Separating Interaction Concerns from Distributed Feature Components

Jianxiong Pang Lynne Blair

Computing Department

Lancaster University, Bailrigg

Lancaster, LA1 4YR, U.K.

j.pang@lancaster.ac.uk lb@comp.lancs.ac.uk

Abstract

Interaction problems between heterogeneous applications require consideration of the semantic issue of reliable composition. This problem has become significant and ubiquitous in distributed systems as the Internet rapidly grows as a mainstream service platform and requires increasing automatic coordination and cooperation between services at two ends. A feature must be able to adjust itself to work with other features or services - a highly relevant problem called feature interaction. In line with this, in this paper we propose a complexity controlling method that is suitable for distributed systems in which each feature has two concerns, namely a hard logic and a soft logic. The hard logic implements exactly the specification of a feature, while the soft logic deals with the adaptation aspects of a feature, i.e. resolving interaction problems and making features work together. A two level architecture, particularly designed for aspect oriented programming, is described with a meta level being used to describe interaction resolution, with features being at the base level. Through a case study of email systems, we explain the architecture and highlight the cause of resolution interaction problems and how this particular problem is solved.

Keywords: aspect-oriented programming, feature interaction, feature interaction resolution, metalevel, adaptation, hard logic, soft logic, feature-driven development.

1. Introduction

Recently, the potential for undesirable *interactions between heterogeneous applications* has increased rapidly. The phrase "interactions between heterogeneous applications" refers to the co-execution or cooperation of *loosely related software components*. "Loosely related software components" means that the software units were developed without a strictly unified design process. They might come from different domains or providers or even the same provider but from a different development team, or from the same team but developed at different time. Maintaining the semantic reliability when composing such software units is vital to the quality of services. An adaptive capability must be provided to facilitate the smooth resolution of semantic conflicts, and permit coordination and cooperation between different feature components. This has been a difficult area within distributed systems. A very related example is the *feature interaction (FI) problem* experienced by the telecommunication industry, e.g. as surveyed in [15]. We argue that this problem is significant and ubiquitous in distributed systems, as the majority of software development migrates to be Internet- or web-based systems (e.g. web services, agents and p2p) and changing requirements and faster time-to-market become top concerns of a software product.

A *feature,* as a term, is used for describing a small piece of particular interesting capability/functionality. In FDD [18], a feature is "a client-valued function that can be implemented in two weeks or less". Within telecommunication systems, it is "a unit of functionality existing in a system and usually perceived as having a self-contained functional role" [2]. Telecommunication systems have a tradition of organizing development

projects, people, and even marketing by features [17]. Microsoft has also apparently followed this process in their software product line for a number of years [12]. Lessons learnt from these systems are valuable when using FDD methodology to develop a wider-range of distributed software. The *feature interaction problem* involves an undesired feature interaction in which "the behaviour of one feature is affected by the behaviour of another feature or another instance of the same feature" [9].

Although the FI problem has been largely associated with the telecommunication industry, it is not limited to the telephony domain. Any complex software systems that need to evolve frequently or to compose with feature components from different providers would have the same problem. For example, [8] documents interactions occurring within an email system. One such scenario concerns the interaction between *AutoResponder* and *ForwardMessage*, as follows:

Bob sets up forwarding to Lynne who has Auto-Responder enabled. A third party sends a message to Bob, which is forwarded to Lynne. The auto responded message is sent back to Bob and then forwarded to Lynne. According to the specification, AutoResponder only responds to the same source address once and hence the forwarded message is discarded by Lynne's AutoResponder feature. Thus, any messages sent to Lynne via Bob are not effectively auto-responded.

Numerous well-known cases of feature interactions in traditional telecommunication systems are documented in [6] and [9]. More wide-ranging interaction cases have also been identified in [7] (in a variety of miscellaneous examples) and in [4] and [1] (in multimedia, mobile and internet services, and component-based middleware respectively). Our literature investigation has found that the service composition problem [11], interaction merging problem [5] and insecure component composition [19] can also be considered as some form of feature interaction problem.

FDD emphasizes *small features* as building blocks; a system will inevitably have many features that potentially complicate the interaction issue. Our solution is to separate the interaction issues from the feature's functional logic and allow for convenient plugging/unplugging of the interaction resolution module. In other words, we raise the feature interaction concern up to the meta-level. We use aspect-oriented programming [10] to describe this metalevel.

The work described in this paper has pushed our previous feature interaction study further by studying more acute feature interaction problems. Specifically, this paper presents two *inter-provider* feature interaction problems in email systems. The resolution strategies for FIs of this kind are much harder to find when compared to that of [16]. Importantly, this paper takes the issue of resolving interaction problems further by considering two resolutions that themselves interact. A solution to the composition of resolutions based on operation precedence is proposed. The ongoing study in this paper further enriches previous work with some initial knowledge regarding the creation of domain-specific resolution pattern libraries. We plan to build up these pattern libraries gradually by large-scale case studies.

This work also further extends our earlier work on the study of feature interaction [4][1], run time managers [2] and auto adaptation [3].

2. Case Study

We choose an email system as our case study. An email system is a typical application of an Internet system, reflecting many interaction problems in distributed and communication-intensive systems. Our cases come from [8]. Due to the space limitation, we select FilterMessage, ForwardMessage and RemailMessage features for discussion because they help to illustrate very interesting interaction properties, e.g. resolution interaction as will be discussed later in this paper. We follow the same approach as [8] regarding the architecture (pipe&filter) and the features, i.e. a user originates a message, from an email client program. This message then passes through one or more feature processing components, termed *email feature components* (efc), until it is delivered to the email client of the intended recipient(s). For simplicity, the routing issues will be ignored.

FilterMessage: This efc is provisioned with a list of address suffixes. Any message received whose sender's address has a suffix on the list is simply discarded (filtered). Other messages are passed on without change. This is modelled on existing spam filters.

ForwardMessage: This efc is provisioned with an email address. Every message received is remailed to the provisioned address. This is useful, for example, if one moves to a new email account provider, either permanently or temporarily. If no address is provisioned, then the efc simply transmits the input message unchanged.

RemailMessage: This efc allows anonymous messages to be sent. It is provisioned with a mapping from user addresses to pseudonyms. A user sends a message to remail@rmhost with the intended recipient as the first line of the body of the message. The remailer then looks up the pseudonym of the sender and replaces the actual sender address with the pseudonym in the headers and envelope of the message, finally sending on the modified message. To handle replying, anyone may send in the other direction as well: a message addressed to <pseudo-nym>@rmhost is first translated to have the true name of the recipient and then sent accordingly. Thus, the user of the remailer is assured of anonymity, as long as nothing in the body of the message gives their identity away.

Two interactions among these features have been found. They were documented in [8] as follows:

Scenario 1*: suppose Bob sets a ForwardMessage feature to forward any incoming message to a new account of his in another email server. But the administrator of that server happens to use a FilterMessage feature to filter all messages from Bob's domain without informing Bob of this. Consequently, when Bob move to collect messages in the new domain, he will not receive any message from then on. The FilterMessage subverts the ForwardMessage!*

Scenario 2*: Suppose we situate in the domain "lancaster.ac.uk", and provision FilterMessage to discard all messages from the domain "devil.co.uk". However, a user from within "devil.co.uk" obtains a RemailMessage pseudonym and sends a message to lynne@lancaster.ac.uk through RemailMessage. This message is not filtered out, but instead delivered to Lynne. Thus, RemailMessage subverts FilterMessage.*

These are notorious feature interactions that are the hardest to resolve for the following reasons:

1. Features belong to and reside at two different providers. Both providers try to achieve their own goals, and follow their own interests.
2. The conflict is acute, therefore difficult to reconcile. In favour of any one side might acutely harm another side's interest.

In scenario 1, Bob's interest is to forward the incoming message to another account, so as to, for example, read all messages in another domain. In contrast, the administrator's interest is to filter the message from Bob's domain in order to prevent spam/virus email from arriving in its domain.

Similarly, in scenario 2, both features have their worth of existence while their goals are acutely conflicting. FilterMessage is obviously a reasonable feature; all in all, everybody has their right to filter messages they do not want. Similarly, RemailMessage is also a reasonable feature; for privacy, a user's request for anonymity should be respected.

These interactions can also be mapped to many other applications, for instance, "*Incoming Call Screening*" vs. "*Undelivery of Number*" in telephony systems. *Incoming Call Screening* feature is analogous to FilterMessage, while *Undelivery of Number* is similar to RemailMessage, thus there is obviously a similar feature interaction between them. *Anonymous servers* in the WWW cause quite similar problems for *IP screening* as well. The resolution of this kind of conflicts is inevitably an important part of systems if we want to yield better quality of services. Generic techniques such as negotiation agents or arbitration mechanisms are developed to modularise resolution concerns. For example, in [14], a solution based on win/win principle is proposed, where negotiation attempts to find an alternative proposal that is acceptable for both sides. However, most of the existing programming paradigms force developers to program any resolution code into the core functionality of a feature (we refer to this as a feature's hard logic). The entanglements of different functional roles can quickly complicate a system, making it harder to maintain and evolve. This insight has led us to propose a two-level architecture for complexity control.

One thing that must be pointed out here is that although we will suggest resolutions for each feature interaction case in this paper, we have no intention to strictly validate them, because our focus is on the separation techniques, rather than the FI resolution issues themselves. As an aside, we believe there is no precise definition of *resolution*. The reason is that resolutions on the same feature interaction problem may vary from developer to developer. When we say something is a resolution of an interaction, it is quite subjective. Sometimes it just meets a requirement of feature users, other than a sound rationalization. In this paper, we assume that any solution that is able to mitigate an acute feature interaction constitutes a *resolution* of that interaction. Therefore, the simplest resolution is to disable one of the interacting features. However, real world applications might need a more deliberate resolution so as to improve the quality of service.

3. Separation of Interaction Concerns

In our proposed framework, we assume that every feature has a clear specification of its functionality. Although the implementation of that specification varies, it is generally easy to distinguish the pure feature code. We call this the *hard logic* of the feature, i.e. the inevitable part for the implementation of specification.

However, in feature driven development, features must clearly be able to work with other features. Since hard logic is actually rigid business logic, it is unable to adapt itself to different execution contexts (different connected features). We also need a corresponding *soft logic* to soften the behaviour, making it flexible enough to adapt to other interacting features. Therefore, a feature's soft logic is responsible for gluing features together and taking action to smooth any incompatibilities.

Since a feature designer cannot foresee the future features that will interact with his/her developed features, soft logic should be able to be added to the hard feature logic at any stage, i.e. complementing the inevitable lack of meticulousness with interaction resolution issues when hard logic is designed. To support this kind of addition, it is ideally raised up to the meta-level so as to provide a separation from the hard logic and facilitate reuse and easier maintenance/evolution.

It is this soft logic that we believe is ideally suited to aspect oriented software development techniques.

As an example, an overview of a possible java implementation of FilterMessage's hard logic is shown in figure 1:

```
class Filter implements Pipe {
  String myID;
  ArrayList screened;
  public Filter() {
   //code to construct a Filter Box
  }
  public void send(Message msg) {
  //Typically send to the MailHost feature for
  //delivery
  }
  public void receive(Message msg) {
    filterFeature(msg);
  }
  private void filterFeature(Message msg) {
    String sender = msg.getSender();
    if(!isFilteredDomain(sender))
      send(msg);
    else
      discard(msg);
  }
  public boolean isFilteredDomain(String
                                   sender){
     String domain =
      sender.substring(sender.indexOf("@")+1);
     return screened.contains(domain);
  }
  public void discard(Message msg) {
  //the code to discard the message
  }
}
```

Figure 1. FilterMessage's hard logic, implements only what the specification specifies

The hard logic takes care of filtering the incoming message against a filter list. In order to do this, for a incoming message, it will get the sender's address and check it against the filter list, then decide to either deliver or discard it depending on the checking result. The Pipe interface, which contains two methods, receive (..) and send(..), must be implemented for the connection of feature boxes.

We can see that the hard logic of a feature is simple, cohesive and highly consistent with its original specification. Typically, features have two basic parts:

1. Some data (structures) such as a forward address, a list of filter addresses or a list of <pseudonym, real name> pairs.
2. Some methods to operate on the data and provide necessary feature logic to implement a service feature.

To illustrate the separated soft logic of features, we will list two typical examples.

FilterMessage vs. ForwardMessage

For the interaction between ForwardMessage and FilterMessage, there are many ways of resolution. One way might use a form to ask a feature owner to specify options/preferences/policies for dealing with the interaction so as to form data for negotiation, while another way might just design a default resolution/policy. We show a simple resolution based on a default policy here. As ForwardMessage is the passive party of this interaction, the policy needs deciding by FilterMessage. A reasonable default policy of FilterMessage might be "allow Bob's ForwardMessage feature to forward all third party messages to his new account as long as the third party is not from a domain that is filtered by the FilterMessage ". As this email server itself has created an account for Bob, Bob has already obtained some certain privileges anyway. Thus there is no reason to prevent him from sending and receiving message from the server though his original domain is screened. Of course, messages from other users at the same domain as Bob will continue to be filtered out. In brief, the default resolution is described as follow:

> *Every time FilterMessage is about to discard a message, it should additionally check if the message is from ForwardMessage; if so, it further checks if the forwarded message is from a forbidden domain; if not, then let the message go through without filtering it out. Otherwise, discard the message as the basic feature function prescribes. To let FilterMessage know that the message is a forwarded message, ForwardMessage must add a <forward> tag in the content with original sender's address that allows FilterMessage to check against its filtering address list and decide if it is filtered out or not.*

The following code in AspectJ [10] shows an implementation of the resolution.

```
aspect  SaveForwardedMessage {
 void around(Message msg,Filter filter):
   call(void Filter.discard(Message)) &&
          args(msg) && target(filter) {
  if (isForward(msg)){
   String address = getAddressOfForwarded(msg);
   if (!filter.isFilteredDomain(address))
    //send to MailHost to deliver
    filter.send(msg);
   else proceed(msg,filter);
  }
  else proceed(msg,filter);
 }
boolean isForward(Message msg) {
   ……
}
String getAddressOfForwarded(Message msg) {
    …….
}
}
```

Figure2. Resolve FI between FilterMessage and ForwardMessage

FilterMessage vs. RemailMessage

To resolve the interaction problems between FilterMessage and RemailMessage, we must answer the question "what is the real goal of the filter?" In many cases, the FilterMessage feature is deployed to filter messages from a domain that has security risk, e.g. potentially carrying a virus. If this is the case, the FilterMessage only needs to care about the virus-vulnerable part - the attachments. Perhaps we can require that messages with attachments must obtain a certificate from a trusted organization to prove no virus exists, so as to permit the delivery. Based on this insight, we proposed a resolution as follows:

> *Every time FilterMessage is about to deliver a message, it should additionally check if the message is from RemailMessage; if so, further check if the remailed message contains any attachment; if yes, then check if this attachment is certificated; if not, then instead of delivering the message, just discard it. Otherwise, the message will be delivered as the basic feature logic prescribes. To let FilterMessage know that the message is a remailed message, RemailMessage must add a <remail> tag in the content. Also any attachment in a remailed message should attach a certificate from a trusted organization.*

In AspectJ, the resolution can be implemented as follows:

```
aspect  StopDoubtfulRemailMessage {

 void around(Message msg,Filter filter):
     call(void Filter.send(Message)) &&
     args(msg) && target(filter) {

     if (isRemail(msg)&&
         isContainAttachment(msg)&&
         !isCertificated(msg)
        )
        filter.discard(msg);
     else proceed(msg,filter);
  }
  boolean isRemail(Message msg) {
   .......;
  }
  boolean isContainAttachment(Message msg) {
  .......;
  }
  boolean isCertificated(Message msg) {
  .......;
  }
}
```

Figure 3. Resolve FI between FilterMessage and RemailerMessage

4. Composition Problems of Resolutions

The previous section has shown that the use of aspect-oriented programming techniques for the representation of FI resolution is an effective way of feature composition. It is also flexible with respect to further evolution of the system. However, feature interaction problems are complicated issues, and a resolution is unlikely to be independent of other resolutions. This is not unexpected, since resolutions themselves can be viewed as features, which, of course, are prone to interactions. Both interaction resolutions in the previous section require new behaviour (or "advice") around FilterMessage. Basically, a FilterMessage feature is used for discarding unwanted messages. For every incoming message, it either delivers the message or discards it. Interestingly, there is an antithesis between the two resolutions. The first, i.e. the case of FilterMessage vs. ForwardMessage, says that a forwarded message sometimes should not be discarded because it might originally come from a non-screened party though the forwarder is from a screened party. The resolution for this circumstance is that we must *rescue* it from being discarded. While the second, i.e. the case FilterMessage vs. RemailMessage, says that a remailed message might need to be discarded because it may disguise its original address and try to fool the filter into believing that it is from a harmless party. The resolution for this case is that we have to bar it before it is *almost* delivered. If we observe the two aspect algorithms in Figure 2 and 3, we can find the antithesis. To analyse the interaction problem clearly, we simplify it as follows:

```
StopDoubtfulMessage =
  advice around the deliver method ;
  action: might change to run discard.
```
(a)
```
SaveForwardedMessage =
  advice around the discard method ;
  action:might change to run deliver.
```
(b)

Figure 4. A concise description of resolutions for FilterMessage interacting with ForwardMessage & RemailMessage

When composing the two resolution features together, the problem is: while one resolution requires discarding a message, the other resolution wants to deliver it.

Before giving a solution to this problem, we need to form a comprehensive view about the messages that are join-created by RemailMessage and ForwardMessage. The typical configuration is showed as follow:

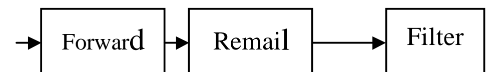

Figure 5. A configuration of three features whose resolutions themselves interact

After passing through ForwardFeature and RemailFeature, the status of such a message can be expressed as:

<rt,att,c,ft,o,s>,

where *rt* denotes a remail tag, and the other:

att - attachment,

c – Is the attachment certificated or not,

ft - forward tag,

o - Is the forwarder's address screened or not?

s - Is the original sender screened or not?

Note that there are dependencies between these elements. Namely if there is no attachment, there won't be any certificate. Similarly, if there is no forward tag, there won't be any original sender address of forwarding. Among all possible values of this status array, consider the following configuration:

<rt,att,c,ft,o,s> = <Yes,Yes,No,Yes,No,Yes> .

It can be interpreted as:

"*A message contains a remail tag and attachment, but the attachment is not certificated, contains forward tag, the original address of forwarded message is not screened, but the sender address is screened.*"

Obviously this message should not be delivered. The resolution for the interaction is to let the **StopDoubtfulMessage** have higher priority. However, down to the implementation level with AOP as the tool, the composition is realised as weaving and the resolution interaction becomes aspect interaction. We have found that AspectJ lacks language-level features to support the elegant expression of our resolution of this aspect interaction. The direct weaving of the two aspects results in an infinite loop, because they advise each other infinitely. The "*dominate"* construct in AspectJ is irrelevant in this case because the execution of the two aspects require some inter-nested coordination. As AspectJ stands at the moment, the language does not support "aspects of aspects", i.e. aspects cannot be defined over other aspects. Thus, though we can manually add resolution code (figure 6), it is not so clean and intuitive.

```
aspect  SaveForwardedMessage {
 void around(Message msg,Filter filter):
   call(void Filter.discard(Message)) &&
          args(msg) && target(filter) &&
          !within(StopDoubtfulRemailMessage) {
  ………..
}
```

Figure 6. Modification to compose the resolutions correctly.

At this point, it should be noted that other AOP approaches do provide support for "aspects of aspects". In an email message posted on the Demeter web-site [20], a list of approaches supporting this is given, namely Incremental Programming [25], Aspectual Collaborations [23], Hyper/J [26] and DJ [24]. These require further investigation to determine if they would provide a more appropriate language choice for our work than AspectJ.

5. Evaluation of our approach

We claim that the two-level architecture we have proposed has key benefits regarding flexibility with respect to future evolution of systems. In order to evaluate the architecture against this claim, we have evolved the email system presented so far, by extending it to all ten features of [8]. In order to make a working system, we also refactored some of the GUI modules from ICEMail, an email client written in Java and based on the new Java Mail API [21].

We can classify our evaluation into a number of different properties including cleanness of separation, re-use, faithfulness of implementation to specification, adaptability to requirement change, and performance. It should be noted that these properties are, by their nature, more qualitative than quantitative.

A summary of these properties is presented below; for more details the reader is referred to [28].

Cleanness of separation

- The approach avoids the tangling of core behaviour with resolution code (to allow a feature to work with other features).
- All features from [8] illustrate an elegant separation when implemented. Note that not every interaction requires a separate resolution module, see below, thus motivating our search for more general interaction resolution *patterns*.
- By refactoring some of the ICEMail GUI's modules, one module was reduced from 800 lines of source code to 70 by removing what we classified as tangled concerns or interactions.

Re-use

- Reuse for very specific interaction resolution modules (e.g. figures 2 and 3 above), is limited. The best opportunities for re-use are at the base level rather than the meta-level due to the modules' cleanness and simplicity.
- The refactored GUI modules mentioned above have been easily re-used in a second implementation since all interaction concerns had been extracted thus leaving a generic feature component.
- Many interaction resolutions are similar in that they involve boundary condition checking. For the 26 feature interactions identified in [8], plus an additional feature interaction identified by ourselves, more than half are generic resolutions, hence the identification of resolution patterns appears a realisable aim.

Faithfulness of implementation to specification

- By design, our two-level architecture keeps the feature's implementation faithful to its specification.
- This opens the door to generative programming techniques to generate code (or code templates) automatically from the specification.

Adaptability to requirement change

- The separation provided by our architecture allows the developer to integrate new features into the system, without needing to consider, or worse rewrite, existing features.
- The aspect-oriented approach for the separate resolution modules allows the developer to implement a feature without considering the interactions with other features, then focus on the interaction issues separately.
- Removal of features from a system is also clean and efficient (the architecture helps to avoid redundant code being left embedded in feature boxes, a situation that leads to unnecessary complexity and low efficiency).

Performance

- In line with other meta-level/ reflective approaches, our approach will likely incur at least a minor performance overhead. We have not yet investigated this further, although information regarding the performance of AspectJ can be found on the AspectJ web-site (FAQ), see [21].

6. Related Work

Although the era of traditional telecommunications is rapidly passing, the feature interaction research that germinated from this particular industry is becoming of increasing importance. Negotiation agents, as a general resolution for inter-provider feature interactions, have grown as an active research area recently [13][14]. Negotiation is inherently a cooperative search for a certain value pattern in a global policy data model, which is a meta-data structure. In [13], it is carried out via a blackboard model where different features declare their intentions. Policies/preferences of features are represented as CLIPS rules and the negotiation process is powered by CLIPS rules engine. In [14], all policy features are classified as a hierarchy structure, called a *goal hierarchy*, where policy is specified. The negotiation process uses this hierarchy structure to find an agreement between the two sides. Thus, rules and the algorithms that operate on rules are functionally comparable to our aspect program. This implies that negotiation can be treated as an interaction concern and negotiation can be implemented as aspects. The two examples of this paper are functionally similar to a negotiation process but are obviously simpler than conventionally rule-based negotiation. Therefore we believe that aspect oriented programming has the potential to provide an effective alternative to rule-based negotiation programming. The similarity of rewriting rules with aspect programs in [5] seems to suggest that this is a promising approach.

In [11], an approach to describe CORBA services (event service, transaction service, etc.) as aspects has been presented. To resolve the service composition problem, which has different aims comparing with our feature interaction problem, a service composition model was proposed in which there are three levels of architecture, namely the application level (base level), service level (meta level or aspect level), and composition level (meta-meta level or aspect of aspect level). The separation of the composition issue from the aspect accelerates the reuse of the composition pattern and helps to simplify those two level compositions, but at the cost of defining a so-called aspect on aspect language and its corresponding compiler. We think that this is still not a one-shot solution to all compositions because there is no a clear closure of the composition operation in a complex system. An aspect-on-aspect may still need to compose with another aspect-on-aspect in some cases. In the end, we still face the same problem as in the lower meta level.

6. Conclusions and Future Work

Today's distributed systems are prone to feature interaction problems, largely because of the heterogeneous service nature. We believe that the separation of interaction concerns is key to the success of reusability and maintenance of an evolving system. Hard logic and soft logic are metaphors for the relationship between a feature's functional logic and its adaptation/resolution logic. The soft logic softens the hard logic so as to allow it to adapt to a feature interaction. Lifting up the adaptation code to a meta level is the vital decision for the separation. The emerging area of aspect-oriented programming is a suitable platform to represent these interaction resolutions. By using aspects, the implementation of dynamic reconfiguration or auto adaptation becomes very convenient.

The feature interactions in distributed systems, including Internet-based systems are still not well understood. More investigation is needed to abstract further *interaction resolution patterns*, and further *interaction resolution pattern libraries* for different domains (e.g. p2p, multi-agent and web services). The focus of interaction resolution is the composition problem, namely the semantic conflicts occurring when two interaction resolutions composing together.

Furthermore, as negotiation can be treated as a sub-interaction concern, utilizing AOP to develop negotiations required between conflicting services is also a valuable new direction.

References

[1] L.Blair, G.Blair., J.Pang. & Efstratiou C., 'Feature' Interactions outside a Telecom Domain, Workshop on Feature Interactions in Composed Systems, held at ECOOP2001, Budapest, Hungary, 18-22 June 2001.

[2] L. Blair, T. Jones and S. Reiff-Marganiec, A Feature Manager Approach to the Analysis of Component-Interactions, In Proceedings 5th International Conference on Formal Methods for Open Object-Based Distributed Systems, 20-22 March, University of Twente, The Netherlands, Kluwer,2002.

[3] J.Pang, L.Blair, An Adaptive Run Time Manager for the Dynamic Integration and Interaction Resolution of Feature, In Proceedings 22nd International Conference on Distributed Computing Systems Workshops, pp445-450 IEEE, Los Alamitos, California, 2002.

[4] L.Blair. & J.Pang., "Feature Interactions - Life Beyond Traditional Telephony", In [6], pp 83-93, 2000.

[5] M.Blay-Fornarino, A.Pinna-Dery and M.Riveill, Towards Dynamic Configuration of Distributed Applications, In Proceedings 22nd International Conference on Distributed Computing Systems Workshops, pp487-492 IEEE, Los Alamitos, California, 2002.

[6] M.Calder, E.Magill , editors, "Feature Interactions in Telecommunications and Software Systems VI", Glasgow, Scotland, IOS Press(Amsterdam), 2000.

[7] M. Ryan (Coordinator), "FIREworks: Feature Integration in Requirements Engineering", Esprit Working Group 23531, started 1997. See www.cs.bham.ac.uk/~mdr/fire-works/casestudies.html.

[8] R.Hall, "Feature Interactions in Electronic Mail", In [6], pp 67-82, 2000.

[9] E.Jane Cameron, Nancy D.Griffeth, Y. Lin, M.Nilson, W.Schnure, and H.Velthuijsen. A feature-interaction benchmark for IN and beyond. IEEE Communications XXXI(3):64-69, March 1993.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, Getting started with ASPECTJ , Communication of the ACM, Vol. 44, Issue 10, ACM Press (New York), pp59-65, October 2001.

[11] L. Bussard, Towards a pragmatic composition model of Corba services based on AspectJ, Mémoire de DEA, Université Nice-Sophia Antipolis, juin 2000.Available VIA:www.essi.fr/~riveill/ARCAD/publications_du_projet.html

[12] M.A.Cusumano and R.W.Selby, "Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People", Simon & Schuster, 1998. ISBN: 0684855313.

[13] Buhr, R. J. A., Amyot, D., Elammari, M., Quesnel, D., Gray, T., and Mankovski, S.: Feature-Interaction Visualization and Resolution in an Agent Environment. In: Fifth International Workshop on Feature Interactions in Telecommunications and Software Systems (FIW'98), Lund, Sweden, October 1998. IOS Press, Amsterdam, 135--149.

[14] Nancy D. Griffeth and Hugo Velthuijsen. The negotiating agents approach to runtime feature interaction resolution. In Bouma and Velthuijsen [3], pages 217--235.

[15] M Calder, E. Magill, M. Kolberg, and S. Reiff-Marganiec. Feature Interaction: A Critical Review and Considered Forecast. Accepted for publication, Computer Networks, North-Holland. 2002. Available via http://www.dcs.gla.ac.uk/~muffy/papers.html

[16] J.Pang, L.Blair, Aspect Solutions to Decentralised Interaction Concern, currently under submission, August 2002, Available from Computing Department, Lancaster University, LA1 4YR, UK.

[17] P.Zave, " FAQ Sheet on Feature Interactions ", AT&T, 2001, Available via: http://www.research.att.com-/~pamela/faq.html.

[18] P.Coad and S.Palmer, Feature-Driven Development, TogetherSoft Corporation 2002, Available via http://www.thecoadletter.com/download/bookpdfs/JMCUCH06.pdf.

[19] P. Sewell and J. Vitek, Secure Composition of Untrusted Code: Wrappers and Causality Types, 13th IEEE Computer Security Foundations Workshop (CSFW'00) July 03 - 05, 2000, Cambridge, England.

[20] "Aspects of aspects", email correspondence on Demeter website, www.ccs.neu.edu/research/demeter/related-work/aspects-of-aspects/reading-list , 2000.

[21] "Getting Started with AspectJ", Communications of the ACM, Vol. 44, No. 10, pp. 59-65, 2001. See also http://www.eclipse.org/aspectj/

[22] ICEMail email client, http://www.icemail.org. See also D. Nourie, "The Java™ Technologies Behind ICEMail: An Open-Source Project", June 2001, see http://developer.java.sun.com/developer/technicalArticles/javaopensource/icemail/.

[23] K. Lieberherr, D.H. Lorenz and J. Ovlinger, "Aspectual Collaborations: Combining Modules and Aspects", Technical Report NU-CCS-02-?, 2002. Available from http://www.ccs.neu.edu/research/demeter/papers/publications.html

[24] D. Orleans and K. Lieberherr, "DJ: Dynamic Adaptive Programming in Java", Proceedings of Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, Springer Verlag, 2001. www.ccs.neu.edu/research/demeter/biblio/DJreflection.html

[25] D.Orleans, "Incremental Programming with Extensible Decisions", First International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, G. Kiczales (ed), ACM Press, 2002. See also http://www.ccs.neu.edu/home/dougo/papers/aosd02/

[26] H. Ossher and P. L. Tarr, "Hyper/J: multi-dimensional separation of concerns for Java", International Conference on Software Engineering, pp. 734-737, ACM, 2001. See also http://www.research.ibm.com/hyperspace/

[27] J. Pang and L. Blair, "Resolving Feature Interactions with AOP and Feature Driven Software Development", draft paper available from Computing Department, Lancaster University, Lancaster

[28] L. Blair and J. Pang, "Aspect-Oriented Solutions to Feature Interaction Concerns using AspectJ", currently submitted, available from Computing Department, Lancaster University, Lancaster