

An Adaptive Run Time Manager for the Dynamic Integration and Interaction Resolution of Features

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

With modern software systems, an important requirement is the ability to be auto adaptive, i.e. being able to adjust itself to the changing environment. In line with this, a run time manager for dynamic feature integration of telecommunication systems, interaction detection and resolution is described in this paper with aspects being used to implement features. The manager manages the interaction of features/aspects by monitoring the managed program. The program is represented by a labelled transition system (LTS) model, stored in a flexible data structure, and executed by calling action subroutine represented by the label of the LTS model, forming a reflective facility for the composition and analysis of features. It is the reflective mechanism that makes dynamic feature addition, run time model checking, as well as adaptive interaction resolution possible. Runtime model checking is possible because the checked program is stored within itself, and the interaction resolution will be done by selecting behaviour traces according to the resolution rules.

Keywords: aspect-oriented programming, auto adaptive systems, feature/aspect interaction detection and resolution, run time manager, reflection, model checking, LTS (Labelled Transition Systems).

1. Introduction

The innovation of next generation Internet brings about new aspects of software development. Among them are dynamic service integration, context adaptation (including

context awareness) and dynamic reconfiguration. In line with these, there is a growing need for auto adaptive systems. An adaptive system is a system where the implementation of the system software can change even while the system is used. The implementation change happens automatically by monitoring and modifying the running system. Adaptiveness is an important concept, because it offers capabilities at run time that conventional systems lack. These capabilities are significant because recent distributed software systems have necessitated decentralized service deployment and more heterogeneous device interaction. Many adaptive systems are supported by mechanisms that permit behaviour addition dynamically. In the context of telecommunication systems, as will be discussed in this paper, this means integrating or composing features/aspects dynamically. Recently, aspect-oriented programming [15] has emerged as one of the approaches for achieving adaptiveness, for example through the definition of “aspects”, i.e. crosscutting concerns. However, as the number of features/aspects grows rapidly, systems get more complex, making consistency difficult to maintain. In our work, we use aspects to implement telecom features that can be seen to crosscut the basic telephony behaviour. Consequently, a highly relevant problem is that of feature interaction [16][6][7][9][12][8].

Features, and feature-interactions, have been the subjects of extensive research within the telecommunications industry over the last 10 years. Within this domain, the term *feature* refers to a unit of additional functionality, added to an existing system and usually perceived as having a self-contained functional role [2]. Examples of telephony features include call forwarding on busy, call waiting, voice mail, etc. Problems arise when new features do not work correctly, or consistently,

with existing ones. This problem is termed the *feature interaction problem* and, more specifically, occurs when “the behaviour of one feature is affected by the behaviour of another feature or another instance of the same feature” [13].

Within the telecommunications domain, there are numerous well-known cases of feature interactions, for example, consider the two features “Call Waiting”(CW) and “Call Forwarding When Busy”(CFB). CW notifies an engaging user if another caller tries to reach him. Then the user can place the existing call on hold, answer the second call, and easily switch back and forth between the two calls. While CFB diverts all incoming calls to another telephone number of choice when the subscriber is already busy with another call. There is now an interaction between CW and CFB. Suppose a customer has invoked both features and is engaged in a call. Another call arrives; should that second call be forwarded or will the customer hear the call waiting alert tone? For further examples see [9], [12] and [8]. However, the interaction problem is not restricted to telecommunication systems. More wide-ranging interaction problems have also been identified in [11] (in email systems), in [10] (in a variety of miscellaneous examples) and in [4] and [1] (in multimedia, mobile and internet services, and component-based middleware respectively).

Many different styles of techniques have been proposed for the detection, and subsequent resolution, of feature interactions. With *off-line* (or design-time) techniques, a model of the base system and its additional features is specified in a formal language while the properties that the system should exhibit are (typically) specified through the use of temporal logic. A wide range of modelling languages has been used, including Finite State Machines (FSMs), LOTOS, Petri-Nets, Promela and SDL. However, as the number of services grows, there is clearly an issue of the scalability of such techniques and tools. Major improvements have been forthcoming in model-checking techniques recently, for example through the use of on-the-fly and symbolic techniques and also the use of abstractions or symmetries. Such techniques help to greatly reduce the state-space explosion problem. A further problem, however, is that the level of success depends on the accuracy and level of abstraction of the specified properties. An inaccurate (or rather, not precise enough) property specification will inevitably lead to interactions remaining undetected (as occurred in [3]). Off-line techniques must also rely on a-priori knowledge of the behaviour of the individual features.

In contrast, adaptive *on-line* (or run-time) 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, while 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.

The AOP community is also recently noting the interaction problems between aspects. Work in [17][18][19] shows similar concepts and examples. For instance, in [19], the usual cause of aspect interaction is given as “several aspects interact when their patterns overlap.” In addition, after defining an AOP framework from a monitoring perspective, the authors further describe some typical aspect interactions on top of this framework. In that framework, execution monitors serve as an operational model for AOP, and the execution of the base program generates events so that monitors can survey the execution. A crosscut can then be defined as a pattern of events, which are tested for each time an event is emitted. Now consider the two following aspects (defined as event pattern), which are extracted from an example in [19]:

The first,

```
c=select_customer();
call_service(s) => charge(c,s)
```

Says that after selecting a customer and calling a service, a function must be called to charge the selected customer for the called service.

The second,

```
c=select_customer();
call_service(s) => authorize(c,s)
```

States that after selecting a customer and calling a service, a function must be called to authorize the selected customer for the called service.

Suppose there is an execution trace:

```
c = select_customer(); ...;
call_service(s);
```

Obviously, both of the above two aspects are cross-cutting at `call_service`. This is clearly an interaction; One possible resolution will let `authorize` take the precedence over `charge`. The issue of resolution will be revisited in section 2 and 3 below.

Complexity is always a challenging problem within software engineering, in our case, with the complexity arising due to a large number of interacting features/aspects. Our response to this challenge is a runtime manager, supporting dynamic feature integration, and more importantly, being deliberately designed for the dynamic detection and resolution of feature interactions. Adaptiveness usually requires some reflection capability, i.e. observing the running system so that the adaptive programs can be automatically adapted. We achieve this by storing the program model in a data structure. Our reflective mechanism not only allows the dynamic examination and integration of features, but also helps to carry out run time model checking on the program model and its features, and provides a resolution whenever feature interactions occur. Furthermore, the resolution is adaptive, in that it can evolve itself according to the detection and resolution result.

This work further extends our earlier work on the study of feature interactions [4] [1] and run time managers [2] [5].

2. The architecture of the run time manager

One of the most valuable benefits of AOP is that it enables a design and implementation based on separate concerns of a complex system - a core system, or framework, and a set of aspects/features. The core-system/framework is usually an intact program that can be executed independently. The aspect set represents a list of features that are added on top of the core system, but that may crosscut the core system. For example, in telephony systems, the base system can be represented by a set of Java classes/methods, while all features such as CFB, TCS (Terminating Call Screening) and TL (Teen Line) are represented by aspect definitions (one per feature). However, the crosscutting creates the potential for aspect-to-aspect or aspect-to-core system interactions. A run time aspect interaction manager needs to know the data that represents the core and aspects' behaviour to support the run time model checking. Note that we choose to provide a *model* of the running program itself, for simplicity but also because it has better support for model checking. We have chosen LTS (Labelled Transition Systems) as a modelling tool, which has proved good for behaviour analysis [14].

The run time manager supports feature integration, monitors program execution, and also checks and resolves feature interactions. The behaviour of the managed program and its associated aspects is represented by LTS models, and stored in a data structure. Note that each LTS transition is labelled with messages that represent actions. The program is executed by calling methods corresponding to the actions occurring in the LTS models.

Furthermore, an interface will be defined in order to expose functions for the manipulation of feature models. Since the system contains information about itself and provides method to change its behaviours, we have a reflective mechanism, and the interface is a meta-interface, using reflection terminology.

Figure 1 shows the architecture of the system. The Manager (run time) is the central module, in which the behaviour model, feature properties and resolution rules are stored. Any stimuli from outside the system are passed to it through a "Messaging system" that acts as middleware. The manager checks the property against the behaviour model and at the same time responds according to the stimuli. The Registry records the configuration information of the system, and Feature logic module contains those action sub-routines that giving semantics to actuate the computation (i.e. messaging and control signals). Therefore, we have an event driven state transition system.

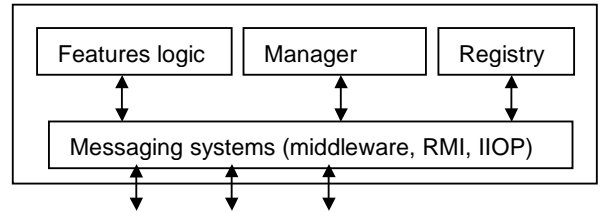


Figure 1. Architecture of our run time aspect-interaction manager

While the ultimate aim of this research is in interaction resolution, resolution depends on first detecting an interaction. Run time detection concerns behaviour checking at the same time as system execution. In the case that the program is executed simultaneously to the detection, when a conflict is found by the detection, the program is at the same time violating the properties. Under this circumstance, the resolution requires a roll back mechanism. The conventional transaction semantics can be applied to this case. A solution, which can avoid the roll back operation, is to do the detection in advance of the program execution, so as to allow for a pre-awareness of the violation. When a violation is detected, the program has not yet executed to the violation point, thus can do the resolution in time before the interaction happens.

A combination of a specific resolution strategy (using "a priori" knowledge of interactions) and generic resolution techniques is applied when interactions are detected. The specific resolution strategy depends on specific feature logics. It offers fine-grained results than generic resolution techniques, but lacks complete coverage, whereas generic resolution techniques are guaranteed to cope with all interactions. As an example of a generic rule pattern, we may require that "a terminal device must never receive message x followed by message y, as that will lead to deadlock" [20]. Another generic technique is to assign a priority value to each feature so as to allow the resolution process to choose one feature with highest priority to run whenever conflict occurs. Specific resolution techniques using pre-determined resolution methods can be found in [21] [Error! Reference source not found.] [2] [2]. Note that in our approach pre-determined resolution is not a mandatory requirement, but rather an option. Conflict resolution always enforces a specific resolution method

first. At the time when specific methods are not found, it then carries out a generic method that is always available.

Note that a program executed around a LTS model of the program itself will be viable for both detection and resolution. In this way, the LTS model has got two roles to play. On the one hand the LTS model is data, therefore is analysed and modified; on the other hand, it is also a program, and executed by taking the transition label and calling the action method corresponding to it.

3. The adaptive capability

The interaction detection is performed by checking the program model against feature properties. The interaction resolution will be guided by the resolution rules. Both the properties and rules are determined at design time (constraints on features and interleaving of operations). All are represented by LTS models, to allow composition with the base model and aspect/feature models to permit model checking. Upon a feature/aspect interaction being found, the manager will choose execution traces according to these rules.

In the literature, model checking is mostly described as a design time tool. Due to the state explosion problem, scalability is usually weak. With the support of our stored models, we are able to postpone the model checking to run time, and term it "implementation checking". Note that our approach is not intended to reduce the state space, but just to check the properties on the current execution path. In other words, rather than centralize the model checking computation in one go, our approach is to decentralize the checking into each specific execution trace at run time. We are only concerned about whether or not the current trace violates the feature properties without being concerned about whether other execution traces contain violations.

All the traces, i.e. the behaviour tree, exist in the model. We can logically view this as a binary tree, where the traces can be binary encoded. Every checking is actually a dynamic composition of property models into the behaviour model. The run time manager calculates the result of each composition, making the behaviour model change all the time. Because the resolution process is also represented as LTS model, it will be composed into the behaviour model as a component. Whenever a resolution is done, the solution space for the next time will become smaller. Therefore, the checking itself is an iterative process, where the system is refined each time. As the system executes repeatedly, the system will evolve itself automatically, eventually reaching a point where it will resolve all the feature/aspect interactions itself.

Interleaving plays an important role in our definition of feature/aspect interaction and its resolution. A correct system consists of a set of inevitable operations and their

correct ordering. The majority of interactions that we have experienced have, as their root cause, improper interleaving. In this sense, the complexity results from concurrency, namely the arbitrary interleaving of aspects when they are executed concurrently. Concurrent execution can also be modelled as dynamic interleaving. The potential interleaving of feature operation is a blind area of interaction where the final interleaving is unknown to the developer. This is highly appropriate for the use of run time checking.

In fact, every advice on an aspect constitutes a decision made on synchronisation or interleaving/interaction. Since the decision (can be visualised as a kind of resolution) is made before run time, we can call it static interleaving. Every aspect definition is an attempt to arrange some static interleaving. Thus, aspects themselves are targeted at reducing arbitrary interleaving – aspect programmers consciously put a certain section of aspect code before/after a certain execution point for the enforcement of a restraint relation of operation interleaving.

4. Implementation issues

With the LTS model stored as data, it is necessary to select a proper storage representation. At the moment, we choose adjacent lists, although variations of this structure will be considered as required in the future. Taking this data structure design as foundation, we are able to define analysis algorithms on it. The analysis algorithm is a scaled down version of those used in design time tools, but being revised to be as lightweight as possible. Once the program is represented as data, all the operations on the data are visualised as an operation on the program itself, and the modification towards the model will change the behaviour of the system. As mentioned above, this is a reflection mechanism. Based on it, we can then build adaptive systems for the dynamic integration and resolution of feature/aspect interaction.

The adaptive ability of our system can be represented in two different aspects. Dynamic feature composition is achieved by inserting new feature nodes to the model data structure. Take the "Teenline" feature in telephony as an example. A "PIN"(Personal Identification Number) is required whenever an offhook happens. This feature can be added to the basic call model at run time by inserting certain data nodes to the model data structure. Figure 2 shows sketch of the implementation; the procedures include (a) creating new state nodes (e.g. checking state), (b) adding new behaviour nodes to represent features and modifying the links.

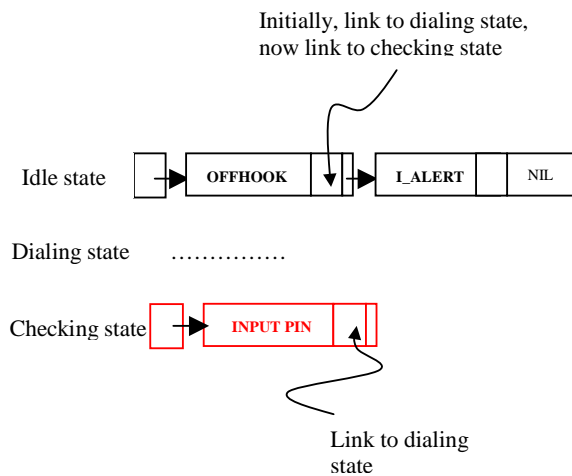


Figure 2. an integration of TeenLine feature

Again, adaptive model checking and resolution is accomplished by recording the detection and resolution and inserting the recorded data to a management structure. AS with the LTS models of features, this concern can also be implemented as aspect. When the checking process recognises a trace, it will “remember” it and next time will not check it again. Also when the resolution corrects a wrong interleaving, it will recognize it in the future.

For example, “Teenline” will interact with basic call model at “OFFHOOK”. After the detection of that interaction, resolution rules are enforced, hence the “ANNOUNCE” will replace the “DIALTONE” in the data structure. Since the replacement is recorded, later

when we withdraw Teenline feature, the initial basic call data node can be restored.

5. Conclusion

All the adaptiveness of our system is achieved by the dynamic addition of behaviour. This kind of addition will accumulate the features (aspects) to a degree where feature interaction becomes a problem. A mechanism in response to this problem is a run time manager, which not only supports feature integration but also feature interaction detection and resolution. This manager permits us to check the feature properties, discovering and resolving feature interactions at run time. Run time model checking is proposed for this purpose, and interaction resolution will be done by selecting behaviour traces according to the resolution rules.

All of the above are supported by a reflective facility, which is powered by building a system that encompasses its LTS model. Because the program is represented as data, program addition and modification become data operation, making the adaptive resolution possible. The reflective mechanism, implemented by storing the program model in a flexible data structure, is a key idea to support the dynamic integration of features, and to provide feature interaction detection and resolution in an auto adaptive manner.

Currently, we have an implementation of the basic call model and a number of features in Java and AspectJ, but the implementation of a more complete version of the framework still needs further work.

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, To appear at the 5th International Conference on Formal Methods for Open Object-Based Distributed Systems, 20-22 March, University of Twente, The Netherlands, Kluwer,2002.
- [3] L.Du Bousquet, “Feature Interaction Detection using Testing and Model-Checking: Experience Report”, In World Congress on Formal Methods, Toulouse, France, Springer, 1999.
- [4] L.Blair. & J.Pang., “Feature Interactions - Life Beyond Traditional Telephony”, In [8], pp 83-93, 2000.
- [5] L. Blair and S. Reiff-Marganiec, Runtime Resolution of Interactions of Multimedia Features. Submitted for Publication, 2001.
- [6] L.G. Bouma and H. Velthuisen, editors, Feature Interactions in Telecommunication Systems. IOS Press (Amsterdam), May 1994..
- [7] K.E. Cheng and T.Ohta, editors, Feature Interactions in Telecommunications Systems III. IOS Press (Amsterdam), October 1995.
- [8] M.Calder, E.Magill , editors, “Feature Interactions in Telecommunications and Software Systems VI”, Glasgow, Scotland, IOS Press(Amsterdam), 2000.
- [9] P.Dini, R. Boutaba, and L. Logrippo, editors. Feature Interactions in Telecommunication Networks IV.IOS Press (Amsterdam), June 1997.
- [10] M. Ryan (Coordinator), “FIREworks: Feature Integration in Requirements Engineering”, Esprit Working Group 23531, started 1997. See www.cs.bham.ac.uk/~mdr/fireworks/casestudies.html.
- [11] R.Hall, “Feature Interactions in Electronic Mail”, In [8], pp 67-82, 2000.
- [12] K.Kimble and L.G.Bouma, editors, Feature Interaction in Telecommunications and Software Systems. Lund, Sweden, IOS Press(Amsterdam), September1998.

- [13] K.Kimbler, H.Velthuisen, "Feature Interaction Benchmark", Discussion paper for the panel on Benchmarking at FIW'95, in [7], 1995.
- [14] J. Magee, J. Kramer, Concurrency-state models & java programs, John Wiley & Sons, 1999.
- [15] 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.
- [16] H. Velthuisen, N. Griffeth, and Y.-J. Lin, editors, International Workshop on Feature Interactions in Telecommunications Software Systems, December 1992. Not in Print.
- [17] L.Bussard, L. Carver, E. Ernst, M. Jung, M. Robillard and A. Speck. "Safe Aspect Composition." Workshop on Aspects and Dimensions of Concern at ECOOP'2000, Cannes, France, June, 2000.
- [18] Aspects in Real-time Embedded Systems (AIRES project), via www.dist-systems.bbn.com/projects/AIRES/. 2000
- [19] M. N. Bouraqadi-Saâdani, R. Douence, T. Ledoux, O. Motelet, M. Südholt, Status of work on AOP at the OCM group, April 2001, École des Mines de Nantes, technical report, no. 01/4/INFO, 2001, available via <http://www.emn.fr/cs/object/biblio/publications.html>
- [20] M.Calder, E. Magill and D.Marples, Hybrid approach to software interworking problems: managing interactions between legacy and evolving telecommunications software, IEE Proceedings - Software, 146(3):167-180, 1999
- [21] D.Marples, S.Tsang, E.H.Magill and D.G.Smith, A Platform for Modelling Feature Interaction Detection and Resolution Techniques, in [7], pp.185-199, 1995.
- [22] M.Cain, Managing Run-Time Interactions Between Call-Processing Features, IEEE Communications, February 1992, pp44-50.
- [23] L.Schessel, Administrable Feature Interaction Concept, ISS'92, October 1992, Vol.2, B6.3, p122-126.
- [24] N.Fritsche, Runtime Resolution of Feature Interactions in Architectures with Separated Call and Feature Control, in [7], pp43-64, 1995.