7 below) highlighting different combinations of possible responses from the security, access control and load balancing managers; we will use this to explain the subsequent steps in creating the behaviour tree. Note that we have also abbreviated the occurrence of three events, and all their possible inter-leavings, through the use of sets of events, e.g. $\{2,4,7\}$ corresponds to event orderings 2,4,7 or 2,7,4, or 4,2,7, etc.

Figure 7. Part of the tree generated from initial responses to the feature manager

Consider the first labelled path on this tree, corresponding to a password being requested, and the checks for access control and load-balancing both being ok. The feature manager takes each of these three responses in turn and feeds them back to all components in the system to see if they generate further responses. Suppose *passwdreq!* (event 2) is the next trigger event (see figure 8). This might lead to the client providing a password (event 14), which in turn provides input to the security manager and may subsequently trigger the response *service_granted!* or *service_denied!* from the security manager. No other components respond. Carrying on in a depth-first manner, now consider broadcasting *service_granted!* to all components (event 3). This corresponds to an input event in the client and, if the client (internally) times out we have reached a new stable state. The security manager also responds by relaying the service request to the server (*request !s !id*). In turn, this event (event 12)

results in the server responding with event *response !s !head(queue)* (event 13) again bringing the client to a new stable state. Rolling back a few stages, *service_denied!* (event 10) also causes a return to the initial state where *request_service !s !id* is the only response.

Now consider the second of the three trigger events, the access control manager's *service_granted!* event (event 4). On broadcasting this to all components, the only responses generated are a *request !s !id* event caused by the access control manager relaying the request on to the server (event 11) or the reaching of a new stable state. Subsequent steps for this, and for the third trigger event (event 7), are similar and are all depicted in the following figure.

Figure 8. Subsequent stages in generating part of the behaviour tree

This process obviously continues with the other initial responses until the complete tree is generated.

4.3.3 Pruning the Behaviour Tree

Having generated the behaviour tree, we must now provide rules to remove any impossible or inappropriate traces from the tree by the process of *pruning*. Note that *extracting* the most appropriate *resolution* is a separate step and will be addressed in the following section.

Typically we find that rules for pruning the behaviour tree are *message dependent*, that is they require knowledge about the events and the environment in which they occur. However, no internal knowledge of the components is required. For example, we may require that pruning should remove all traces where:

- a *service_granted !* event occurs before a *passwd_required !passwd*
- a *service_denied !* event occurs before a *passwd_required !passwd*

- a *request !s* or *request !alt(s)* event occurs before a *passwd_required !passwd*

As can be seen, this stage requires an understanding of the domain and thus the formulation of pruning rules requires input from a domain expert.

In addition to the above rules, we can also require that we remove traces that incorporate:

- any response from one or more of the managers {SM, ACM, LBM} *and* any response from NMO (no manager operation), i.e. responses from NMO should only be considered in the absence of all other managers.

In contrast to the above rules, this latter rule requires no knowledge of the actual events (just where they come from), but requires knowledge of the inter-operation of different components, for example, which components can (or cannot) be expected to work together with other components.

4.3.4 Extracting our Final Resolution

Having removed all inappropriate traces from our behaviour tree, we may still be left with a number of acceptable solutions. However, for the running system we must clearly commit to one solution. In order to achieve this, the extraction process makes use of message independent rules (more general than the above pruning rules). Such rules typically involve the application of priorities between different components and a notion of “satisfying” (i.e. allowing to proceed) a maximal number of components.

Note that the solution actually selected may depend on the particular trigger event received and values of parameters, both known only at run-time. For example, we can determine whether a server is password protected or not from the identifier of the requested service; this in turn dictates whether access is granted immediately or whether a password must first be supplied. Consequently, we would expect both traces to be returned by the extraction process, but only one will be applicable at run-time. If, at the end of the extraction process, we are left with a number of acceptable solutions that we cannot distinguish between qualitatively, we simply select one.

5. Analysis and Evaluation

The run-time detection and resolution technique that we have adopted for our analysis of component interactions requires no knowledge of the internal behaviour of components – it relies solely on their (externally observable) responses to certain trigger events. This has been important for our application to component-based systems due to the characteristics of components themselves. The detection of interactions and the subsequent

building of the behaviour tree can be completely automated and performed at run-time. However, the resolution process requires some initial off-line analysis and/ or additional input from a domain expert in order to provide adequate resolution rules (perhaps achieved, as we have done, by empirically gathering evidence from the analysis of examples). Once these rules exist, the resolution of interactions can be completed at run-time.

The question of performance is obviously important for run-time techniques, as a high performance overhead is not acceptable. In theory, the size of the behaviour tree is exponential in the number of components (features) and the length of their responses. However, in practice, this worst case is unlikely to occur since not all components (features) respond to all events, as has been seen in our example. More details on the algorithm's complexity can be found in chapter 7 of (Reiff-Marganiec, 2001).

One consideration that might be assumed to be beneficial is that component contracts provide enough detail as to which possible reactions could occur at any point and thus the tree could be reduced in size. However, since we are applying the technique at run-time we can perform even better: only the real response of the components is considered and not all possible choices – any non-determinism is resolved for free.

In order to give an indication of the actual performance overhead, performance figures have been generated for multimedia quality of service (QoS) management features in (Blair and Reiff-Marganiec, 2001) and telephony features in (Reiff-Marganiec, 2001), both based on a Haskell implementation (see table 4 below). We have also partially implemented a Java component manager, but are not yet at a stage when we can generate performance figures for this.

Table 4. Known performance figures (all figures are in seconds)

		Construction and pruning	
		Construction only	
<i>Multimedia QoS management features</i>			
initial 2 management features		<0.02	<0.02
more advanced management strategy		<0.02	0.36 – 0.40 *
<i>Telephony features</i>		<i>original</i>	<i>on-the-fly</i>
4 features	0.04	0.08	0.02
6 features	0.74	0.92	0.32
7 features	32.24	33.94	1.40

Note that the relatively high overhead marked with a * is almost entirely due to the mechanism used in the Haskell implementation for matching regular expressions on input sequences. Since this can be performed at design-time, or at latest system start-up, it can be removed from the actual cost of run-time resolution (Blair and Reiff-Marganiec, 2001). The performance figures for the telephony features exclude this start-up latency, and are also for where pruning takes place as soon as possible, using an on-

the-fly technique, thus avoiding the complete construction of undesired solutions. It is this latter approach that provides very promising results and indicates the scalability of the method.

6. Conclusions

This paper has presented an approach originally developed for the detection and resolution of feature interactions, and has shown how it can be applied to component interactions. Thus, the contribution of this work is not in the technique itself but rather in its application. In order to evaluate the technique, we have presented a component-based example, developed specifically for this paper. Implementation of a Java-based component manager is also underway, building on our knowledge and understanding acquired throughout our work. From the performance results above, it is clear that we must adopt an on-the-fly approach to the construction and pruning of the behaviour tree in order to gain acceptable performance and scalability in a run-time environment. Notably, the technique provides an advantage over other resolution schemes, e.g. order of events or simple priorities, as it can handle scenarios where the responses of all components need to be taken into account, as is the case in our example.

In terms of future work, we need to complete our Java implementation in order to validate that the expected performance results do indeed apply to component-based systems. There is also a very interesting question of how to distribute the component (feature) manager. In a distributed system, rather than a single component manager as described above, it seems likely that there would be one manager per node (at least). This leads to the potential for the component managers themselves to have undesirable interactions.

Acknowledgements

The authors wish to thank Gordon Blair and Muffy Calder for their help and support and for the many discussions on issues relating to this paper. We would also like to acknowledge that the method was developed during Stephan's previous period of employment, at the University of Glasgow. Thanks are also due to EPSRC, in particular for funding the FILBETT project (Lancaster University, GR/N35939/01), the HFIG project (University of Glasgow, GR/M03429/01) and the FORCES project (GR/M00275/01).

References

- Allen, R. and Garlan, D. (1997). "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, 6(3), pp 213-249.
- Blair, L. and Pang, J. (2000). "Feature Interactions - Life Beyond Traditional Telephony", In (Calder and Magill, 2000), pp 83-93.

- Blair, L., Blair, G. S., Pang, J. and Efstratiou, C. (2001). "'Feature' Interactions Outside a Telecom Domain", Workshop on Feature Interactions in Composed Systems, ECOOP01, Budapest, Hungary.
- Blair, L. and Reiff-Marganiec, S. (2001). "Runtime Detection and Resolution of Interactions of Multimedia Features", Submitted for journal publication, August. Contact either author for further information.
- Du Bousquet, L. (1999). "Feature Interaction Detection using Testing and Model-Checking: Experience Report", In World Congress on Formal Methods, Toulouse, France, Springer.
- Calder, M., Magill, E. and Marples, D. (1999). "Hybrid Approach to Software Interworking Problems: Managing Interactions between Legacy and Evolving Telecommunications Software", IEE Proceedings - Software, Vol. 146, No. 3, pp167-175.
- Calder, M. and Magill, E. (eds). (2000). "Feature Interactions in Telecommunications and Software Systems VI", Glasgow, Scotland, Amsterdam: IOS Press.
- Calder, M., Magill, E., Reiff-Marganiec, S. and Thayanathan V. (2000). "Theory and Practice of Enhancing a Legacy Software System", in "Systems Engineering Business Process Change", Henderson P. (ed), pp 182-195, London: Springer Verlag.
- Dini, P., Boutaba, R. and Logrippo, L. (eds). (1997). "Feature Interactions in Telecommunications Networks IV", Montreal, Canada, Amsterdam: IOS Press.
- "FIREworks: Feature Integration in Requirements Engineering", Esprit Working Group 23531, M. Ryan (coordinator), <http://www.cs.bham.ac.uk/~mdr/fireworks/casestudies.html>
- Garlan, D., Monroe, R.T. and Wile D., (2000). "Acme: Architectural Description of Component-based Systems". In (Leavens and Sitaraman, 2000).
- Griffeth, N.D. and Velthuisen, H. (1994). "The negotiating agents approach to runtime feature interaction resolution", Proceedings of the 2nd International Workshop on Feature Interactions in Telecommunications Systems, Bouma, W. and Velthuisen, H. (eds), Amsterdam: IOS Press.
- Hall, R.J. (2000). "Feature Interactions in Electronic Mail". In (Calder and Magill, 2000), pp 67-82.
- Leavens, G.T. and Sitaraman, M. (2000). "Foundations of Component-based Systems", Cambridge: Cambridge University Press.
- Luckham, D.C. and Vera, J. (1995). "An Event-based Architectural Description Language", IEEE Transactions on Software Engineering, 21(9), pp 717-734.
- Kimble, K. and Velthuisen, H. (1995). "Feature Interaction Benchmark", Discussion paper for the panel on Benchmarking at FIW'95 (Feature Interaction Workshop).
- Kimble, K. and Bouma L.G. (eds). (1998). "Feature Interactions in Telecommunications and Software Systems V", Lund, Sweden, Amsterdam: IOS Press.
- Marples, D. and Magill, E. (1998). "The Use of Rollback to Prevent Incorrect Operation of Features in Intelligent Network Based Systems", in (Kimble and Bouma, 1998), pp 115-134.
- Medvidovic, N. and Taylor, R.N. (2000). "A Classification and Comparison Framework for Software Architecture Description Languages", IEEE Transactions on Software Engineering, Vol. 26, No. 1, pp. 70-93.
- Reiff-Marganiec, S. (2000). "Identifying Resolution Choices for an Online Feature Manager", in (Calder and Magill, 2000), pp 113-128.
- Reiff-Marganiec, S. (December 2001). "Runtime Resolution of Feature Interactions in Evolving Telecommunications Systems", PhD thesis, Department of Computing Science, University of Glasgow, Glasgow, Scotland.
- Roman, E. (1999). "Mastering Enterprise Java Beans and the Java 2 Platform, Enterprise Edition", New York: John Wiley and Sons.
- Shaw, M. and Clements, P. (1996). "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems, Computer Science Department and Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213.
- Szyperki, C. (1998) "Component Software, Beyond Object-Oriented Programming", New York: ACM Press/ Addison-Wesley.