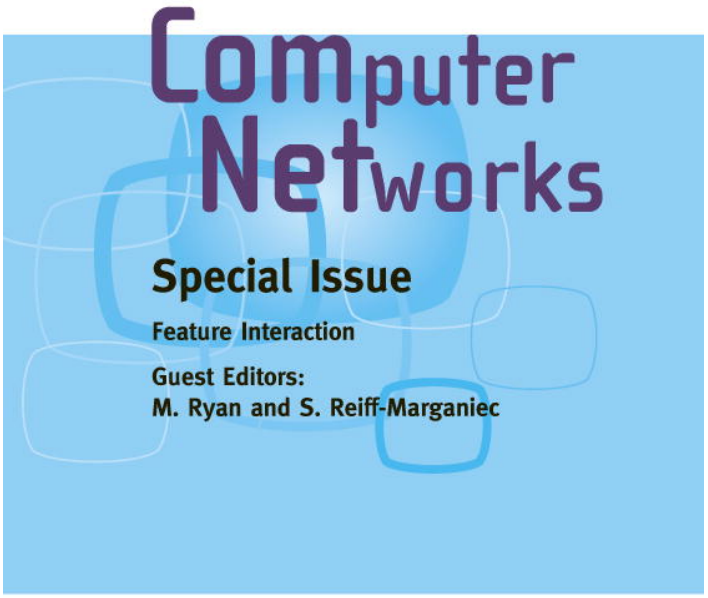
Volume 51 Issue 2 7 February 2007 ISSN 1389-1286



Available online at www.sciencedirect.com
ScienceDirect
Computer Networks 51 (2007) 480–495
Computer Networks
www.elsevier.com/locate/comnet

Program entanglement, feature interaction and the feature language extensions

Wu-Hon F. Leung

Computer Science Department, Illinois Institute of Technology, Chicago, IL 60616, USA

Available online 14 September 2006

Responsible Editor: H. Rudin

Abstract

One of the most difficult tasks in software development is that the programmer must implement a feature going through a laborious and error prone process of modifying the programs of other features. The programs of the different features entangle in the same reusable program units of the programming language, making them also difficult to be verified, maintained and reused. We show that if (C1) the features interact, (C2) they are executed by the same process and (C3) they are implemented in a programming language that requires the programmer to specify execution flows, program entanglement is inevitable and the problem cannot be solved by software design alone. Applications with interacting features are common including those that require exception handling.

The feature language extensions (FLX) is a set of programming language constructs designed to enable the programmer to develop interacting features as separate and reusable program modules even though the features interact. The programmer uses FLX to specify non-procedural program units, organize the program units into reusable features and integrate features into executable feature packages. He develops a feature based on a model instead of the code of other features. FLX supports an automatic procedure to detect the interaction condition among features; the programmer then resolve the interaction in a feature package without changing feature code. FLX features and feature packages are reusable; the programmer may package different combinations of them and resolve their interactions differently to meet different user needs. An FLX to Java compiler has been implemented; our experience of using it has been very positive.

Keywords: Feature interaction; Program entanglement; Programming language; Feature interaction resolution; Reusable programs; Exception handling; Inheritance

1. Introduction

Software development projects often use the term *feature* to denote a development unit. In that context, a feature represents a set of related and testable functionalities of the system. For example, the reliable data transport and the congestion control functions are two different features of the internet transmission control protocol (TCP) [49]. Presently, a feature is often implemented by modifying the code of other features. This is a labor intensive and error prone process. The programmer must go through the code of the other features line by line to determine where to make the changes. At the

E-mail address: leung@iit.edu

1389-1286/$ - see front matter
doi:10.1016/j.comnet.2006.08.002

end, he is often left wondering whether he has correctly identified all the code that needs to be changed and what impact his changes may have on the functionality of the other features. It will require iterations of testing and debugging before the job is considered done. The programmer needs to thoroughly understand and test many times more code than the code needed to implement the new feature.

When a feature is implemented by modifying the programs of other features, the programs of the features are *entangled*: their program statements intertwine in the same reusable program unit (e.g. a method) of the programming language. The entanglement is often *scattered* into many program modules. Entangled and scattered programs are difficult to verify, reuse and maintain.

Most existing efforts to combat the program entanglement problem have focused on software architectural design. Such efforts may succeed initially (and many do not), but as more features are added to the system, entanglement and scattering become severe again. This phenomenon is often called architecture erosion. We will show that for a large class of applications, the program entanglement problem cannot be solved by software architecture design alone.

The problem of program entanglement is related to the notion of *feature interaction*. Two features *interact* if their *behaviors* change when they are *integrated* together. In other words, the behavior of some feature programs cannot be predicted by the input to the programs alone; it also depends whether the feature programs have been integrated with other feature programs. The term feature interaction was first introduced by developers of telecommunications systems [24]. They observed that when the message "`termination request`" comes in and the phone is idle, the programs of the plain old telephone service (POTS) will ring the phone. But if call forwarding is added to POTS, the combined program will give a ping-ring then forwards the call to another phone. The concept is general and not confined to telecommunication software.

Feature interaction is common in embedded systems. Take TCP as an example. Before congestion control was implemented, a duplicated acknowledgement will prompt the reliable data transport feature to retransmit. After congestion control is added, the same message may cause the sender also to retreat to slow start. Applications that desire exception handling encounter feature interaction. Without exception handling, a program running on UNIX will crash when someone hits `control-c`. When exception handling is added, the program does not terminate and may even ask "why are you hitting control-c?" Call forwarding, congestion control and exception handling have been called features, services, concerns or aspects interchangeably in the literature.

Researchers on the subject use the terms behavior and integrate broadly because feature interaction affects all stages of a software development, from the difficulties in recognizing interaction conditions during system specification to the difficulties in testing as it requires the code base to be constantly changing. This paper focuses on design and implementation, although the results reported here have implications on specification and testing. More precise definitions of these two terms within the context of our focus are given in Section 2.

The program entanglement problem relates to feature interaction in the following way: If (C1) two features interact, (C2) they are executed by the same sequential process, and (C3) they are implemented by a programming language that require the programmer to specify execution flows, then the programs of the two interacting features will entangle. If the two features do not interact, their programs do not have to entangle. Feature interaction is a requirement of the application, but it causes the program entanglement problem and the problem cannot be solved by software design alone. Existing general purpose programming languages require C3. Since C1 and C2 are dictated by the application, changing C3 is essential to solving the program entanglement problem.

A solution to the program entanglement problem should allow (R1) the programmer to develop the programs of a feature independent of its interacting features. It should allow (R2) the interacting features to be integrated without changing their programs. (R1) and (R2) imply that the feature programs become reusable. We shall call the conditions under which the behaviors of two interacting features will change their *interaction conditions*. Presently, the programmer must read code to determine when the condition becomes true. Therefore, the solution should enable (R3) a tool to detect the interaction condition among the features. When these requirements are met, adding features will not modify the code of other features and the task of testing becomes easier. The tool that detects feature

interaction conditions will also take the guessing out of specifying them.

We have been developing a set of programming language constructs, called the Feature Language Extensions or FLX, to meet the above requirements. FLX enables non-procedural programming in which the programmer does not specify the execution flows of the program units. A FLX *program unit* consists of a *condition part* and a *program body*. The program body gets executed when its corresponding condition part becomes true. A FLX *feature* is composed of a set of program units that implements the functionalities of a feature. FLX features are integrated into *feature packages*. FLX supports a tool to detect the interaction condition among the features in a feature package where the interaction condition is resolved without requiring changes to the features. Features and feature packages are reusable programs; the programmer may put together different combinations of them in a feature package to meet different user needs. FLX encourages a development paradigm in which the programmer develops a feature following a *model*, which defines the condition space and the basic functionality of the application, instead of examining the code of its interacting features. A FLX to Java [5] compiler has been developed. Using the compiler, an executable FLX program is invoked from a Java program and FLX programs can reuse existing Java programs.

The focus of this paper is to show that FLX meets (R1) and (R2). For (R3), several previous results have shown that the problem of interaction detection is a problem requiring the determination of whether certain assertions on program variables are satisfiable (e.g. [13]). In our case, the assertions are condition parts of program units and they are first order formulas on variables and predicates defined using FLX provided constructs. We elaborate on this briefly in Section 4. The details of the FLX constructs for that purpose and the satisfiability algorithm that we use are described in [63].

FLX takes the position that the software of complex computer applications should be organized into components and FLX is designed for the programmer to develop feature rich components. As an example, FLX is used to develop a telephony system. Each phone object in the system is associated with two feature packages: one for digit collection and analysis (allowing for features like speed calling), and the other for call processing (allowing for features like call waiting). Different phone instances can be associated with different sets of feature packages. Other objects (e.g. GUI) that control the phones are more conventionally coded. Telephony systems are among the most difficult software to develop [9]. Compared to TCP, the control mechanisms in our examples have similar number of states but have more control messages, features and interactions. We have also applied FLX to develop a program that simulates human behavior and are in the process of using FLX to develop a multiplayer networked computer game.

We will discuss the entanglement conditions (C1, C2 and C3) in Section 2. The set of foundation FLX constructs is described in Section 3. They allow the programmer to establish a model, write program units and put them together in a feature. In Section 4, we describe FLX constructs and facilities to resolve interactions among features when they are integrated in a feature package. In Section 5, we give a brief overview of the inheritance and exception handling mechanisms of FLX. Throughout Sections 3–5 we use examples from the telephony system developed using FLX. All the FLX constructs described in this paper have been implemented. We acknowledge related work in Section 6 and conclude the paper in Section 7.

2. Conditions of entanglement

Recall that two features interact if they change the *behavior* of one another when *integrated* together. A feature is implemented by a set of computer programs. The programs of a feature include the program statements that get executed to invoke the feature, those that execute the feature logic and those that terminate the feature. In this context, the *behavior* of a feature is manifested in its output values and execution flow, referred to as the sequence of statements that gets executed, for a given input to its programs. We do not consider two features to be interacting if the programs of one merely change the input to the other but not its behavior.

Given C2, when the programs of two features are *integrated* together and if the programs share variables, it implies that there will be only one instance of the shared variables. As an example, when a TCP duplicated acknowledgement is received, the receiving program will write the relevant content of the message into some variables and there is only one instance of these variables to both the reliable data transport and congestion control features of TCP. The behavior of a program is determined by the

values of its variables at any point in time after the program is invoked and before it terminates. A necessary condition for two programs to interact is that these two programs share variables; otherwise they cannot affect the behavior of one another. In fact, one of them must update some variables used by the other while the other program is still being executed. It follows that the two programs are executed concurrently.

When two programs are executed concurrently by a sequential process as required by C2, then one of the program gets invoked first, but before its execution is completed, some statements of the other program will get invoked. When C3 is true, the only way the programmer can meet this requirement is to change the code of one of the feature programs to insert the code of the other feature program, leading to entanglement. Often, the programmer has to change the programs of both features.

We illustrate the above conclusion with an example. Fig. 1 shows a portion of the finite state machine that controls call origination and termination of the `POTS` feature in our telephony system. We choose to use the well researched State design pattern from the seminal book on the topic [22] to implement finite state machines. The design is shown in Fig. 2. The `Context` class in the figure represents a single interface to the outside world. It offers a method for each event in the finite state machine. Each state in the finite state machine is represented by a class derived from the State abstract class. `Context` has a reference to the current state of the finite state machine which is initially `Idle`. When `Context` receives the `Termination Request` (shorten as TermReq in Fig. 2) event, it calls the corresponding method in

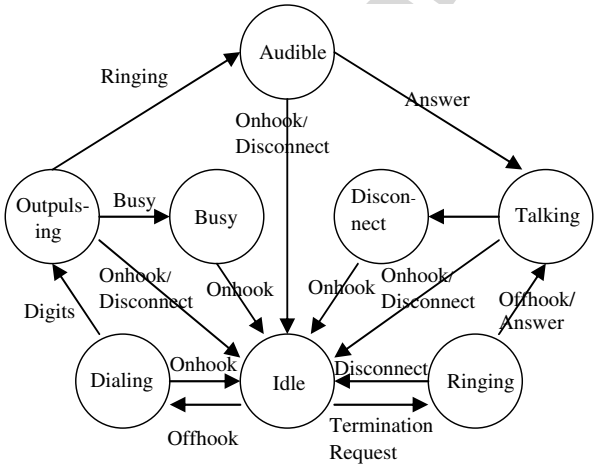


Fig. 1. A finite state machine (partial) of the plain old telephone services.

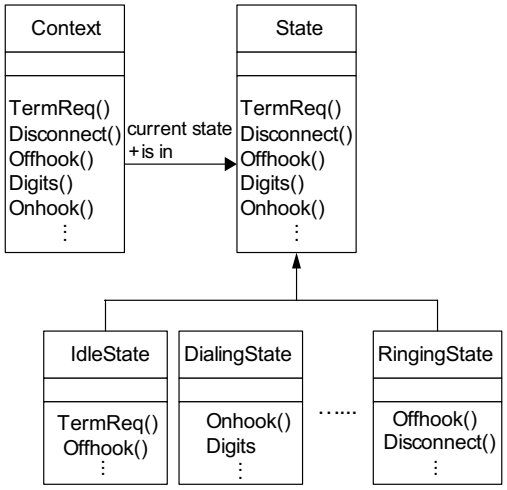


Fig. 2. The design of POTS using the state design pattern.

`IdleState` which will then ring the phone. The State design pattern does not specify where the state change may take place. It can be done either in `Context` or in `IdleState`. But the choice is immaterial for this paper.

Suppose that we want to add the `Call Forwarding Busy Call (CFBC)` feature. `CFBC` responds to the `Termination Request` event by forwarding the call if the phone is in the `Talking`, `Ringing`, `Dialing`, and `Audible` states but not in other states. The only way to add the feature to the design of Fig. 2 will be to insert code to the `TermReq()` methods in state classes corresponding to `Talking`, `Ringing`, `Dialing` and `Audible`. The programs of `CFBC` are therefore entangled and scattered with those of `POTS`. The situation will get exceedingly worse when more features that interact with `POTS` are added to the design of Fig. 2.

The above example should not be considered as a criticism to design patterns which is an important contribution and deserves its coverage in text books on software engineering. Other architectural proposals that we have seen suffer from the same problem.

C1, C2 and C3 are sufficient conditions. If we relax C1, the programs of the two non-interacting features can be executed consecutively without change when integrated. Program entanglement is not necessary. Since we do not presently have automatic means to enforce that the programs of non-interacting features will be written separately, this is where disciplined software design methods, such as object oriented programming, may help.

If we relax C2, the two interacting features will be executed by two different processes. On the surface, their code may not entangle. However, the two

processes will affect the behavior of one another because they exchange information either via a protocol or other means of inter-process communication. In practice, integrating the two processes typically lead to code changes in both processes and their development and verification are even more difficult. A recent article [35] eloquently argues against writing concurrent programs not to exploit natural parallelism in the application but as structuring method. To discuss this topic further will be outside the scope of this paper.

FLX relaxes C3. Features written in FLX are integrated in a feature package that is executed by a sequential process.

We derive the three conditions using a definition of feature interaction specific to feature programs. We use the term *behavior* to denote the computation carried out by a feature program for a given input. Ordinarily, the computation of a program changes when its input changes. But if the program is integrated with programs of its interacting features, its computation may change even when the input does not. When programs of interacting features are integrated, we require that there is only one instance of the variables that the programs share. This requirement reflects what needs to be done when programs are integrated in the same sequential process. We make no assumption on the nature of the feature interaction and what triggers the interaction.

A number of studies have been conducted to establish taxonomy of feature interaction according to the cause or pattern of the interaction (e.g. [52] and [33]). Such classifications are useful to alert the programmer to recognize situations where interaction may occur. The classification given in [33] is event driven and is based on the manners in which the features apply actions to and receive trigger events from devices. While the examples given in this section are event driven, the conditions of entanglement are not. The conditions are neutral to the taxonomy given in [33]; they will predict program entanglement independent of the different classes of the taxonomy. As will be seen later that FLX is event driven, but it allows the programmer to specify program units without triggering events and the interaction condition among FLX programs does not necessarily involve trigger events.

3. The foundation FLX constructs

In this section, we describe the FLX constructs to specify FLX program units, features, and the notion of a model. A FLX model defines the condition space and basic functionalities of an application. The condition space is specified in a *domain statement*. The basic functionality is specified in a special feature called an *anchor feature*. Features are designed following the model and can be thought of as extensions or enhancements to the anchor feature.

3.1. Domain statement

The domain statement contains the definition of the condition variables, called domain variables, and events used in the condition part of a program unit as well as objects shared by all features called resources.

The domain statement for the call processing feature packages in our telephony system is given in Fig. 3. It contains a domain variable `state` that can have a range of values and is initialized to `State.IDLE`. It is declared to be of type `DTenum`. `DTenum` is a *domain data type*, implemented with a simple extension to the class `enum` recently defined in Java 1.5 [29]. The extension is needed for a fast algorithm (often called a first order SAT solver) to determine the satisfiability of first order predicate formulas and is elaborated in [63]. A domain data type must contain public Boolean variables or predicates methods (methods that return Boolean).

A domain variable is instantiated. Memory is allocated to it. On the other hand, an event declared in the domain statement merely says that it is in the model. Multiple instances of the same event may be used by FLX programs using the domain statement.

Some of the events specified in the domain statement of Fig. 3 come from another phone

```
domain BasicTelephony {
       variables:
            DTenum State {DIALING, OUTPULSING,
                    BUSY, AUDIBLE,TALKING,
                    RINGING, DISCONNECT,IDLE};
           State state= State.IDLE; //initial value
  events:
            TerminationRequest;
            Busy;
            Ringing;
            Answer;
            Disconnect;
            Onhook;
            Offhook;
            Digits;
            TimeOut;
  Resources:
            Phone   fone;
            Router  rt;
  }
```

Fig. 3. The domain statement for call processing.

announcing its intent (`TerminationRequest`, `Disconnect`) or its state (`Busy`, `Ringing`, `Answer`) to this phone. Other events are signals (`Onhook`, `Offhook`. `Digits`) coming from the lower level software of this phone. There is also a `Timeout` event generated by the operating system. An event object is of type `event`. It may contain *qualifying variables* that are of domain data type. For example, the event `TerminationRequest` contains the qualifying variable `FromPhoneID` identifying the phone that sends the event.

The domain statement also identifies the objects `fone` and `router` as resources. Features are applied to the `fone` and control messages for call processing are sent to the switch `router`. Resources are declared but not necessarily initialized in the domain statement. They are initialized when a feature package using the domain statement is instantiated.

3.2. Program units and features

Recall that a program unit consists of a condition part and a program body part. In the present implementation of FLX, the program body part is a Java statement. It gets executed when the assertions in its corresponding condition part becomes true.

The condition part of a program unit consists of two statements. The condition part is true (and its corresponding program body will get executed) if both statements are true. The *condition statement* is a first order predicate formula of some data members of domain variables (if they are of type Boolean) and their predicates. FLX does not explicitly support the existential and universal quantifiers. But if the programmer has the need to say something like "there exists some elements" we ask the programmer to specify a predicate method, say non-empty `()`, for the domain data type. To support a fast first order SAT solver described in [63], a domain variable cannot be a function of other domain variables. The compiler checks for that.

The *event statement* of the condition part specifies a list of events and their respective *qualifications*. The qualification of an event is a quantifier free first order formula composed of qualifying variables and their predicates of the event. An event statement is true if an event in the list has been received, its qualification is true and the run time system of FLX chooses to process that event. FLX provides the `enter` and leave pseudo-events for the programmer to specify conditions without an actual triggering event. An event statement specified with `enter` becomes true when its corresponding condition statement becomes true. If an event statement is specified with `leave`, then it becomes true when its corresponding condition statement becomes false. The definitions for these pseudo-events came from an earlier work [23].

```
anchor feature Pots {
     domain BasicTelephony;
     Phone fone;
       Router rt;
MakeCall {
        condition: state.equals(State.IDLE);
        event: Offhook; {
                fone.applyDialTone();
state=  State.DIALING;
        }
}
ReceiveCall {
        condition: state.equals (State.IDLE);
        event: TerminationRequest e; {
state=  State.RINGING;
                }
}
RingPhone       {
        condition: state.equals (State.RINGING);
        event:  enter; {
                fone.applyRinging();
        }
}
RemoveRinging {
        Condition: state.equals (State.RINGING);
        Event:  leave; {
                Fone.removeRinging();
        }
}
}
```

Fig. 4. A portion of the FLX POTS code.

The anchor feature of the call processing feature package, `POTS`, is given in Fig. 4 showing only four of its program units: `MakeCall` applies dial-tone when the user picks up the phone; `ReceiveCall` responds to a `TerminationRequest` event by updating the state of the call to `RINGING` and returning an event to the calling party of that fact. The program unit `RingPhone` rings the phone whenever the state of the phone is `RINGING`; and the program unit `RemoveRinging` removes the ringing when the state of the phone is no longer `RINGING`.

A successfully compiled anchor feature is executable. The instantiation of a feature object is similar to that of a class object as illustrated in the code fragment of Fig. 5. In the example, the POTS anchor feature is instantiated to the object `fp`. The arguments passed to the object will initialize the shared resources in the domain statement. The base class of all executable feature objects includes a number of system methods including the method `SendEvent()` which allows other programs to send an event to a feature object.

```
SomeJavaMethod ( ) {

  // Other code

  // Create phone and switch objects
  Phone thisFone = new Phone (foneID);
  Router thisRouter = new Router
(routerID);

  // Create POTS feature and associate
  // it with thisFone and thisRouter
  Pots fp = new POTS (thisFone,
thisRouter);

  // Some more code

  //User picks up phon
  fp.sendEvent (Offhook);

  // Some other code

}
```

Fig. 5. Code fragment to show invocation of an executable feature object.

Once the domain statement and the anchor feature are defined, the programmer uses them as the basis to design additional features. We show two features, `DoNotDisturb` and `CallForwarding` (showing only the essential program unit for each) in Figs. 6 and 7 respectively. `DoNotDisturb` sends back a Busy message to the caller whenever the phone is called. The keyword `all` used in the condition statement of `SayBusy` specifies that the condition statement is true no matter what values

```
feature DoNotDisturb {
domain BasicTelephony;
anchor POTS;
Router rt;

SayBusy {
condition: all;
        event: TerminationRequest e; {
           Busy b = new Busy();
           rt.sendEvent (Event.FromPhoneID,
b);
           }
       }
}
```

Fig. 6. The feature `DoNotDisturb`.

```
 feature CallForwarding {
domain _BasicTelephony;
anchor __POTS;
Router rt;
String forwardNumber;

ForwardCall {
     condition:state.equals(State.IDLE);
     event: TerminationRequest e; {
        rt.sendEvent (forwardNumber, e);
        }
     }
}
```

Fig. 7. A portion of the `CallForwarding` feature.

the domain variables have. `CallForwarding` forwards the `TerminationRequest` to another phone if it receives the message when the phone is `IDLE`. The forwarding is done by relaying the `TerminationRequest` message to another phone if the call forwarding number has been specified (not empty) and the call forwarding number is not the same as the caller (identified by the `FromPID` field of the `TerminationRequest` message `e`).

These two features are quite easy to write using FLX. If they were implemented with a procedural language using the state design pattern of Fig. 2, one will not be able to put the programs of these features in a single module and they will entangle and scatter among the many programs of POTS. The author was a developer in a telephony switching system. In that system, the code for `DoNotDisturb` was inserted hundreds of times into all the program modules in which the message `TerminationRequest` may be handled.

A feature references the domain statement and anchor feature that it is based on so that the compiler can perform a number of semantic analyses such as that the condition statement of at least one of the program unit of an anchor feature is satisfiable given the initial values of the domain variables.

4. Feature interaction resolution and feature packages

The features `POTS`, `DoNotDisturb` and `CallForwarding` interact with each other. We will show how to put them together into a feature package in this section. Before we do that we discuss briefly how we know that they interact.

4.1. Interaction detection

An instantiated feature package is executed by a sequential process. The domain variables of the feature package are accessible by the features of the feature package and nowhere else. Variables declared within features and feature packages are accessible only locally. The run time system for FLX programs satisfies these two properties:

Property 1: It chooses only one event at a time to evaluate whether the condition part of some program units has become true. Once it finds such a program unit, the event is consumed and execution of the program unit begins.

Property 2: Execution of the program body of a program unit is not interrupted because the values of some domain variables have been changed and some events have been received during the execution.

Given these two properties, one can show that if the conjunction of the condition parts of two program units is satisfiable, the two program units interact and the satisfiable condition is their interaction condition. When the interaction condition becomes true, either program units may get executed. FLX requires the programmer to remove, or resolve, the ambiguity for the run time system. When a feature is compiled, interaction among its program units, if any, will have been resolved.

In FLX, two features interact if they are integrated in the same feature package and one contains a program unit that interacts with some program units in the other. FLX requires the programmer to resolve the interaction in the feature package so that the features themselves need not be modified. The mechanisms provided by FLX to resolve interaction between program units within a feature or between features within a feature package are similar. We will only give examples in resolving feature interaction.

A number of researchers had recognized that feature interaction detection is a satisfiability problem. While we deal with different logical systems (sometimes only slightly different), the derivation to the conclusion is similar. Several of these researchers' work are referenced in Section 6.

4.2. Interaction resolution with precedence list in a feature package

The programmer uses a feature package to integrate a set of features together. As a result, it contains a statement specifying the list of domains used by the features in the package. The anchor features used by other features are included in the list of features.

Fig. 8 shows the code of the feature package `QuietPhone` integrating the features `POTS` and `DoNotDistrub`. The two features interact when the `TerminationRequest` message is received. POTS will ring the phone while `DoNotDisturb` will return `Busy` to the caller. The interactions among the two features are resolved by the precedence statement `priorityPrecedence` which specifies that if a condition becomes true for some program units in the list of features at the same time, the program unit belonging to the highest priority feature executes. Consequently when the phone that uses `QuietPhone` receives a `TerminationRequest` message, only the program unit `SayBusy` of `DoNotDisturb` will be executed. But when the phone receives an `OffHook` event and the phone is `IDLE`, then the `MakeCall` program unit of `POTS` gets invoked and the user can make phone calls. This simple example shows that the two interacting features can be integrated together without changing each other's code.

```
feature package QuietPhone {
    domain: BasicTelephony;
    Phone fone;
    Router rt;
    features: DoNotDisturb, POTS;
    priorityPrecedence (DoNotDisturb, POTS);

}
```

Fig. 8. The `QuietPhone` feature package.

FLX currently supports another type of precedence list called `straightPrecedence`. When the condition parts become true for program units from different features in the `straightPredence` list, all these program units will get executed following the order specified in the precedence list. Other types of precedence lists are possible as long as they define a partial ordering of the features and hence resolve interaction condition. FLX allows multiple precedence lists in a feature package, and the compiler checks that they do not result in contradictions. One can show that `priorityPrecedence` and `straightPrecedence` are primitive: they can be used to implement an arbitrary partial ordering that does not contain contradictions [51].

4.3. Interaction resolution with program units

We shall use another example to illustrate two points. First, feature packages with the same list of features may have different functionalities depending on how the interaction is resolved. Secondly, the programmer may use program units in a feature package to gain finer control of interaction resolution on any specific condition.

Suppose that we want to integrate `DoNotDisturb`, and `CallForwarding` onto `POTS`. `CallForwarding` interacts with the other two features when the phone is called while `IDLE`. If we use precedence lists only and put `DoNotDisturb` ahead of `CallForwarding`, no call will get forwarded. If we reverse the precedence, then all calls will be

```
feature package SelectiveForwarding1 {
        domain BasicTelephony;
        features DoNotDisturb, CallForwarding(rt), Pots;
        priorityPrecedence (DoNotDisturb, CallForwarding, Pots);

        LinkedList phoneIDlist=  new LinkedList (empty); // forwardable phones


        SelectToForward {
                condition: state.equals (State.IDLE);
                event: TerminationRequest e; {
                    if (phoneIDlist.contains (TerminationRequest.FromPhoneID))
                        CallForwarding;
                        /* The program unit in CallForwarding that satisfies the
                        condition part is invoked.*/
                    else
                        DoNotDisturb;
                    stop;
        }
}
 }
```

Fig. 9. The `SelectiveForwarding` feature package.

forwarded while the phone is `IDLE`. The user may not like either alternative.

The `SelectToForward` program unit of the `SelectiveForwarding` feature package given in Fig. 9 resolves the interaction condition that a call comes in when the phone is idle. It forwards the call if the caller belongs in a `phoneIDlist`, otherwise the call is blocked. By convention, a program unit in a feature package has the highest precedence. The stop statement at the end of `SelectToForward` instructs the compiler to stop executing any other program units whose condition has also become true. `SelectToForward` does not explicitly call the program units `ForwardCall` of `CallForwarding` and `SayBusy` of `DoNotDisturb`. Instead, it refers to the features. The FLX compiler generates code to invoke the correct program units. Alternatively, the programmer can explicitly call `CallForwarding.ForwardCall` and `DoNotDisturb.SayBusy`. The compiler then checks whether the program units are called "within context".

5. Exception handling and inheritance

Exceptions handling and inheritance mechanisms are both important topics in programming language design. The programmer uses exception handling mechanisms to specify what should happen after an exception event, usually due to error conditions, has occurred; and he uses inheritance mechanisms to extend an existing class with additional data and methods. For applications with fault tolerance (or robustness) requirements, a significant portion, sometimes a majority of their programs are devoted to exception handling. Large scale software tends to evolve over time and inheritance, when it can be applied, is an elegant solution to deal with change. Both topics continue to be actively researched.

We give an introduction to these mechanisms in FLX in this section with emphasis on exception handling. The issues with exception handling both in the applications domain and in programming language support are complex. It is not surprising that researchers have observed that exception handling programs are more likely to contain software bugs than any other part of the software [15]. We focus on two issues here: the observations that existing mechanisms do not encourage writing reusable exception handling code [36], and that they do not handle change in exception handling definition and policy well: a change often ripples through large amount of code.

5.1. Exception handling programs in FLX are features

To encourage reuse, FLX provides constructs for the programmer to organize exception handling code into *exception features*. FLX requires that program units that are triggered by an exception event to be contained in an exception feature, but not all program units in an exception feature must be triggered by an exception event. We show the reusable exception feature `DamageControl` in Fig. 10. The feature guards against illegal conditions, such as when the phone is `IDLE` but the feature package receives an `Onhook` event or when a hardware error event `RingCktBrokenException` is received. The condition statement of a program unit triggered

```
exception feature DamageControl {
     domain: BasicTelephonyWithExceptions;
     anchor: POTS;

     IllegalOnhook {
        condition: state.equals (State.IDLE);
        event: Onhook; {
             System.out.println ("Illegal Onhook");
             fone.Disable();
             stop;
         }
     }

    BrokenRingCKT {
        context: {
             condition: state.equals (State.IDLE);
             event: enter;
             }
        exception: RingCKTBrokenException; {
             System.out.println ("Ring CKT broken");
             stop;
        }
    }

    //Other program units not shown
}
```

Fig. 10. The DamageControl exception feature (partial).

by an exception is called `context`, which identifies the condition under which the exception was thrown. In the program unit `BrokenRingCkt`, the `context` is when the feature package enters the `RINGING` state. Alternatively, the programmer can identify a program unit in the condition statement of the context. The context statement in BrokenRingCkt can be written as "`context: {POTS.RingPhone()}`".

The `CatchAll` exception feature shown in Fig. 11 catches any condition not anticipated by other program units in the feature package when it is placed at the bottom of the precedence lists in a feature package. `CatchAll` is a very useful feature especially during the testing and debugging phase of development. We use it in every feature package that we have developed.

The feature package shown in Fig. 12 integrates these two exception features with the `Selective-Forwarding` feature package of Fig. 9. Exceptions that are not caught in a feature package are thrown to the Java program that instantiates the feature package.

5.2. Inheritance and exception features that enhances portability

The call processing feature package of our telephony prototype can be executed either in a central switch or in a smart phone. In the former case, the designer may decide to share a few ringing circuits for all the phones controlled by the switch. The method `fone.applyRinging()` will then throw a `NoMoreRingCktException` that is not present if the feature package is executed in the smart phone. Importantly, the exception handling policy will be different. In the switch, one can take advantage of the existence of multiple hardware units to

```
exception feature CatchAll {
        domain: BasicTelephonyWithExceptions;
        anchor: POTS;

        CatchNormal {
            condition: all;
            event: any; {
                   System.out.println ("CatchNormal: Unexpected condition and event");
                   this.dump (domain, event);
                   }
        }

        CatchException {
            context: {all};
            exception: any; {
                   System.out.println ("CatchException: Unexpected exception");
                   this.dump (domain, event);
                   }
        }
}
```

Fig. 11. The `CatchAll` exception feature.

```
feature package RobustSelectiveForwarding {
       domain: BasicTelephonyWithExceptions;
       features: SelectiveForwarding, DamageControl, CatchAll;
       priorityPrecedence (DamageControl, SelectiveForwarding, CatchAll);
}
```

Fig. 12. The `RobustSelectiveForwarding` feature package.

```
domain BaiscTelephonyWithExceptions extends BasicTelephony {
        exceptions:
                RingCKTBrokenException;
                ConfCKTBrokenException;
                // and others
}
```

Fig. 13. The BasicTelephonyWithExceptions domain statement.

```
domain SwitchTelephony extends BasicTelephonyWithExceptions {
        exceptions:
                NoMoreRingCKTException;
                NoMoreConfCKTException;
                // and others
}
```

Fig. 14. The SwitchTelephony domain statement.

```
exception feature Retry {
        domain: SwitchTelephony;
        anchor: POTS;

        RetryRinging {
                context: {
                        condition: state.equals (State.RINGING);
                        event: any;
                };
                exception: NoMoreRingCKTException e; {
                        fone.repair (e);
                        Thread.sleep (500);
                }
        }
}
```

Fig. 15. The `Retry` exception feature.

make the system more fault-tolerant. Using a procedural language, the programmer will have to change code to handle the new exceptions and exception handling policy. The complexity of doing so typically leads to the development of two different versions of the software.

With FLX, the different exception handling policies become different exception features that the programmer chooses to include in a feature package. The programmer can use inheritance to accommodate the additional exception events of the switch. The `BasicTelephonyWithExceptions` domain statement shown in Fig. 13 is extended from the `BasicTelephony` domain of Fig. 3 with common exception events that will be needed no matter where the application will be executed. The example in Fig. 14 extends from the one in Fig. 13 to add exceptions that will be needed only if the application is executed in the switch. The two exceptions shown in Fig. 14 indicate no ringing circuit and no conference circuit is available. These two exceptions make no sense in a smart phone.

When extending a domain statement, the programmer may add new domain variables, events, exceptions and resources to an existing one. The original anchor feature and features written on the original model may be integrated with features using the new domain statement. We show a simplified Retry exception feature in Fig. 15. The program unit `RetryRinging` calls the `fone.repair()` method then waits for half a second when notified that there is no ringing circuit for the time being. When `Retry` is put ahead of `POTS` in a `straightPrecedence` list of a feature package, it will have the effect of pausing for half a second and then try ringing again. The `Retry` feature will not be included in the feature package if it is to be executed on a smart phone.

6. Related works

The heritage of FLX comes from AI languages that support non-procedural execution. What we called feature interaction is *conflict* in rule-based languages. Because of the differences in intended usage, the structure and facilities provided by AI languages are quite different from FLX. AI languages do not require explicit conflict detection

and resolution before programs can be executed. Programmers using them typically do not know beforehand what program units may interact with each other and their ability to resolve conflicts is limited. Jackson [28] provides a very good coverage on the most important AI languages.

FLX supports event driven programming: the condition part of a FLX program unit is usually triggered by an event. Event driven programming is supported by the most popular programming languages such as Java, Visual C++ and C# as many applications require it. The most important server side platforms such as Enterprise JavaBean [39], Microsoft. NET [41] and the newly defined Spring Framework [54] provide event handling API's. Each of them tries to handle asynchronous and unpredictable events within the framework of a procedural language, requiring programming concepts, such as the non-reusable "inner class" and "anonymous class" of Java. A nonprocedural approach is more natural.

Aspect oriented programming (AOP) [17] is arguably the most influential programming language concept proposed in recent years in terms of the number of papers that have been written about it and cited it. AOP has made important contributions to the understanding of the problem of program entanglement. FLX supports the AOP goal of separation of concern. The fundamental difference between FLX and the main stream AOP as embodied in AspectJ [32] is that AOP applies to base code written in a conventional programming language with the obvious advantage that it can be applied to legacy code.

In AOP, the programs of a feature that would have been scattered throughout the base code are put into a module called an `aspect`. An `aspect` contains a set of `pointcuts` and `advices`. A `pointcut` identifies some statements in the base code and its corresponding `advice` instructs code to be inserted `before`, `after` or `around` those statements through a process called *weaving*. AspectJ does not have a separate facility to combine aspects. An aspect actually instructs the compiler to modify the base code but the modification does not appear in the base code. Some argue that the transparent modification of other programs is detrimental [2]: An aspect cannot be understood in isolation. Unless we limit its usefulness, an aspect is fragile and has to be changed when the base code changes and vice versa. Since a `pointcut` can refer to very specific program artifacts such as the name of a variable or a method, an `aspect` is tightly coupled with its base program and in general is not reusable.

Researchers of AOP have been working to overcome the above problems. There are two basic approaches: (1) write aspects without knowledge of the base program and then later connect the aspects to the base program (e.g. [56]), or (2) limit the aspect's access to the artifacts of the base program unless it is specifically allowed (e.g. [1,44]). In general, the task of finding the connection in the first approach will require examining and modifying base and aspect code. The second approach will constrain the usefulness of the aspects. Without constraints, aspects can be very powerful as those given in [14] that would modify queues managed by an operating system kernel but they are also very risky.

A number of empirical studies (e.g. [36]) have shown that AOP can reduce the amount of code written quite substantially. But that does not necessary translate to programmer productivity. An earlier study [42] to measure whether AOP improves programmer productivity was inconclusive. It noted that programmers often needed to "restructure the base code to expose suitable join points." This problem has not been improved upon over the years [19]. Another recent empirical study to investigate whether AOP is beneficial in the implementation of cross cutting (or interacting) design patterns is quite critical of the AOP approach [12].

There are other AOP approaches. The multidimensional concerns approach [45] requires the programmer to extensively review existing code to support multiple concerns, although there is language help for the restructuring effort. The composition filters approach [7] has its root from the `UNIX pipe` [10] and `streams` [53] mechanisms. Again, it requires programmer effort to find the programs in the base to forward information to the filters. Similar to the mainstream AOP, these approaches also do not meet the requirements R1 and R2 put forth in the introduction section of this paper.

FLX belongs to the family of programming languages that provides language constructs to help specify systems that continuously respond to external stimuli to produce outputs. Among the languages in the family, Statechart [25], LOTOS ([57,38]), SDL [16] and Esterel [8] are the most important. A good description and discussion of the last three languages can also be found in [58]. Statechart is a component of the industry standard unified modeling language (UML) [61]. LOTOS

and SDL are both international standards. Esterel and its associated languages and tools have been developed for over two decades. All of them have attracted substantial research and real products have been developed using them.

These languages take different approaches. Statechart provides a structured method to represent finite state machines graphically. Esterel is a synchronous language with the semantic that the execution of a reaction to an input event is instantaneous (or uninterruptible). LOTOS is algebraic (or symbolic) inducing the programmer to use refinement. SDL is more concrete and is defined with most of the artifacts of a general purpose programming language. All of them either directly support the definition of finite state machines or generate one, thus programs written in them are also directly amenable for automatic analysis.

These languages were pioneers. Some of the criticisms on them can now be fixed: that the synchronous semantics of Esterel may lead to "causality cycles" (an emitted signal leads to the instant generation of the same signal) [11], that LOTOS lacks modularity constructs [3], and that the state condition in SDL is the name of a state instead of a more general condition expression. The most important lesson comes from an empirical study of using LOTOS in a telecommunication system: "we have to rewrite POTS several times as features were added," and "how could reusable (sub)specifications be declared and used?" [3]. These comments apply to the other three languages as well.

There were subsequent improvements to these languages (e.g. [11,46,55]) but the above two issues were not addressed. A number of programming languages were designed specifically for telecommunication software. VFSM [21] allows the programmer to specify finite state machines with non-procedural program units but it is less developed compared to SDL. VoiceXML [62] and the call processing language (CPL) [34] are both markup languages. Adding features in these languages requires changes to the existing features.

CRESS [59] is the first graphical language that explicitly supports the specification and combination of features. It is a substantial work that also includes tools to check for the correctness of programs written in CRESS and to translate them into various formal languages. A feature written in CRESS is reusable in the sense that it can be invoked multiple times by another feature; but combining features is done by "splicing" and "inserting" features to one another requiring the programmer to manually determine the insertion points.

Among the many contributions from the FIREworks project [20], the results of Plath and Ryan ([47] and [48]) are most relevant to our work. They extended the input languages of the model checkers SMV [40] and SPIN [26] to allow specification of features extending from a base system. Since the input languages of the model checkers are largely non-procedural, their results on feature specification are very similar to ours. They already have the concepts of anchor features and features; we provide additional facilities for the integration of features. In their approach, feature specifications are directly translated into input to a model checker and are readily verified; our feature specifications are translated into Java (or other implementation languages) and can readily integrate with the rest of the implementation software. Their result will influence our next phrase of work to verify FLX programs even more.

The term "Feature (or Service) Oriented Programming (or Architecture)" has recently become fashionable. Prehofer first coined the term when he introduced language extensions on Java to specify features [50]. His extensions are procedural programming extensions and require the programmer to resolve the interaction between every pairs of features often by changing existing programs. Batory and his group proposed an elegant mathematical model for composing features (e.g. feature interactions are derivatives as in calculus) ([37], [6]). But in practice, they require the programmer to have thorough understanding of the feature programs and to integrate features by changing code in a manner similar to the method of Prehofer.

FLX supports automatic detection of feature interaction. The topic was studied quite extensively using temporal logics, such as Linear Temporal Logic (LTL) in [18] and in Computation Tree Logic (CTL) in [31], as well as using Petri Nets [43] and other formalism. The problem is a satisfiability problem and a number of researchers use existing model checkers to solve the problem (e.g. [4,13,48] and [60]). Our contribution to the topic is to use programming language facilities to assist a fast algorithm to determine the satisfiability of first order formulas [63]. Most existing results on the topic were concerned with determining interaction on the specification of the features; we are concerned with determining interaction on the implementation of the features.

A large amount of work has been done to minimize program entanglement in the presence of feature interaction by software architectural design. Many innovative ideas have been put forth, among them the distributed feature composition [27], a `pipe-and-filter` like architecture, and the feature interaction manager architecture [30]. We benefited from the observation in [30] and others that the interaction among some features can be resolved by arranging the execution of the features according to priority.

We are indebted to the above and many other results. The special strength of FLX is that it enables the programmers to develop interacting features as independent and reusable program modules, so that features can be added without changing existing code.

7. Conclusions

FLX has two design objectives: (1) to enable the development of interacting features as separate and reusable program modules, and (2) to facilitate the automatic verification of programs written in FLX. From our experience so far, FLX meets the first design objective and has a positive impact on programmer productivity. We had implemented a fast first order SAT solver and the FLX compiler generates a finite state machine from an executable FLX program, therefore programs written in FLX are amenable for automatic analysis. But we have not yet developed the verification tool and objective (2) has not been accomplished.

About thirty different features and feature packages were written in FLX for the telephony system described earlier. Most of the features were developed by two graduate students over a period of about nine months when they also have to take classes and work on the compiler. `POTS` were developed with 195 lines of FLX code. The student who wrote `POTS` did not need to be concerned with call waiting and other features, nor `Retry` and other exception handling policies. The student who wrote call waiting only need to become familiar with `POTS` but not other features that call waiting also interacts with. Integration of features in feature packages was not a problem and usually accomplished in a couple of days. Interaction conditions are automatically detected and most are resolved by precedence lists. We did not encounter a single case when the implementation of a new feature requires changes to existing features.

When a feature is developed, it is tested with its anchor feature first. When well-tested features are integrated in a feature package, the testing and debugging can focus on the feature package. There was only one instance when the testing of a feature package revealed an error in a feature and the error was fixed with changes to the programs of that feature only.

The generated code of a FLX program looks a lot like how one may write the program in Java. We therefore suggest that the performance of FLX programs will be comparable to those written in Java. The FLX code written for the prototype is several times less than its generated code. This is partly due to the non-procedural nature of the language and partly due to short hands, such as the keywords `all` and `any`, supported by the language. Consider the `DoNotDisturb` feature given in Fig. 6. Its code will have to be duplicated many times if the features are written in a procedural language using the design pattern given in Fig. 2.

A more rigorous and extensive study to evaluate FLX is planned. Another focus of our future work will be to develop a verification tool for programs written in FLX.

Acknowledgements

The author wishes to acknowledge the contributions of Manohar Dudda, Benson Fung, Yimeng Li, Karthik Ramachandran, Jasmine Vadgaama and Lu Zhao. Some of their contributions are overviewed in this paper. The author also wishes to express his appreciation to the editors of this special issue and the reviewers of this paper. Their efforts had helped to make this paper better.

References

[1] J. Aldrich, Open modules: modular reasoning about advice, in: Proceedings of ECOOP, 2005.
[2] R. Alexander, The real costs of aspect-oriented programming, IEEE Software (November/December) (2003).
[3] M.A. Ardis, Lessons from using basic LOTOS, in: Proceedings of the International Conference on Software Engineering, May 1994.
[4] C. Areces, W. Bouma, M. de Rijke, Feature interaction as a satisfiability problem, in: Feature Interaction in Telecommunication and Software Systems, 2000.
[5] K. Arnold, J. Gosling, D. Holmes, The Java Programming Language, third ed., Addison-Wesley, 2000.
[6] D. Batory, J.N. Sarvela, A. Rauschmayer, Scaling step-wise refinement, in: Proceedings of International Conference on

Software Engineering 2003 (ICSE 2003), Portland, Oregon, May 2003.
[7] L. Bergmans, M. Aksit, Composing crosscutting concerns using composition filters, Communications of the ACM 44 (10) (2001).
[8] G. Berry, G. Gonthier, The ESTEREL synchronous programming language: design, semantics, implementation, Science of Computer Programming 19 (1992) 87–152.
[9] F.P. Brooks Jr., The Mythical Man-Month, third ed., Addison Wesley, 1995 (Chapter 8).
[10] S.R. Bourne, The UNIX shell, The Bell System Technical Journal 57 (6, part 2) (1978).
[11] F. Boussinot, Reactive C: an extension of C to program reactive systems, Software – Practice and Experience (April) (1991).
[12] N. Cacho et al., Composing design patterns: a scalability study of aspect-oriented programming, in: Proceedings of AOSD'06, March 2006.
[13] M. Calder, A. Miller, Using SPIN for feature interaction analysis – a case study, in: Proceedings of SPIN2001, 2001.
[14] Y. Coady, G. Kiczales, Back to the future: a retroactive study of aspect evolution in operating system code, in: Proceedings Aspect-Oriented Software Development, 2003.
[15] F. Cristian, Exception handling and tolerance of software faults, in: M. Lyu (Ed.), Software Fault Tolerance, 1995, pp. 81–107.
[16] J. Ellsberger, D. Hogrefe, A. Sarma, SDL Formal Object-oriented Language for Communicating Systems, Prentice Hall Europe, Hemel Hempstead, 1997.
[17] T. Elrad, R.E. Eilman, A. Bader, Aspect-oriented programming, Communications of the ACM 44 (10) (2001).
[18] A.P. Felty, K.S. Namjoshi, Feature Specification and Automated Conflict Detection, Feature Interaction in Telecommunications and Software Systems VI, IOS Press, 2000.
[19] F. Filho, C. Rubira, A. Garcia, A quantitative study on the aspectization of exception handling, in: Proceedings of ECOOP Workshop on Exception Handling in OO Systems, July, 2005.
[20] Available from: <http://www.cs.bham.ac.uk/~mdr/firworks/>.
[21] A.R. Flora-Holmquist, E. Morton, J.D. O'Grady, The virtual finite-state machine design and implementation paradigm, Bell Labs Techincal Journal (1997).
[22] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.
[23] T.L. Hanson, W.E. Hyatt, W.H. Leung, W.E. Montgomery, State control for a real-time system utilizing a non-procedural language, US patent 4727575, February 23, 1988.
[24] J.A. Harr, E.S. Hoover, R.B. Smith, Organization of the No. 1 ESS Stored Program, The Bell System Technical Journal (1964).
[25] D.e.a. Harel, Statemate: a working environment for the development of complex reactive systems, IEEE Transactions on Software Engineering 16 (4) (1990).
[26] G.J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley Professional, 2003, September 4.
[27] M. Jackson, P. Zave, Distributed feature composition: a virtual architecture for telecommunication services, IEEE Transactions on Software Engineering 24 (10) (1998).
[28] P. Jackson, Expert Systems Harlow, Addison Wesley Longman Ltd, England, 1999.
[29] New Features and Enhancements, J2SE 5.0, Available from: <http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html>.
[30] Y. Jia, J.M. Atlee, Run-time management of feature interactions, in: ICSE Workshop on Component Based Software Engineering (CBSE6), May 2003.
[31] J. Kamoun, L. Logrippo, Goal-oriented feature interaction detection in the intelligent network model, in: Feature Interaction Workshop, 1998.
[32] G. Kiczales et al., An overview of aspectJ, in: Journal European Conference on Object Oriented Programming (ECOOP 2001), Springer-verlag, 2001.
[33] M. Kolberg et al., Compatibility issues between services supporting network appliances, IEEE Communications Magazine (November) (2003).
[34] J. Lennox, X. Wu, H. Schulzrinne, Call Processing Language (CPL): A Language for User Control of Internet Telephony Services, IETF RFC 3880, October 2004.
[35] E.A. Lee, The problem with threads, Computer (May) (2006).
[36] M. Lippert, C.V. Lopes, A study on exception detection and handling using aspect-oriented programming, in: Proceedings of International Conference on Software Engineering, ICSE 2000.
[37] J. Liu, D. Batory, S. Nedunuri, Modeling interactions in feature oriented software design, in: Proceedings of Feature Interactions in Telecommunications and Software Systems,VIII, June 2005.
[38] L. Logrippo, M. Faci, M. Haj-Hussein, An introduction to LOTOS: learning by examples, Computer Networks and ISDN Systems 23 (5) (1992).
[39] V. Matena, S. Krishnan, L. DeMichiel, B. Steams, Applying Enterprise JavaBeans, Addison-Wesley Books, 2003, May.
[40] McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.
[41] <http://www.mircosoft/Net/>.
[42] G.C. Murphy, R.J. Walker, L.A. Baniassad, Evaluating emerging software development technologies: lessons learned from assessing aspect-oriented programming, IEEE Transactions on Software Engineering 25 (4) (1999).
[43] M. Nakamura, Y. Kakuda, T. Kikuno, Petri-net based detection method for non-deterministic feature interactions and its experimental evaluation, in: Feature Interaction in Telecommunications Systems, IOS Press, Amsterdam, 1997.
[44] N. Ongkingco et al., Adding open modules to aspectj, in: Proceedings of the 5th International Conference on AOSD, March 2006.
[45] H. Ossher, P. Tarr, Using multidimensional separation of concerns to (Re)shape evolving software, Communications of the ACM 44 (10) (2001).
[46] C. Passerone et al., Modeling reactive systems in Java, ACM Transactions on Design Automation of Electronic Systems (October) (1998).
[47] M. Plath, M.D. Ryan, A feature construct for Promela, in SPIN'98, in: Proceedings of the 4th SPIN Workshop, November 1998.
[48] M. Plath, M.D. Ryan, Feature integration using a feature construct, Science of Computer Programming (January) (2001).
[49] J. Postel, Transmission Control Protocol, RFC 793, September 1981, Available from: <http://www.refc-editor.org/rfc793.txt>.

[50] C. Prehofer, An object oriented approach to feature interaction, in: Proceedings of the Feature Interaction Workshop, 1997, IOS Press.
[51] K. Ramachandran, Exception Handling in the Feature Language Extensions, M.S. Thesis, ECE Department, Illinois Institute of Technology, June, 2005.
[52] S. Reiff-Marganiec, K.J. Turner, Feature interaction in policies, Computer Networks (August) (2004).
[53] D.M. Ritchie, A stream input–output system, AT& T Bell Laboratories Technical Journal 63 (8 Part 2) (1984).
[54] <http://www.springframework.org/>.
[55] V.C. Sreedhar, M.-C. Marinescu, From Statecharts to ESP*: programming with events, states and predicates for embedded systems, in: Proceedings of Embedded Software, EMSOFT'05, September 2005.
[56] D. Suvee, W. Vanderperren, V. Jonckers, JAsCo: an aspect-oriented approach tailored for component based software development, in: Proceedings of Aspect-Oriented Software Development (AOSD 2003), 2003.
[57] K.J. Turner, A LOTOS-based development strategy, Formal Description Techniques II (1990) 117–132.
[58] K.J. Turner (Ed.), Using Formal Description Techniques – An Introduction to Estelle, LOTOS and SDL, Wiley, 1993.
[59] K.J. Turner, Modular feature specification, Proceedings of MICON (August) (2001).
[60] T. Tsuchiya, M. Nakamura, T. Kikuno, Detecting feature interaction in telecommunication services with a SAT solver, in: IEEE Pacific Rim International Symposium on Dependable Computing, 2002.
[61] Unified Modeling Language, Version 1.5, Available from: <http://www.omg.org/technology/documents/formal/uml.htm>.
[62] www Consortium, "Voice Browser Activity," 2005, Available from: <http://www.w3.org/Voice>.
[63] L. Zhao, Y. Li, W.H. Leung, A first order satisfiability solver for the feature language extensions, submitted for publication and it is available from: <http://www.openflx.org>.

Wu-Hon Francis Leung received his M.S. and Ph.D. degrees in Computer Science from the University of California, Berkeley. He is an IEEE Fellow, cited for his contributions to operating systems, protocols and programming methods supporting the development of distributed systems and multimedia communication applications.

For more than twenty years, he was a developer, researcher and manager at Bell Laboratories and Motorola and worked on projects in telecommunication systems software, broadband protocols and applications, and cellular systems architecture. Earlier in his career, he served as an editor of the IEEE Transactions on Computers for a number of years. Presently, he is on the faculty of the Computer Science Department of the Illinois Institute of Technology. His research interests now center on software engineering and computer network protocols.