Resolving Feature Interaction with Precedence Lists in the Feature Language Extensions

L. Yang, A. Chavan, K. Ramachandran, W. H. Leung[1]
Computer Science Department
Illinois Institute of Technology, Chicago, IL 60616

Abstract. With existing general purpose programming languages, interacting features executed in the same process must be implemented by changing the code of one another [1]. The Feature Language Extensions (FLX) is a set of programming language constructs that enables the programmer to develop interacting features as separate and reusable program modules. Features are integrated and have their interactions resolved in feature packages. FLX provides the *precedence list* facilities for the programmer to specify the execution order of the features in a feature package. While not applicable in all situations, precedence lists can be used to resolve many interaction conditions in a single statement. This paper describes the two types of precedence lists supported by FLX and their usage. We give the contradiction conditions that may occur when multiple precedence lists are used in a feature package and show how to resolve them. Finally, we show that the two types of FLX precedence lists are primitive: they can be used to implement arbitrary precedence relations among features that do not exhibit contradictions.

Keywords: Feature interaction, program entanglement, feature interaction resolution, reusable feature modules, Feature Language Extensions.

1 Introduction

In software engineering literature, the terms *feature*, *aspect* and *concern* are often used synonymously to denote certain functionality of a software system. For example, reliable data transport and congestion control are two features of the Internet TCP protocol. Features are implemented by computer programs. Two features *interact* if their *behaviors* change when their programs are integrated together. The behavior of a computer program is manifested in the sequence of program statements that gets executed and its output for a given input. Consider TCP again. Without congestion control, reliable data transport will retransmit when a duplicated acknowledgement is received. After congestion control is added, the same message may cause the sender to retreat to slow start. Thus these two features interact. The term feature interaction was coined by developers of telecommunications systems, but its occurrence is common place: when a software system *evolves*, it usually means that new features are added to the system changing the behavior of existing features.

We showed earlier [1] that if (C1) two features interact, (C2) they are executed by the same sequential process, and (C3) they are implemented by a programming

[1] Corresponding Author: W. H. Leung, Computer Science, Illinois Institute of Technology, 10 West 31st Street, Chicago, Illinois 60616, USA; E-mail: leung@iit.edu.

language that requires the programmer to specify execution flows, then the programs of the two features will inevitably *entangle* in the same reusable program unit of the programming language. If the features do not interact, then program entanglement is not necessary. Program entanglement implies that features are implemented by changing the code of one another. Besides making it difficult to develop features, entangled programs are difficult to reuse, maintain and tailor to different user needs. And feature interaction is the root cause of program entanglement.

(C1) and (C2) are generally dictated by the application such as the examples given earlier in TCP. Today's general purpose programming languages require (C3). Existing TCP implementations are notoriously entangled (e.g. see [2]). It is not because the programmers lacked skill; they could not help it.

The Feature Language Extensions (FLX) is a set of programming language constructs developed to solve the program entanglement problem. 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. The programmer does not specify the execution flows of program units; hence FLX relaxes (C3). A *feature* is composed of a set of program units; it is designed according to a *model* instead of the code of other features. Features are integrated in a *feature package*. Features and feature packages are reusable. Different combinations of them can be packaged to meet different needs. We have added the foundation FLX constructs to Java. A research version of the FLX to Java compiler is downloadable from [3].

We call the conditions under which two interacting features change their behavior their *interaction conditions*, and the interaction is *resolved* with specification on the new behavior. Presently, the programmer read code to determine when the interaction conditions may become true, and change code to resolve the interaction conditions. This is a labor intensive and error prone process, and a main reason why software development is complex.

Due to the way that the FLX compiler generates code, two program units written in FLX interact if the conjunction of their condition parts is satisfiable, or equivalently, if the condition parts of the two program units can become true at the same time. Two features interact when some of their program units interact. The satisfiable condition is their interaction condition. Several other researchers have constructed systems with this property (e.g. see [15]). As we shall see later, the condition part of a program unit is a set of quantifier-free first order predicate formulas. Detecting feature interaction in programs written in FLX then requires an algorithm, often called a satisfiability solver, which determines the satisfiability of such formulas.

The first order predicate satisfiability solver of FLX does not require iterations of trial and error incurred in prior art and is overviewed in [4]. This paper focuses on using FLX to integrate features and resolve their interaction without changing their code. In particular, we discuss usage of the *precedence list* facilities provided by FLX. A precedence list establishes a strict partial ordering[2] among a set of features in a feature package. FLX supports two types of precedence lists: a *straight precedence list* specifies that if the interaction condition for some of the features becomes true the programs of the features with higher precedence will get executed before the programs

[2] A strict partial order is an irreflexive, asymmetry and transitive relation between two elements of a set, denoted by "<". For all a, b and c in P, we have (i) ~(a < a) (irreflexivity); (ii) if (a < b) then ~(b < a) (asymmetry); and (iii) if (a < b) and (b < c) then (a < c) (transitivity).

of features with lower precedence; and a *priority precedence list* specifies that only the program unit belonging to the feature with the highest precedence will get executed.

Precedence list is a powerful facility. For example, in a telephony application written in FLX, the feature `DoNotDisturb` interacts with the plain old telephone service (`POTS`) whenever the phone is called. The interaction conditions of the two features are resolved in a single precedence list statement in a feature package. One of the authors came from the telecommunication industry and was involved in the development of `DoNotDisturb` in a production digital switch. The programmers in that project needed to go through hundred of thousands lines of code to find several hundred places to insert code for the feature. Later, as new features are added to the system, they had to remember not to forget including the code for the feature.

We first introduced precedence lists in [5]. A more detailed discussion is given in this paper. We review briefly the FLX constructs to specify features and feature packages in Section 2. In Section 3, we describe the two different types of precedence lists implemented in FLX. We also show there that precedence lists alone is not sufficient in certain situations. When that happens, the interaction condition is resolved by program units in the feature package. In Section 4, we discuss the integration of multiple precedence lists. This can happen, for example, when two feature packages each with its own precedence list is integrated in a feature package. Multiple precedence lists can lead to *contradictions* that need to be resolved. In the same section, we introduce the compound precedence statement which specifies the precedence relations among precedence lists. It is a short hand for multiple precedence lists. In Section 5, we show that the two types of precedence lists supported by FLX are primitive in the sense that they can be used to specify arbitrary precedence relationships that do not contain contradictions. We review related work in Section 6. Our method to integrate interacting features without changing feature code appears to be new. The use of precedence lists as language mechanisms to resolve interaction is also new. We conclude the paper in Section 7.

2 Some FLX basics

FLX supports the view that complex software should be organized as a collection of components and FLX is meant for the development of feature rich components called feature packages. In a telephone system developed using FLX, each telephone object is associated with two feature packages: a call processing feature package for features like call forwarding, and a digit analysis feature package for features like speed calling. Different telephone objects can be associated with different feature packages containing different sets of features, or the set of features can be the same but the feature interactions are resolved differently. We will use the call processing feature package as a running example for this paper.

As mentioned earlier, a feature written in FLX is developed according to a *model*. The model is composed of an *anchor feature* and a *domain statement*. The anchor feature implements the basic functionality; other features can be considered as its enhancements. Condition variables, called *domain variables* and *events*, are defined in the domain statement. They are used in the condition part of a program unit. Domain variables are initialized in the domain statement and space is allocated for them when a feature package using the domain statement is instantiated.

For this paper, we will skip showing the syntax of a domain statement. The domain statement for the call processing package, called `BasicTelephony`, contains a domain variable `state` which is a simple extension of the class `enum` defined in Java 1.5. We will not describe the extension here as it is related to the FLX satisfiability algorithm only. The possible values of `state`, such as `IDLE`, `RINGING`, `TALKING`, define the different states that the phone associated with the feature package can have.

Some of the events specified in the `BasicTelephony` domain statement come from another phone announcing its intent (`TerminationRequest`, `Disconnect`) or its state (`Busy`, `Ringing`, `Answer`) to this phone. Other events are signals (`Onhook`, `Offhook`. `Digits`) coming from the device driver of the phone. Events exchanged between phones contain a field `FromPID` identifying the sending phone.

Surprisingly large number of features can be developed using this relatively simple domain statement. Since a domain statement can be extended using the inheritance mechanisms of FLX [1], we advocate a *minimalist* approach in designing domain statements: Define only those domain variables and events for the set of features to be implemented at the time. Later, if new features require new domain variables, such as the role play by the phone, and new events, such as request to add a video channel, they can be added without affecting features that have already been developed.

A portion of the code for the anchor feature of the call processing feature package, called the plain old telephone service (`POTS`) feature, is given in Figure 1 showing only two 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 telling the calling party of that fact.

```
anchor feature Pots {
     domain BasicTelephony;

MakeCall {
    condition: state.equals(State.IDLE);
    event: Offhook; {
       fone.applyDialTone();
       state = State.DIALING;
               }
       }
ReceiveCall {
    condition: state.equals (State.IDLE);
    event: TerminationRequest e; {
       Ringing r = new Ringing (e.FromPID);
       rt.sendEvent (r);
       state = State.RINGING;
               }
       }
    }
                  .
                  .
                  .
}
```

Figure 1. A Portion of the FLX POTS code

The condition part of a program unit is composed of a *condition statement* and an *event statement*. The condition statement is a quantifier free first order formula of domain variables and their predicate methods. We do not support the existential and

universal quantifiers explicitly. When the programmer has the need to say something like "there exists some elements", we ask him to write a predicate method `non-empty()` instead. The event statement specifies a list of events. Each event may be attached with a *qualification* which is a first order formula on data carried in the event.

The feature `DoNotDisturb` is shown in Figure 2. Its program unit `SayBusy` returns a `busy` event to the caller (identified by the `fromPID` of the received event `e`) whenever the phone receives a `TerminationRequest` event.

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

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

Figure 2. The feature `DoNotDisturb`

```
feature package Fp3{
domain: BasicTelephony;
features: Fp1, F2;
priorityPrecedence(Fp1, F2);
    PU1{

     }
}
```

Figure 3. `feature package in FLX`

FLX requires that the interaction among the program units in a feature be resolved before the feature is compiled; similarly, the interaction conditions among features in a feature package must be resolved before the feature package is compiled. `DoNotDisturb` and `POTS` interact: when the event `TerminationRequest` is received, the condition part of `SayBusy` in `DoNotDisturb` becomes true and the condition part of several program units in `POTS`, including `ReceiveCall`, may become true. We show how these two features may be integrated in a feature package in Section 3.1.

The essential elements of a feature package are shown in Figure 3. It identifies one or more features and feature packages that will be integrated in the package and the domain statement used by them. The FLX compiler checks that the anchor feature is included in the list of features. The feature package may contain several precedence lists and program units. We will show how to use them to resolve interactions.

3 Resolving Feature Interaction with Precedence Lists

FLX provides two types of precedence lists, priority precedence and straight precedence. They are described in this section, illustrated with examples. We also show that while precedence lists are powerful mechanisms, there are situations that they are not sufficient.

3.1 Priority Precedence

When a programmer decides to use priority precedence list to resolve feature interactions, he specifies the features in descending order of priority in a list with the highest priority feature at the first position of the list. When an interaction condition becomes true, the program unit from the feature with the highest priority gets executed.

To help explain this, we show in Figure 4 the code of the feature package QuitePhone which integrates DoNotDisturb, POTS and CatchAll and uses a priority precedence list to resolve their interaction. The code for the feature CatchAll is given in Figure 5.

```
Feature package QuitePhone {
domain: BasicTelephony;
features: DoNotDisturb, CatchAll, POTS;
priorityPrecedence(DoNotDisturb, POTS, CatchAll);
}
```

Figure 4. QuitePhone feature package

```
feature CatchAll{
domain: BasicTelephony;
anchor: Pots;
    Catch{
      condition: all;
      event:   any; {
            System.err.println("unexpected condition and event”);
            BasicTelephonyEvent.getEventID(e));
      }
   }
}
```

Figure 5. CatchAll feature package

When a phone assigned with QuietPhone receives the TerminationRequest event (i.e. when it is called), SayBusy of DoNotDisturb will be invoked and a busy event will be sent back to the caller. Program units of POTS and CatchAll will not be invoked. However, when the phone receives an OffHook event and it is idle, then the MakeCall program unit of POTS gets invoked and the user can make phone calls. The Catch program unit of CatchAll will be invoked only when the phone is in a particular state and an event unexpected by DoNotDisturb and POTS arrives. CatchAll is a very useful exception handling feature. We introduced exception handling in FLX in [1] and will cover it more fully in a separate article.

3.2 Straight Precedence

With a straight precedence list, when an interaction condition becomes true program units from features that satisfy this condition are executed following the order in which the features are specified in the list. Figure 6 shows the StartMeter program unit of the Billing feature. It creates a billing record and starts the timer when a call is answered. StartMeter interacts with the CallAnswered program unit of POTS. Other program units of Billing and POTS also interact. The two features are integrated in the feature package NoFreeCalls as shown in Figure 7 with their interactions resolved in a straight precedence list.

Using this method, changing billing policy becomes quite easy. One can simply substitute one billing feature with another to gather different billing data.

```
feature Billing {
domain: BasicTelephony;
anchor: POTS;

StartMeter {
   condition: state.equals(State.IDLE);
   event: Answer e; {
       CallRecord     =     new     CallRecord
(e.fromPID);
       meter.start (1 second);
       }
   }
}
```

Figure 6 A program unit in Billing

```
feature package NoFreeCalls {
    domain:    BasicTelephony;
    features: Billing,Pots;
    straightPrecedence (Billing, Pots);
    }
```

Figure 7 The NoFreeCalls feature

3.3 Precedence Lists Are Not Always Sufficient

In the examples of Figure 4 and Figure 7, a single precedence statement is used to resolve the interactions of features in a feature package. But very often, more flexible and finer control of the interaction condition is needed.

Consider the `CallForwarding` feature with its most important program unit shown in Figure 8. `CallForward` transfers an incoming call by relaying the `TerminationRequest` event to the forward number if that number is defined and the call is not coming from that phone. Suppose that we integrate `DoNotDisturb` and `CallForwarding` together and place `DoNotDisturb` ahead of `CallForwarding` in a priority precedence list, no call will be forwarded. If we place `CallForwarding` ahead of `DoNotDisturb`, all calls will be forwarded.

```
feature CallForwarding {
domain: BasicTelephony;
anchor: POTS;

ForwardCall {
     condition: state.equals (State.IDLE);
     event: TerminationRequest e; {
        if ((forwardNumber != “”) && (forwardNumber != e.fromPID)) {
               rt.send (forwardNumber, e);
               stop;
               }
       }
      }
}
```

Figure 8. The ForwardCall program unit of CallForwarding

The programmer can choose to use a program unit in the feature package to resolve the interaction among the two features. In the example given in Figure 9, the

interaction between DoNotDisturb and CallForwarding is resolved depending on whether the caller is identified in a list of phone numbers.

```
feature package SelectiveCallForwarding {
     domain: BasicTelephony;
     features: DoNotDisturb, CallForwarding, Pots, CatchAll;
     priorityPrecedence (DoNotDisturb,CallForwarding,POTS,CatchAll);

     LinkedList phoneList = LinkedList (empty); // forwardable phones

    SelectToForward {
        condition: state.equalsTo(State.idle);
        event: TerminationRequest e; {
                if (phoneList.contains (e.FromPID))
                        CallForwarding;
                else
                        DoNotDisturb;
                stop;
                }
        }
    }
```

Figure 9. SelectiveCallForwarding feature package

By convention, a program unit in a feature package has highest precedence. Thus when the interaction condition becomes true, `SelectToForward` is executed first. The `stop` statement at the end of the program unit instructs the compiler not to invoke program units of lower precedence. In the example, SelectToForward refers to the features instead of calling their program units. The FLX compiler generates code to invoke the correct program units in these features. Alternatively, the program may call the program units of the features explicitly. In that case, the compiler will check that the program units are called with the correct condition as `SelectToForward`.

4 Multiple and Compound Precedence Lists

FLX supports multiple precedence lists and compound precedence lists in a feature package. When feature packages containing precedence lists are integrated together, the integrating feature package contains multiple precedence lists by definition. A compound precedence list is a short hand to multiple precedence lists. Some FLX programmers argue that it is easier to understand than multiple lists. Precedence lists may contradict one another. The FLX compiler needs to identify the contradiction and enable the programmer to resolve the contradiction.

4.1 Combining Precedence Lists of the Same Type

FLX encourages its programmer to develop a feature based on the anchor feature only. The feature is usually tested with the anchor feature and beneficially with `CatchAll` in a feature package. When the programmer is finished with testing, he has two reusable programs: the feature itself and the feature package that he used to test the feature. The feature package that integrates the 3-way calling test package, called `3WayPackage`, and `SelectiveCallForwarding` (Figure 9) is given in Figure 10. The new feature package has two priority precedence lists: one from

3WayPackage containing the features 3Way, POTS and CatchAll; the other comes from SelectiveCallForwarding containing DoNotDisturb, CallForwarding, POTS and CatchAll. The interaction between the two feature packages is resolved in another priority precedence list.

```
feature package 3WayAndSelectiveCallForwarding {
    domain: BasicTelephony;
    anchor: POTS;
    features: 3WayPackage, SelectiveCallForwarding;
    priorityPrecedence (3WayPackage, SelectiveCallForwarding);
}
```

Figure 10. 3WayAndSelectiveCallForwarding feature package

When combining precedence lists of the same type, which is the case in the example of Figure 10, the FLX compiler applies two rules: First, a feature may appear in multiple lists but only one instance of it will appear in the combined list. Second, the partial ordering specified in the different lists is merged into a combined list. Following these two rules, the priority precedence list of the feature package in Figure 10 contains the following features in descending order: 3Way, SelectiveCallForwarding, DoNotDisturb, POTS, and CatchAll. SelectiveCallForwarding is considered a feature as it contains program units of its own. The net effect of combining the precedence lists in the example is adding the 3Way feature to SelectiveCallForwarding: When the phone is in its talking state, it can invoke the 3Way feature. Incoming calls are no longer blocked by DoNotDisturb in talking state; they will cause an audible signal to the speaker as specified by 3Way.

The first rule of combining precedence lists is similar to virtual base classes in C++. The second rule may not be possible if in one list a feature f1 precedes feature f2 but in another list f2 is specified to precede f1. When that occurs, the FLX compiler will identify an *order contradiction.* An order contradiction is relevant only in the condition where f1 and f2 interact. The FLX compiler will identify that condition and the programmer can resolve the contradiction in a program unit of the feature package that combines the two lists. The condition part of the program unit will include the interaction condition and its program body will specify the computation when the condition becomes true.

4.2 Integrating Precedence Lists of Different Types

Suppose that we want to integrate the features Billing (Figure 6), POTS and CatchAll together. Billing has a straight precedence relationship over POTS and both of them should have priority precedence over CatchAll. The programmer can simply put these three precedence relations in the feature package that integrates these three features as shown in Figure 11.

Following the precedence specifications, when some interaction condition between Billing and POTS becomes true, appropriate program units from these two features will be executed in order, and CatchAll will not get invoked as the other two features have priority precedence over it.

```
feature package BillingPackage {
    domain: BasicTelephony;
    feature: Billing, POTS, CatchAll;
    straightPrecedence (Billing, POTS);
    priorityPrecedence (Billing, CatchAll);
    priorityPrecedence (POTS, CatchAll);
}
```

Figure 11. `BillingPackage feature package`

When precedence lists of different types are combined, the FLX compiler checks for whether there is *type contradiction*. A type contradiction occurs when a feature is specified as having both priority and straight precedence over the other. For example, feature `f1` has straight precedence over `f2` and `f3` in one list. In another list `f2` has priority precedence over `f3`. When an interaction condition for the three features becomes true, it is not clear what should be done for the program unit in `f3` after program units from `f1` and `f2` have been executed. The FLX compiler identifies the interaction condition, and the programmer must specify in a program unit in the integrating feature package to resolve the ambiguity.

Precedence lists of the same type can be combined into a single partial ordering list, but not for precedence lists of different types. If we have a first priority precedence list including `f1`, `f2` and `f3`, and a second priority precedence list including `f2` and `f4`, we know that `f1` has priority precedence over `f4` from the transitivity property of strict partial orderings. But if the second list specifies straight precedence, then we do not know the precedence relationship between `f1` and `f4`.

4.3 Compound Precedence List

One observes that in the feature package of Figure 11, both `Billing` and `POTS` have priority precedence over `CatchAll`. Using a method similar to factorization in algebra, one can reduce the multiple precedence lists into a single compound precedence list as shown in Figure 12.

```
feature package BillingPackage {
    domain: BasicTelephony;
    feature: Billing, POTS, CatchAll;
    priorityPrecedence(straightPrecedence(Billing,POTS), CatchAll);
    }
```

Figure 12. `BillingPackage feature package`

The precedence list of Figure 12 says that when an interaction condition becomes true for the three features, program units in `Billing` and `POTS` will be executed in order according to the straight precedence clause. The program unit in `CatchAll` will not be executed because of the priority precedence specification. In essence, the compound precedence list of Figure 12 is a short hand of the precedence lists in Figure 11. We know that they are equivalent because both will generate the same partial ordering as well as precedence types among the different features.

5. Priority and Straight Precedence Lists are Primitive

With two features, when we say one precedes the other there can be only two meanings: that the program of one overrides that of the other (priority precedence), or the program of one should be executed before the other (straight precedence). When there are more features, a question arises: Can we use only the priority and straight precedence list to implement arbitrary precedence relations among arbitrary number of features? Arbitrary combinations of these features may have different precedence relations with one another.

The question is important because precedence list mechanisms directly affect the way the compiler generates code. Each time we discover a precedence relation that cannot be implemented by the mechanisms already provided, we need to modify the compiler to support it.

Fortunately, the answer to the question is affirmative if the desired precedence relation among the features does not contain order nor type contradictions. We use two steps to show the above statement. First, we show that features with arbitrary precedence relations and without contradiction can be represented generically. Step 2 shows that such a generic representation can always be implemented by the two types of precedence relations.

Consider a set of features `f1` to `fn`. Since there is no order contradiction, the features can be arranged linearly according to their partial ordering (such that `f(i-1)` either precedes or has no precedence relationship with `fi)`. The possible precedence relationship among the features can be represented in a square matrix with the features arranged in order on the coordinates of the matrix. The diagonal of the matrix is empty as it makes no sense to say a feature precedes itself. The lower left triangle of the matrix underneath the diagonal is also empty because we already say that `fi` does not precede `fj` for `i` > `j`.

Each element in the upper right triangle above the diagonal will indicate the type of precedence between `fi` and `fj`, for all `i` < `j`. Since there is no type contradiction, the value in each such element is either empty, showing priority precedence or showing straight precedence. Figure 13 shows such a matrix for the feature package given in Figure 12.

	`Billing`	`POTS`	`CatchAll`
`Billing`		Straight precedence	Priority precedence
`POTS`			Priority precedence
`CatchAll`			

Figure 13. Precedence relation matrix among the features in the feature package of Figure 12.

Given a precedence relation represented by such a matrix, the simplest way to implement it will be to use the appropriate precedence list for each nonempty element linking `fi` and `fj`, hence the answer is affirmative to the question of whether priority and straight precedence lists are primitive. The interested reader is encouraged to devise algorithms, similar to Karnaugh maps [6], which will generate the minimum number of precedence lists using compound precedence lists. In a feature package with many features, the programmer may find it useful to construct such a table to aid in the feature interaction resolution design.

6. Related Work

The feature interaction problem affects all stages of software development, from the difficulties in recognizing interaction conditions during system specification to the difficulties in testing as feature programs constantly get changed if they are implemented with existing programming languages. FLX and its precedence lists facilities focus on solving the *implementation* problem of enabling the programmer to develop reusable feature modules without entanglement. The discussion in this section therefore emphasizes on work that allows specification of executable feature code.

Among the pioneers that used programming language to facilitate the development of features, the languages Statechart [7], LOTOS (e.g. [8]), DSL [9] and Esterel [10] are the most important. They take different approaches. For example, Statechart is graphical and Esterel assumes instantaneous reaction to input. All of them support explicitly definition of finite state machines. Several of them, e.g. Esterel and LOTOS, developed verifiers for programs developed using them. But one cannot use them to develop interacting features as reusable program modules without entanglement. An empirical study [11] using these languages observed that "we have to rewrite POTS several times as features were added," and asked: "how could reusable (sub)specifications be declared and used?"

CRESS [12] is the first graphical language that explicitly supports the specification of features and their integration. It is a substantial work that included a model checker. But integrating features in CRESS require the programmer to manually determine where to "splice" and "insert" code to the features.

Plath and Ryan extended the input languages of the model checkers SMV [13] and SPIN [14] to allow for the specification of features extending from a base system ([15] and [16]). Since the input languages of SMV and SPIN are mainly nonprocedural, their result on feature specification is quite similar to ours. They already had the concepts of anchor feature and features; we provide additional constructs and facilities to integrate features without requiring changing code.

The notion that feature interaction can be resolved by arranging the features in some priority or precedence order has been suggested by a number of authors (e.g. [29]). We are aware of only one other work that has defined programming language facilities to specify feature precedence. Similar to FLX, the "Stack Service Model (SSM)" [30] associates a phone with a number of features. The features in SSM are put into a stack. The priority of a feature is determined by its position in the stack. The notion of feature interaction resolution in SSM is by preserving the safety assertions of the features; in FLX, it is specifying behavior change. Precedence relationship in FLX is partial ordering of two different types; SSM has a single stack and depends on feature code to determine whether a feature will override or will execute before features of lower precedence. Execution of a feature in SSM is triggered by a token passed from one feature to the other in the same stack; in FLX, it is due to a condition becomes true. Because of these differences, we believe that the feature programs in SSM will tend to be more tightly coupled with one another than the feature modules in FLX, and adding a feature in the stack of SSM will often require code changes in the other features.

The related works discussed so far are often considered to be "specification languages" instead of "programming languages". For example, they typically do not allow the programmer to define more complex data structures. More recently, there are

three other programming language approaches besides our own: Aspect oriented programming (AOP) [17], Call Processing Language (CPL) [18] and Feature oriented programming (FOP) [19].

AOP textually separates the code of a feature from the base code and put it into a program module called an *aspect*. An aspect is a nonprocedural program containing a set of (*point cut, advice*) pairs. A point cut is an assertion on some syntactic artifacts (class and method names, etc.) of the base code; it must pin point some specific statements (called *joint points*) in the base code. Its corresponding advice specifies how the base code is changed by adding to or replacing the joint points. The AOP compiler will weave the aspect into the base code. In general, the programmer manually reviews code to determine the point cuts to the base code and to other aspects; the advices are not independent of the base code and other aspects. As a result, aspects are not easily reusable without one another. Empirical studies conducted a decade apart showed that AOP does not improve programmer productivity ([20] and [21]), despite studies that showed it can significantly reduce the amount of code to be written (e.g. [22]).

The term FOP was first coined by Prehofer in [23] where he introduced procedural language extensions to Java to specify the notion of a feature. But his approach requires resolving the interaction between every pair of features and often by changing code. Batory and his group propose that composition of features follows mathematical formulas (e.g. feature interactions are derivatives as in calculus [24]). But in general his method requires the programmer to significantly reconfigure code manually.

Similar to VoiceXML [25], CPL and its variants such as LESS [26] are mark up languages. Adding features with these languages require changing code.

We should also mention Service Oriented Architecture (SOA). SOA is a recent architectural approach. It proposes to relax (C2) of the entanglement conditions so that every feature is a process. The service processes interact by requesting and providing services to one another. Many features are required to be executed in the same process, and a deeper analysis showed that SOA exhibits a fractal structure with significant performance and complexity implications (A fractal structure leads to chaos) [27].

7. Conclusion

FLX is designed to enable the programmer to develop interacting features separately as reusable program modules. The precedence lists are language facilities that allow the programmer to integrate interacting features and resolve their interaction conditions without requiring changing their code. While there are cases where precedence list cannot apply, they are powerful mechanisms: A single precedence can resolve a large amount of interaction conditions for many features. We gave more than ten examples to illustrate their usage including when multiple precedence lists are combined in a feature package.

For readability, the examples given in the paper are relatively simple. But we have developed fairly complex software using FLX. We used it to develop more than forty features and feature packages on a simulated telephony system. The telephony systems were developed mainly to test FLX concepts and its compiler. We have started to use FLX to develop something that can be used. We recently finished developing the basic features of a call center based on Skype [28]. We started on the development of a composable operating system.

When FLX was first conceived, reviewers can immediately see that it will help in improving programmer productivity in the development of individual features, because the programmer can focus on the feature independent of other features. Many, however, were skeptical that we are just pushing the complexity to the feature package where the features are integrated. Our experience shows that because FLX can detect feature interaction conditions automatically and provides mechanisms like precedence lists to facilitate interaction resolution, integrating features can usually be accomplished without much difficulty.

A number of results on FLX including its interaction detection algorithm, exception handling mechanisms, and language constructs to extend application models are not yet published. But materials (theses, powerpoints) on them as well as a research version of the FLX to Java compiler and example FLX code can be found in its web site [3]. FLX is designed so that programs written in it can be verified using assertions based verification instead of completely relying on testing. The basis of that goal is given in [4].

References

1. Leung, W. H.: Program Entanglement, Feature Interaction and the Feature Language Extensions. Computer Networks, Volume 51, February, 2007, 480-495
2. M. Musuvathi and D. Engler: Model Checking Large Network Protocol Implementations, Proceedings of Symposium on Network Systems Design and Implementation, 2004.
3. www.openflx.org
4. W. H. Leung: On the Verifiability of Programs Written in the Feature Language Extensions, Proceedings of 10th IEEE International Symposium on High Assurance Systems, November, 2007.
5. Leung, W. H.: Writing Reusable Feature Programs with the Feature Language Extensions, Proceedings of Feature Interactions in Telecommunications and Software Systems VIII, IOS Press, 2005.
6. Karnaugh, M.: The Map Method for Synthesis of Combinational Logic Circuits. Transactions of American Institute of Electrical Engineers part I **72** (9): 593-599, November 1953.
7. Harel, D.e.a., Statemate: A Working Environment for the Development of Complex Reactive Systems. IEEE Transactions on Software Engineering, 1990. 16(4).
8. Turner, K. J., "A LOTOS-based Development Strategy," Formal Description Techniques II, pages 117-132, 1990.
9. Ellsberger, J., D. Hogrefe, and A. Sarma, SDL Formal Object-oriented Language for Communicating Systems. Hemel Hempstead: Prentice Hall Europe, 1997.
10. Berry, G., and Gonthier, G., "The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation," Science of Computer Programming, 19:87-152, 1992.
11. Ardis, M. A., "Lessons from Using Basic LOTOS," Proceedings of the International Conference on Software Engineering, May 1994.
12. Turner, K. J., "Modular Feature Specification," Proceedings of MICON, August, 2001.
13. McMillan, "Symbolic Model Checking," Kluwer Academic Publishers, 1993.
14. Holzmann, G. J., The SPIN Model Checker : Primer and Reference Manual, Addison-Wesley Professional, September 4, 2003.
15. Plath, M. and Ryan, M. D., "A Feature Construct for Promela," in SPIN'98 – Proceedings of the 4th SPIN Workshop, November 1998.
16. Plath, M. and Ryan, M. D., "Feature Integration Using a Feature Construct," Science of Computer Programming, January 2001.

17. Elrad, T., R.E. Eilman, and A. Bader, Aspect-Oriented Programming. Communications of the ACM, October, 2001. 44(10).
18. Lennox, J., X. Wu and H. Schulzrinne, "Call Processing Language (CPL): A Language for User Control of Internet Telephony Services," IETF RFC 3880, October 2004.
19. Batory, D., J. N. Sarvela and A. Rauschmayer, Scaling Step-Wise Refinement, Proceedings of International Conference on Software Engineering 2003 (ICSE 2003), Portland, Oregon, May 2003.
20. Murphy, G.C., R.J. Walker, and L.A. Baniassad, Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming. IEEE Transactions on Software Engineering, 1999. 25(4).
21. Filho, F., Rubira, C., Garcia, A., "A Quantitative Study on the Aspectization of Exception Handling," Proceedings of ECOOP Workshop on Exception Handling in OO Systems, July, 2005.
22. Lippert, M., and C. V. Lopes, A Study on Exception Detection and Handling Using Aspect-Oriented Programming, Proceedings of International Conference on Software Engineering, ICSE 2000.
23. Prehofer, C., An Object Oriented Approach to Feature Interaction, Proceedings of the Feature Interaction Workshop, 1997, IOS Press.
24. Liu, J., Batory, D. and Nedunuri, S., "Modeling Interactions in Feature Oriented Software Design," Proceedings of Feature Interactions in Telecommunications and Software Systems, VIII, June, 2005.
25. www Consortium, "Voice Browser Activity," 2005, http://www.w3.org/Voice
26. Wu, X., and Schulzrinne, H.: Handling feature interaction in the language for end system services, Computer Networks, Volume 51, February, 2007.
27. Bussler, C.: The fractal nature of web services, IEEE Computer, March 2007.
28. http://skype.com
29. Chen, Y.L., Lafortune, S. and Lin, F., "Resolving Feature Interactions Using Modular Supervisory Control with Priorities," Proceedings of Feature Interactions in Telecommunications Systems, 1997, IOS Press, Amsterdam.
30. Samborski, "Stack Service Model," Gilmore, S., and Ryan, M., editors, Language Constructs for Describing Features, Springer-Verlag, London Ltd, 2000/2001.