Writing Reusable Feature Programs with the Feature Language Extensions

Wu-Hon F. Leung
ECE Department, Illinois Institute of Technology, Chicago, IL 60616, USA
leung@iit.edu

Abstract. One of the most difficult tasks in the development of interacting features is that it requires a labor intensive and error prone process of examining and modifying code. The feature code typically ends up scattered and not reusable. In this paper, we describe a set of programming language extensions that will allow the programmer to develop interacting features as reusable program modules even though the features may change the execution flow of each other. The programmer uses these language extensions to specify nonprocedural program units, organize the program units into reusable features and integrate features into executable feature packages. The language extensions enable an automatic procedure to detect interaction condition among features; the programmer can resolve the interaction without changing feature code. Furthermore, the programmer may package different sets of features and resolve their interactions differently to meet different user and market needs. These language extensions have been implemented on Java and used to develop a prototype telephony system.

1. Introduction

The problem of feature interaction has perplexed software designers ever since it was observed by the developers of the first telephone electronic switching system [1]. The generally accepted definition of two features interacting with each other is that when their implementations are put together, they change the behavior of one another (e.g. see [2]). The problem affects all stages of a software development, from the difficulties in recognizing interaction conditions during system specification to the difficulties in testing. This paper focuses on the design and implementation of interacting features, although the results reported here have implications on specification and testing as well.

Presently, the programmer implements a feature by modifying the code of other features that it interacts with. 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 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. If the features are developed by different programmers, as often is the case in large scale development, the programmers have to ask for changes in code still under development. The code of a feature often ends up scattered among the code of its interacting features thus not reusable and making the resultant program even more difficult to maintain. Later we will show that given existing general purpose programming languages, these difficulties are inevitable for interacting features executed in the same sequential process.

The problems described in the above paragraph are common software development problems and do not just apply to telecommunication software. When the code of a feature is scattered, researchers on Aspect Oriented Programming (AOP) [3] call the feature a cross cutting concern. In the method of AspectJ [4], to separate a cross cutting concern $C$ from a program $P$ (primary concern), the code for $C$ is put into an aspect $A$. $A$ contains a set of pointcuts and their advices. Each pointcut $J_i$ specifies some locations in $P$. Its corresponding advice $V_i$ specifies the code to be executed when the locations specified by $J_i$ is traversed during the execution of $P$. Inserting the code of $A$ into those of $P$ is called weaving, a procedure invisible to the programmer. Being able to separate cross cutting concerns is an advance in the art. But to determine where the pointcuts are is still a manual process. An aspect is so tightly coupled with its underlying program $P$ that it is not reusable outside of that context. It is not clear how it will be handled if there is another concern cutting across $P$ and some other aspects. Some argue further (e.g. see [5]) that because aspects may read and modify variables outside of an object (de-encapsulating object variables) and the additional logic is not in the same place as the original code, maintaining and verifying aspects and $P$ are more difficult than just simply changing $P$.

We therefore suggest that a solution to the feature interaction implementation problem will allow the programmer to develop a new feature independent of its interacting features. The solution should include a tool that can automatically detect the interaction conditions among features, and the interacting features can be integrated without changing the code of each other. By implication, the feature code then becomes reusable; and they can be packaged differently to meet different user and market needs. Since adding features will not modify the code of other features, the task of testing becomes easier. The tool that detects feature interaction conditions will also take the guessing out of specifying them.

We developed a set of programming language extensions, called Feature Language Extensions, or FLE, on Java [6] to meet the above requirements. FLE enables nonprocedural programming in which the programmer does not specify the execution sequences of the program units. A program unit in FLE consists of a condition part and a program body. The program body gets executed when its corresponding condition part becomes true. FLE supports a design paradigm in which the programmer designs a feature following a model instead of examining the code of interacting features. We will give a more detailed discussion of the design rationale for FLE in Section 2.

We describe the set of foundation language extensions 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 language extensions and facilities to resolve interactions among features when they are put together in a feature package. Throughout
sections 3 and 4 we use examples from a prototype telephony system developed using FLE. Telephony systems are among the most difficult software to develop [7]. Compare to the Transmission Control Protocol (TCP) of the Internet [8], the control mechanism in our examples has similar number of states but has more messages and interactions.

We acknowledge related work in Section 5 and conclude the paper in Section 6.

2. The Basic Approaches of the Feature Language Extensions

2.1 Why Nonprocedural Programming?

We repeat the definition of feature interaction here: two features interact if they change the behavior of one another. A feature is implemented by a set of computer programs. The behavior of a computer program for a given input is manifested in its output value and execution flow, referring to as the sequence of statements that gets executed. We therefore do not consider two features to be interacting if they are executed consecutively and the one that is executed before merely changes the input to the other but not its behavior. It follows that if two features interact, then under certain condition, which we shall refer to as the interaction condition, some of their programs will be executed concurrently.

We focus on when the interacting features are executed by the same sequential process. In this case, when the interaction condition becomes true, one of the features gets invoked first but before its execution is completed, the other features will get invoked. Existing general purpose programming languages ask the programmers to specify the execution flows of their programs. Consequently, one has to change the code of an existing feature to add the code of a new interacting feature. The more interactions (the more execution flows that get affected) usually means that the code for the new feature will be more scattered and the job of adding new features is more difficult.

A large class of applications belongs to this case. For example, in TCP, the same process executes the reliable data transport, flow control, congestion control and other features of the protocol at the same time. In a telephony application, the process that controls a simple telephone call would also execute other features such as giving call waiting indications to the user, and switching between calls. As evident from these examples, one cannot in general design away feature interactions.

2.2 The Design Paradigm of the FLE

The FLE takes the position that the software of complex computer applications should be organized into components and the language extensions are designed for the programmer to develop feature rich components.

A feature contains a set of program units and the programmer specifies a feature following a model. The model is composed of a domain statement that defines the condition variables used in the condition part of a program unit and a special feature called an anchor feature that implements the basic functionalities of a component. Other features of the component can be thought of as enhancements to or extensions of the anchor feature.
Features are put together in *feature packages* where interactions among them are resolved. Different feature packages have different combinations of features or they resolve the interaction differently. It is in this sense that we say the features are *reusable*. A feature package may be considered as a feature in another feature package. Different instances of the same component may be associated with different feature packages. It is in this sense that we say the feature packages are also reusable.

In our telephony prototype, each phone object is associated with two feature rich components, one for digit collection and digit analysis (allowing for features like speed calling and others), and the other for call processing (allowing for features like call waiting and others). Different phone instances can have different sets of features. Other objects (e.g. GUI) that control the phone are more conventionally coded.

Before we leave this section, we should mention that FLE program execution is triggered by events. Event driven programming is a popular design paradigm for GUI and other applications that require inter-process communication. It is supported by the most popular programming languages such as Java and Visual C++. Event-driven programming actually can be applied broadly. The following outlines a possible architecture for the parser of a compiler, a text processing application. The program text first goes through a recognizer, perhaps written in *awk* [9]. When it encounters a keyword, the recognizer sends the keyword as an event to the analyzer which checks the grammar and builds symbol tables and other artifacts for code generation.

3. The Foundation Language Extensions

We now give more details of the FLE constructs of domain statement, program units and features, using examples to illustrate their usage.

3.1 Domain Statement

A domain statement contains the definition of the condition variables, called *domain variables*, and *events*, of a model. The domain statement for the call processing features in our prototype is given in Figure 1. It contains a domain variable *state* that can have a range of values and is initialized to IDLE. *state* is declared to be of type *DTenum*. *DTenum* is a *domain data type* and is implemented with an extension to the class *enum* recently defined in Java 1.5 [10]. The extension is needed for a more efficient way to detect interaction conditions. A domain data type must contain public *boolean* variables or predicates (methods that return *boolean*). The domain statement of Figure 1 specifies a finite state space. The state variable in the digit collection and analysis component of our prototype uses a condition variable of type integer which is unbounded.

Some of the events specified in the domain statement comes from another phone announcing its intend (*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 variable is of type *event*. It is not very important in this paper to describe *event* in detail but to note that an event variable may contain *qualifying variables* that are of some domain data type.
3.2 Program Units

The condition variables declared in a domain statement are used in the condition part of a program unit. We show the program unit MakeCall of the plain old telephone (POTS) feature in Figure 2. It simply specifies that if the phone is IDLE and the Offhook signal is detected, apply dial tone to the phone and change the value of state to DIALING.

The condition part of the program unit has two statements. The condition statement is a Boolean formula of domain variables (if they are of type boolean) and their predicates. In the example of Figure 2, it is a single predicate on the value of state. In general, it is a first order predicate formula. FLE 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.

The event statement of the condition part specifies a list of events and their respective qualifications. The qualification of an event is a Boolean formula composed of its qualifying variables and their predicates.

FLE provides the enter and leave pseudo-events for the programmer to specify condition parts without an actual triggering event. Figure 3 shows two program units: RingPhone rings the phone whenever the state of the phone is RINGING, and RemoveRinging removes the ringing as soon as the state of the phone is no longer RINGING. enter and leave came from an earlier work [11].

In the condition statement, the programmer may use the keyword all to indicate that the condition statement is true no matter what values the domain variables have. For the event statement, the programmer may use the keyword any to indicate that the program unit can be triggered by any event. We will show example usage of these two keywords later.

```plaintext
domain BasicTelephony {
    variables:
        D'Tenum 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;
}

MakeCall {
    condition: state.equals(State.IDLE);
    event: Offhook; {
        fone.applyDialTone();
        state = State.DIALING;
    }
}
```

Figure 2
3.3 Features

A feature is implemented by one or more program units and a constructor. The anchor feature POTS (for Plain Old Telephone Service) is given in Figure 4 showing only two of its program units, MakeCall and ReceiveCall, and the feature constructor. The feature constructor performs the same function as the constructor of a class. In the example, the handle of the phone that the feature controls and the handle of the switch that the feature communicates with are passed to the feature when the feature is first instantiated. The feature is implemented in 18 program units and 195 lines of code.

The reader may notice that the feature declaration is very similar to a class declaration. Instead of directly called methods in a class, a feature contains nonprocedural program units. A feature references the domain statement that its program units use 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.

```java
anchor feature Pots {
    domain BasicTelephony;
    Phone fone;
    Router rt;
    public Pots (Phone fone, Router rt) {
        this.fone = fone;
        this.rt = rt;
    } // constructor for anchor feature Pots
    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;
        }
    }
}
```

This feature is implemented by the following code:

```java
public class Pots {
    private Phone fone;
    private Router rt;
    public Pots (Phone fone, Router rt) {
        this.fone = fone;
        this.rt = rt;
    }
    public void makeCall() {
        if (state.equals(State.IDLE)) {
            // Offhook
            fone.applyDialTone();
            state = State.DIALING;
        }
    }
    public void receiveCall() {
        if (state.equals(State.IDLE)) {
            // Termination Request
            state = State.RINGING;
        }
    }
}
```

Figure 4

```java
RingPhone {
    condition: state.equals(State.RINGING);
    event: enter; {
        fone.applyRinging ();
    }
}

RemoveRinging {
    condition: state.equals(State.RINGING);
    event: leave; {
        fone.removeRinging ();
    }
}
```

Figure 3

```java
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
    fp.sendEvent (Offhook); //User picks up phone
    // Some other code
}
```

Figure 5
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 Figure 5. An executable feature object inherits a number of system methods including the method `SendEvent()` which allows other programs to send a triggering event to the feature.

Once the domain statement and the anchor feature have been defined, the programmer uses them as the basis to design additional features. We show the features `DoNotDisturb` and `CallForwarding` in Figure 6 and 7 respectively. The former feature returns a `Busy` event to whoever calls the phone; and the latter forwards the call to a prearranged number if the phone is idle.

The `DoNotDisturb` feature is implemented with a few lines of code in the FLE. In one software development project that the author participated, it was quite difficult to implement as the programmer had to trace existing code to find all the places where the message `TerminationRequest` may happen. Using tools like “grep” [12] or “Cscope” [13] was not always helpful because in many situations, the message was not processed in existing code and hence the symbol did not appear. The programmer was not able to encapsulate the logic of the feature in one place. Using the terminology of AOP, this feature is a classic example of cross-cutting concern.

```plaintext
feature DoNotDisturb {
  domain BasicTelephony;
  anchor POTS;
  Router rt;
  // constructor not shown
  SayBusy {
    condition: all;
    event: TerminationRequest e; {
      Busy b = new Busy();
      rt.sendEvent (Event.FromPhoneID, b);
    }
  }
}
```

```plaintext
feature CallForwarding {
  domain BasicTelephony;
  anchor POTS;
  Router rt;
  String forwardNumber;
  //constructor not shown
  ForwardCall {
    condition: state.equals (State.IDLE);
    event: TerminationRequest e; {
      rt.sendEvent (forwardNumber, e);
    }
  }
}
```

Figure 6

Figure 7

4. Resolving Interactions Using the Feature Language Extensions

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

4.1 Interaction Detection

The run time system for FLE 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 two FLE program units can be invoked for execution concurrently, i.e. they interact, if and only if the conjunction of their condition parts is satisfiable and the satisfiable condition is reachable. A condition is reachable, if given the initial values of the domain variables, there exists a sequence of events triggering the execution of a sequence of program units leading to the condition being true.

Extending program unit interaction to feature interaction, one can show that two features interact if and only if there is a program unit in one feature that interacts with a program unit in the other feature.

To detect whether two program units interact, we only look for whether the conjunction of their condition parts is satisfiable. In other words, the FLE compiler tells the programmer that two program units interact even if the interaction condition may not be reachable. We feel that it is prudent to do so because what is not reachable in one release may become reachable after some revision or addition of new features. In addition, checking for reachability can be quite time consuming.

The condition parts of program units are predicate formulas on variables and predicates of domain data types. To determine the satisfiability of predicate formulas is considerably harder than to determine the satisfiability of Boolean formulas [14]. We enlist the help from the programmer. Domain data types must include a combination method that is used in our satisfiability solver. To get into more details of our method will be outside the scope of this paper.

If the interaction condition is not resolved, the runtime system must choose which one of the two program units to execute when the condition becomes true. FLE requires the programmer to resolve the ambiguity, a process which we call interaction resolution. The compiler checks for interaction condition among program units in a feature and among features in a feature package until the feature and the feature package is interaction free.

The mechanisms provided by the language to resolve interaction is similar whether the interaction is between program units within a feature or between features, except for the fact that in the latter case, the interaction is resolved in a feature package so that the features themselves need not be modified. Therefore we will give only examples in resolving feature interaction.

4.2 Interaction Resolution with Precedence List

The programmer uses the feature package statement of FLE to integrate a set of features together. As a result, it differs from a feature or an anchor feature statement in that it contains a statement specifying the list of features in the package. The compiler checks that all the features refers to the same domain statement, there is at least one anchor feature in the list, and all other features refers to an anchor feature in the list.

Figure 8 shows the code of the feature package QuietPhone containing the features POTS, DoNotDisturb and CatchAll. The interactions among these features are
resolved by the precedence statement, **priorityPrecedence**. A phone assigned with this feature package will not accept any calls but will allow the user to make phone calls. An executable feature package is invoked similar to an executable anchor feature.

In the example of Figure 8, we introduce a new feature, **CatchAll**, and a new language mechanism, a precedence statement. We discuss the precedence statement first.

The **priorityPrecedence** list in the example resolves the interaction among the features in the list in the following manner: If the condition parts of some program units belonging to the features in the list have become true at the same time, the program unit in the feature with the highest precedence will get executed.

As a result, when the phone that uses **QuietPhone** receives the **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** will get invoked and the user can make phone calls.

FLE currently supports another type of precedence list called **straightPrecedence**. When the condition parts become true for program units from different features in the **straightPrecedence** 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. FLE allows multiple precedence lists in a feature package, and the compiler checks that they do not result in contradictions. It is of interest to note that Mr. Karthik Ramachandran showed that using only **priorityPrecedence** and **straightPrecedence**, he can implement the other precedence lists that we have come up with, suggesting that they may be primitive. Again, to dwell more on this is outside the scope of this paper.
4.3 A Brief Digression to Exception Handling

FLE encourages the programmer to specify exception handling logic as features. CatchAll, whose code is given in Figure 9, is an example. When placed at the bottom of a precedence list, it will catch, at run time, conditions not specified in other features.

Being able to separate exception handling logic as features has many advantages. From the obvious that a feature like CatchAll is reusable; to the more subtle that it will localize impact when exceptions are added or changed unlike “catch and throw” mechanisms. We will address this important topic more fully in a future paper.

4.3 Interaction Resolution with Program Units

We will use examples to show two more points. The programmer may use program units in a feature package to gain finer control of interaction resolution. Second, a feature package may be reused as a feature in another feature package.

The features DoNotDisturb and CallForwarding interact when some one calls while the phone is idle. If we put DoNotDisturb ahead of CallForwarding in a precedence list, no call will be forwarded. If we reverse the precedence, then all calls will be forwarded while the phone is idle. There is another alternative. Each feature written in FLE inherits an activate () and a deactivate () methods. The programmer can use those two methods to specify that these two features are mutually exclusive, meaning that the phone cannot have both features at the same time. The programmer may not like any of these three alternatives.

Figure 10 shows the essential programs of the feature SelectiveCallForwarding. On the interaction condition of DoNotDisturb and CallForwarding, the program unit SelectToForward will check whether the caller is in a phoneIDlist. If he is, the call is forwarded. If not, then do not disturb.

```java
feature package SelectiveCallForwarding {
    domain BasicTelephony;
    Router rt;
    Phone fone;
    features DoNotDisturb, CallForwarding(rt), Pots (fone, rt), CatchAll;
    priorityPrecedence (DoNotDisturb, CallForwarding, Pots, CatchAll);
    LinkedList phoneIDlist = new LinkedList (empty); // forwardable phones
    // constructor not shown

    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;
        }
    }
}
```

Figure 10
In the example, the program unit **SelectToForward** does not explicitly call the program units, **ForwardCall** of **CallForwarding** and **SayBusy** of **DoNotDisturb**. Instead, it refers to the features. The compiler generates code to invoke the correct program units. Alternatively, the programmer can call the program units directly using the notation **CallForwarding.ForwardCall** and **DoNotDisturb.SayBusy**. The compiler then checks whether the program units are called “within context”.

Both a program unit and a precedence list are used in the feature package. In FLE, the program unit overrides the precedence lists when the condition part of the program unit becomes true.

If the **QuietPhone** feature package of Figure 8 has already been coded, the programmer can choose to combine **QuietPhone** with **CallForwarding**. The feature package **SelectiveCallForwarding2** shown in Figure 11 functions exactly the same as **SelectiveCallForwarding1**. The reader may note that the same program unit is used in both feature packages.

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

    /* phoneIDlist stores the list of phone numbers that calls from them should
    be forwarded. */
    LinkedList phoneIDlist = new LinkedList (empty);
    /* constructor not shown */

    SelectToForward {
        condition: state.equals ("idle");
        event: TerminationRequest e; {
            if (phoneIDlist.contains (e.FromPhoneID)) {
                CallForwarding;
            } else {
                DoNotDisturb;
            }
            stop;
        }
    }
}
```

**Figure 11**

5. Related Work

The heritage of FLE comes from AI languages that support nonprocedural execution of program units. What we called **feature interaction** is **conflicts** in rule-based languages. Because of the differences in intended usage, the structure and facilities provided by most AI languages are quite different from us. AI languages do not require explicit conflict detection and resolution before programs can be executed. Programmers using them typically do not know before hand what program units may interact with each
other and their ability to resolve conflicts is relatively limited. Jackson [16] gives a very good coverage on the most important AI languages.

Two programming languages, SDL [17] and VFSM [18], both allow the programmer to specify finite state machines with nonprocedural program units. But neither of them supports automatic feature interaction detection nor provides language facilities for interaction resolution. Adding features to state machines written in both languages require changing the original state machines.

Statechart [20], a component of UML [19], provides graphical representation to model the behavior of software objects. It is sufficiently formal to generate code. Again, given a statechart representation of a software object, addition of features is done by changing the statechart.

Researchers attacking the feature interaction problem have taken two basic approaches: via design methods and via more formal and analytical methods. The work of Jackson and Zave on “Distributed Feature Composition” [30] is representative of the design approach. They propose the use of a pipe-and-filter architecture to integrate features.

For those who take the more formal approach, a number of authors had suggested that feature interaction detection is a satisfiability problem (e.g. [21] and [22]). A large number of papers propose the use of formal means from algebraic (e.g. [23]) to Petri nets (e.g. [24]) for the specification and detection of feature interaction. The more pragmatic researchers use available tools. Faci and Logrippo [25] use LOTOS [26]; Amyot et al [27] use both Use Case Maps [28] and LOTOS.

FLE supports a formal method to detect interaction condition as well as a design paradigm leading to reusable features.

We follow the example of C++ [15] which added advanced language features to C; we added FLE extensions to Java.

6. Conclusions

We design FLE with these objectives in mind: First, to enable the programmer to write a new feature without the manual and error prone process of examining the code of its interacting features; second, to enable the programmer to put features together without changing code making feature programs reusable. Meeting these two objectives leads to reusable feature code.

FLE meets the first goal by supporting a design paradigm in which the programmer designs the feature following a model instead of the code of other features. In addition, FLE supports an automatic procedure to detect interaction conditions. FLE meets the second goal by supporting nonprocedural programming. The programmer is not responsible for specifying the execution sequences of program units, and he does not need to change code to effect change in control flow to accommodate the new feature. Further, FLE provides mechanisms such as feature package and precedence lists so that features can be integrated and have their interaction resolved without changing code.
Meeting these two objectives does not guarantee the correctness of FLE features and feature packages. We are working on computer aided design and verification tools for FLE programs. One can show that an executable FLE program is a finite state machine even if its condition space as defined in the domain statement is unbounded. This makes FLE programs particularly amenable for analysis by a model checker such as SPIN [29].

The generated code of a FLE program looks a lot like how one may write the program in a conventional programming language. We therefore suggest that the performance of FLE programs will be comparable to those written in a conventional object oriented language. The FLE code written for the prototype is several times less than its generated code. This is partly due to the nonprocedural nature of the language and partly due to short hands, such as the keywords all and any, supported by the language. We will report these results when we have more data.

Due to space limitation, we are not able to describe a number of results associated with FLE including the satisfiability solver to detect interaction, inheritance method (the method to extend an existing model) and exception handling method. They will be reported separately.

7. References


