On the Verifiability of Programs Written in the Feature Language Extensions

Wu-Hon F. Leung
Computer Science Department, IIT
leung@iit.edu

Abstract

High assurance in embedded system software is difficult to attain. Verification relies on testing. The unreliable and costly testing process is made much worse because the software base constantly changes: Adding a feature is by changing the code of other features, and the programs of the features entangle in the same reusable program unit of the programming language. For a large class of applications, including those requiring exception handling, this entanglement problem cannot be solved using existing general purpose programming languages. The Feature Language Extensions (FLX) is a set of language constructs designed to enable the programmer to solve the entanglement problem. It provides language support for assertion based verification. The satisfiability of first order assertions composed of variables defined by FLX can be determined without iterations of trials and errors. An executable FLX program is compiled into a finite state machine even if the state variables are unbounded.

1. Introduction

High assurance in embedded system software is difficult to attain. Its verification relies on instance by instance testing. The costly and unreliable testing process is made much worse because the software base constantly changes as new features are added. Changing code is labor intensive and error prone. When a feature, which denotes a functionality of the software, is developed by changing the programs of other features, the programs of the different features *entangle* in the same reusable program unit (such as a method) of the programming language. The entanglement often scatters into many program modules. It also makes the feature programs difficult to maintain, reuse and adapt to different user needs.

The problem of program entanglement is best illustrated with an example. With existing general purpose programming languages, the programs of normal processing and exception handling features are entangled. If a device driver needs to throw a new exception, all the programs that directly or transitively call the device driver may have to be changed. In larger development projects, this often means manually reviewing millions of lines of code.

Java offers a partial solution as it can identify the programs that need to be changed if its compiler happens to type check the new exception. Most other languages offer no help. The programmer either has to manually go through a large amount of code, or leaves the exception uncaught. Uncaught exceptions will crash the application and even the system. It happens often in some popular software. But that is not acceptable for high assurance software.

A solution to high assurance software should, therefore, enable the programmer to develop the programs of a feature without entangling with the programs of other features, and to verify his software formally based on assertion instead of instance by instance testing. These are the design objectives of the Feature Language Extensions (FLX). FLX is a set of programming language constructs with an implementation on Java. The implementation is analogous to C++ which added object oriented programming language constructs to C.

In an earlier paper, we showed that the program entanglement problem cannot be solved using existing general purpose programming languages for a large class of applications [17]. A main reason is that these languages require the programmer to specify execution flows. FLX supports nonprocedural programming that does not ask the programmer to specify the execution flows of program units. It provides language constructs for the programmer to specify a *feature* and to integrate features into *feature packages*.

There has been significant advances in the art of verifying computer systems. It is becoming routine to apply assertion based verifier to hardware design. However hardware designs are mainly composed of finite state machines and its assertions are Boolean formulas (as its variables are binary variables). But in software, a state variable may be unbound (such as when it is an integer), and one must reason on predicate logic asking questions such as whether a link list is empty. Verification of software systems is therefore much more difficult.

Our approach is to use programming language design to reduce the complexity. An executable FLX program is compiled into a finite state machine with relatively small number of states even if the state variables used in the program are unbounded. Secondly, FLX provides language facilities for the programmer to provide semantic input such that determining the satisfiability of first order predicate formulas composed of variables defined using FLX does not require iterations of trials and errors.

The paper is organized as follows. In section 2, we discuss the nature of the program entanglement problem and the challenges of applying assertion based verification to software. The foundation constructs of FLX are overviewed with examples in section 3. The language support and algorithm to determine the satisfiability of first order formulas written in FLX are described in section 4, as well as the fact that an executable FLX program is a finite state machine. The paper concludes in section 5.

2. Program Entanglement and Verifiability

2.1 Program Entanglement and Feature Interaction

The program entanglement problem is related to the notion of *feature interaction*. Two features *interact* if their *behavior* changes when they are integrated together. Features are implemented by computer programs, and for the purpose of this paper, the behavior of a computer program is manifested in its output and the sequence of program statements that gets executed for a given input. The term feature interaction was first introduced by developers of telecommunications systems [13] to describe circumstances like when a phone is called, the programs of the plain old telephone service (POTS) will ring the phone, but if call forwarding is added, the combined program will give a ping-ring then forwards the call to another phone. The concept is common place and not confined to telecommunication software.

Feature interaction is common in embedded systems. Take the Internet TCP protocol [23] as an example. Before its congestion control feature is developed, a duplicated acknowledgement will prompt its reliable data transport feature to retransmit. After congestion control is added, the same message may cause the sender 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.

Feature interaction and program entanglement is related in the following way [17]: If (C1) two features interact, (C2) they are executed in the same sequential process, and (C3) the implementation programming language requires the programmer to specify execution flows, then their programs will entangle. If the two features do not interact, it is not necessary that their programs entangle. In other words, feature interaction is a main reason for program entanglement.

The above conditions imply that the entanglement problem cannot be solved by software design alone. The programs of TCP are notoriously entangled (e.g. see [21]) and have frustrated many efforts to improve them. It is not because their programmers lacked skill; they could not help it. The entanglement conditions also explain why existing general purpose programming languages cannot separate normal processing and exception handling features. C1 and C2 are often dictated by the application such as in the case of TCP and in exception handling. Changing C3 is then essential in solving the program entanglement problem.

We call the conditions under which the behavior of two interacting features will change their *interaction conditions*. Presently, the programmer must examine code line by line to determine when the conditions become true and *resolves* the interaction by changing feature code to specify the new behavior. Because the features are integrated by changing the code of one another, they are not easily separable and are not reusable without one another. A solution to the program entanglement problem should therefore meet these requirements: (R1) The programmer can develop a feature independent its interacting features; (R2) There is a tool that can identify the interaction conditions automatically; (R3) The features can be integrated and with their interaction condition resolved without requiring changing code; and (R4) The features can be reused independent of other features.

Language facilities such as *macros* in C and *Aspects* in AspectJ [15] separate the code of different features textually but do not meet the above requirements. For example, with AspectJ, the programmer in general must go through the base code and the code of other Aspects to determine where the joint points are. Often, they need to change code to make the join points apparent. Some had argued that separating code this way is detrimental [1]. Empirical studies conducted over the years (e.g. [21] and [11]) have shown that aspect oriented programming (AOP) have not meaningfully improved programmer productivity even though some have shown that it can significantly reduce the number of lines of code to be written (e.g. [18]).

Recently, Service Oriented Architecture (SOA) proposes to organize each service as a process. These processes interact by requesting and providing services to one another [5]. SOA thus relaxes the entanglement condition C2. But it is not clear that it will lead to adding new services without requiring changes in other services. An analysis given in [6] showed that service invocations in such a system exhibit a fractal structure (a condition that may lead the system to become chaotic) with significant complexity and performance implications.

FLX relaxes C3 and supports nonprocedural programming. A *program unit* in FLX 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 order of the program units. A *feature* is composed of a set of program units. With FLX, the programmer develops a *feature* following a model instead of the code of other features. FLX provides a tool to detect interaction conditions among features. Features are integrated in a *feature package* and have their interaction resolved without requiring changing code. Features and feature packages are reusable as different combinations of them may be integrated and have their interactions resolved differently to meet different user needs.

2.2 Automatic Formal Software Verification

Advances in the model checking (e.g. see [8]) technology and in satisfiability (SAT) solvers of Boolean formulas (e.g. see [20]) are mainly responsible for the practical application of assertion based verification of hardware designs. A model checker systematically and exhaustively explores the state space of a concurrent system to check for violation of formally specified assertions. A SAT solver determines whether there is a satisfying assignment to the variables of a logical formula. Efficient SAT solvers can greatly (exponentially) improve the efficiency of model checkers [19].

But software verification continues to rely on testing. As discussed earlier, a condition variable in software may be unbounded and one must reason on predicates of complex data structures. Consequently, earlier results in assertion based software verification apply only to an *abstraction* of the actual software. The abstraction is done manually, translating complex software into a simple model expressed in the input language of the verification tool. The abstraction itself is a source of error and can rarely keep up with changes in the actual software.

More recently, a number of research groups developed model checking tools and applied them directly to real software programs. They have taken different approaches. Bandera [10] and SLAM [2] automate program abstraction using program slicing and predicate abstraction techniques respectively. Java PathFinder [26] translates a Java program to the input language of the pioneering model checker SPIN [14]. VeriSoft[12], and CMC [21]) are highly optimized model checkers that integrates with the software to be verified. SLAM is now a commercial product. CMC reported to have verified software subsystems with tens of thousands of lines of code. But all of them also reported significant limitations. For example, CMC reported verification of an implementation of TCP but not the properties of some of its most complex features such as congestion control.

The root cause of the limitation is the *state explosion problem*: the exponential increase in the state space that the model checker must explore as the number of state variables in a program and their value set increase. The problem is becoming a limiting factor even for hardware verifiers as the complexity of hardware circuits grow. But it is much harder in software. Existing model checkers for software are highly optimized and some of their effort to compress the state space are heroic (e.g. see [21]). It will not be sufficient just to keep on improving model checking algorithms.

FLX uses programming language design to reduce the state space: an executable FLX program is a finite state machine; the number of states is proportional to the number of program units in the program. This result will be described in Section 4.

Another approach to increase the capability of model checkers for software verification is to incorporate an efficient SAT solver. This SAT solver must be capable of handling first order predicate logic.

The first order SAT solver of FLX plays additional important roles besides its usage in verification. It identifies interaction conditions and participates in code generation. Its performance is, therefore, critically important and it must analyze first order formulas coming directly from FLX programs.

The problem of determining the satisfiability of first order predicate formulas is in general undecidable [9]. Most first order SAT solvers, including ours, therefore work on a decidable subset of first order formulas. The main difficulty for first order SAT solver is due to the fact that the values of the variables in a first order formula have large ranges and may even be unbounded. Recent results on first order SAT solver take two basic approaches: *instance method* or *predicate abstraction*.

The basic ideas for instance methods is to first assign some values to the variables of a first order formula transforming it to a propositional formula, and then use a Boolean SAT algorithm to determine whether the now *instantiated* formula is satisfiable.

This is basically a trials and errors procedure to search for a satisfying assignment. Although many trials can be conducted in parallel and the searching is systematic, the search space (a Cartesian product of the values of the variables) is huge for nontrivial formulas. To reduce the search space, the search algorithms of partial instantiation methods may branch on partially instantiated formulas (e.g. [3]). Plaisted and his colleagues devised a number of methods that allow the user to provide guidance on the instantiation of the variables (e.g. [22]).

There are four general steps in the predicate abstraction method. The first step is to transform the first order formula α to its conjunctive normal form (CNF). In step two, syntactically identical predicates in α are replaced by a Boolean variable, obtaining a propositional formula $B(\alpha)$. Step three uses a Boolean SAT solver to determine whether $B(\alpha)$ is satisfiable. If it is not, α is not satisfiable. If it is, the satisfiable condition γ obtained from the Boolean SAT solver is used to test whether α is satisfiable. If it is, α is satisfiable. If not, then we go to stepwise refinement of setting $B(\alpha) = B(\alpha) \wedge \neg\gamma$ and return to step three. SLAM uses this method to obtain a Boolean abstraction of the program under analysis before model checking.

Both partial instantiation and predicate abstraction methods require iterations of solving NP complete problems. In the worst cases, the number of iterations can be exponential to the number of literals[1] in the first order formula being analyzed.

FLX provides language constructs and rules for the programmer to provide semantic guidance to its first order SAT solver. The semantic guidance is a decision procedure (instead of variable instantiation as proposed in [22]). While the complexity of the FLX first order SAT solver is still NP complete, it does not require iterations of trials and errors. The basic algorithm of the FLX SAT solver is described in section 4.

3. The Foundation Constructs of FLX

A FLX *program unit* consists of a *condition part* and a *program body part*. The program body gets executed when its corresponding condition part becomes true. FLX is event driven: the evaluation of program unit condition parts is triggered by events, as the primary input of many embedded system applications are random and short-lived events such as in telecommunication systems, sensor networks and in the kernel of operating systems.

A FLX *feature* contains a set of program units that perform the functionality of a feature. A feature is developed according to a *model*, which defines the condition space and the basic functionality of the application. The condition space is specified in a *domain statement*. The basic functionality is specified in a feature called an *anchor feature*. Features designed according to an anchor feature can be considered as an extension or enhancement of the anchor feature.

Features are integrated in a *feature package* without requiring modification. The programmer may package different combinations of features in a feature package, or he may change the way the integration works in different feature packages to meet different user needs. For example, he may choose to use different **Retry** features on platforms equipped with different redundancy.

We will use programs from a telephony system implemented using FLX to illustrate the usage of the basic FLX constructs.

Each phone object in the telephony 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 forwarding). Different phone objects can have different sets of features in their feature packages.

The domain statement of the call processing feature package declares the *domain variable* **state** and a set of *events* that will be used in the condition part of a program unit. A domain variable is of a *domain data type* which must contain public predicate methods and/or Boolean members. It is extended from a Java class with the addition of a *combination function*, needed to support the first order SAT solver of FLX. The domain variable **state** is declared to be of the domain data type **Denum** which is extended from the Java **enum** class. It has values like **IDLE, RINGING, TALKING** and so on. In the digit analysis feature packages, we use condition variables with data type extended from Java Integer which is unbounded. FLX is not limited to defining finite state machines. The domain statement for the call processing features is shown in Figure 1.

The domain statement in Figure 1 also declares a set of *resources* that the features using this domain statement will operate on. When a feature package that uses the domain statement is instantiated, the references to the resources, in this case the phone **fone** and router **rt**, are passed to the feature package. The domain variable **state** is initialized in the domain statement. Space is allocated to it when the feature package is instantiated. Events are instantiated in feature programs when they are needed.

[1] A literal is either an atom or its negation in a logical formula. In a first order formula, an atom is either a Boolean variable or a predicate.

```
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;
     }
```

Figure 1 The Domain Statement for Call Processing

The anchor feature **POTS** is given in Figure 2 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 2. 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 first order formula composed of public Boolean members and predicates of domain variables. 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. We further require that a domain variable is not a function of other domain variables. The FLX compiler checks that the condition statement of at least one program unit in the anchor feature is true given the initial value of the domain variables.

A compiled anchor feature or feature package is executable. It is instantiated similar to an object but its program units are usually not called like the methods of an object. We call an instantiated anchor feature or feature package a *feature object*. The FLX compiler generates a number of standard methods for each feature object. One of them is the method **SendEvent(e)**. The method is called by other programs (it is also possible for itself) to send the event **e** to the feature object.

The feature **DoNotDisturb** is shown in Figure 3. Its program unit **SayBusy** returns a busy event whenever the phone receives a **TerminationRequest** event. A feature by itself is not executable. It needs to be integrated with its anchor feature in a feature package.

It can be shown that if the conjunction of the condition parts of two program units is satisfiable, the two program units interact. When the satisfiable condition, which is the interaction condition, becomes true, either program units may get executed. The programmer is required to remove, or resolve, the ambiguity. Two features interact if some of their program units interact. The first order SAT solver of FLX detects interaction conditions.

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

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

Figure 3. The feature `DoNotDisturb`

Figure 4 shows the code of the feature package **QuietPhone** integrating the features **POTS** and **DoNotDisturb**. The two features interact in all their program units triggered by the **TerminationRequest** message. The interaction is resolved by the **priorityPrecedence** statement with the following semantics: when an interaction condition becomes true, the program unit belonging to the feature with the highest precedence in the list will get executed. A more in depth discussion of using precedence lists to resolve interaction is given in [7].

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** gets invoked and the user can make phone calls.

```
feature package QuietPhone {
   domain: BasicTelephony;
   features: DoNotDisturb, POTS;

   priorityPrecedence (DoNotDisturb, POTS);
}
```

Figure 4. The `QuietPhone` feature package

This simple example shows that the two interacting features can be integrated together without changing each other's code. The feature resolution facilities provided by FLX are general. Besides using precedence lists, the programmer can use program units to resolve interaction for any specific condition. More complex examples of using FLX are given in [17], including those that uses the exception handling and inheritance constructs of FLX.

4. FLX Support for direct verification

4.1 FLX support for first order SAT solver

Determining the satisfiability of a first order formula in general is not decidable [9]. First order formulas from the condition parts of FLX program units are quantifier free and do not contain functional symbols. Determining the satisfiability of first order formulas with these properties is decidable [4] and similarly assumed by many other algorithms. Importantly, the variables of first order formulas from FLX programs are defined by abstract data types and the interpretations of their predicates are well understood by the programmer of the abstract data type.

Existing first order SAT solver methods use a trials and errors approach to search for a satisfying assignment. We avoid that by taking advantage of the knowledge of the programmer. We ask the programmer to associate a *combination function* class for each domain data type. The combination function takes a list of literals of the domain data type as argument, and returns whether the conjunction of the literals is satisfiable. The decision procedure for the combination function is typically well understood. For example, for the data type integer, the conjunction of a set of its predicates (greater than, equal to etc.) should establish a partial ordering of the variables. If not (e.g. if we have a > b AND b > a), the conjunction of the set of predicates is not satisfiable. We have not come across a Java class that we cannot readily come up with a decision procedure for its combination function.

Figure 5 shows the declaration of the combination function class for strict partial order predicates [24]. The combination function class is given a name (**StrictPartialOrder).** The list of predicates, namely **largerThan** and **lessThan**, that the combination function can handle, are given after the keyword **combines**. The combination function class contains exactly one method and the list of literals is passed to it as a set of strings.

```
Public combinationFunction StrictPartialOrder
        combines {largerThan, lessThan} {
   Public static Boolean combinationFunc
        (HashSet <string> group) {
        .......        //code
   }
}
```

Figure 5. The declaration of a combination function

A combination function can be associated with different domain data types. For example, the combination function for strict partial order predicates can be associated with a domain data type that defines a node in a PERT chart used in a project management application, or with one that defines a node in the syntax tree of a compiler. Figure 6 shows the declaration of a domain data type (a node in a PERT chart). It is simply a Java class with an addition declaration of its association to a combination function. The association is indicated with the keyword **uses**.

```
Public class PERTNode uses StrictPartialOrder
{
   .... // Data structure here
   public boolean largerThan (NodeInPERT v){ }
   public Boolean lessThan (NodeInPERT v) { }
   ....   // Other methods
}
```

Figure 6 The declaration of a domain data type

We are now in position to describe the basic algorithm of the FLX first order SAT solver. Given a first order formula from the condition part of a program unit, we first derive its disjunctive normal form (DNF). Each clause of the DNF is a conjunction of literals whose variables belong to different domain data types. Taking advantage of the associative property of the conjunction operator, we partition each clause into subgroups. Each subgroup contains only literals whose variables belong to the same domain data type. We then use the combination function of the domain data type to determine whether the subgroup is satisfiable. The clause is satisfiable if each subgroup is satisfiable. The formula is satisfiable if any clause of the DNF is satisfiable. When this algorithm is used to identify interaction conditions, the algorithm goes through all the clauses in the DNF to see whether they are satisfiable.

The above procedure is NP-Complete because deriving the DNF is NP-Complete. But once the DNF is derived, the algorithm requires no iterations of trials and errors. The algorithm was first described in [16], but this is the first time that it is reported. The

implementation is described in [25] together with several extensions including a modified algorithm to treat predicates that may contain variables of different types. There is also a more extensive discussion on related work.

4.2 An executable FLX program is a finite state machine

The domain variables and events declared in the domain statement define the state space of the application which can be unbounded as we allow the programmer to use unbounded domain data types, such as integers. But one can always discern a finite state machine from an executable FLX program.

The initial state is given by the initial values of the domain variables in the domain statement. An executable anchor feature or feature package contains finite number of program units. The condition part of each program unit defines a state space which is a subset of the state space defined in the domain statement. The state space of a program unit may have nonempty intersection with the state space of other program units if they interact. The first order SAT solver identifies these intersections. For the purpose of analyzing the executable FLX programs, we can count each of the nonempty intersections and their complements as distinct states. Hence, we have a finite number of states. State transition of the finite state machine is triggered by events. A program unit may update the value of some domain variables. If that happens, the next state is defined by change in values of the domain variables.

The nonempty intersections are the interaction conditions among program units and they are resolved either by another program unit or by a precedence list in the feature package. If we lump all intersections covered by a precedence list as one state, the number of states of the finite state machine is roughly equal to the number of program units in the feature package and its features plus the number of precedence lists in the feature package.

5. Conclusions

FLX has two design objectives: (1) to enable the development of interacting features as separate and reusable program modules, and (2) to facilitate assertion based verification of programs written in FLX. FLX meets the requirements for objective (1). For objective (2), we have developed a first order SAT solver and the FLX compiler generates a finite state machine from an executable FLX program.

About forty different features and feature packages were written in FLX for the telephony system described earlier. These features and feature packages were mainly developed as test cases for the compiler. A feature is typically developed and integrated with other features in a few days to a couple of weeks, a significant improvement compared to the author's experience from the industry.

We attribute the observed improvement to the fact that using FLX, one is able to focus on one feature at a time. While writing the programs for, say, call waiting, the programmer does not need to be concerned with designing hooks for three way calling and other features. Integration of features does not require going through and changing code. Interaction conditions are automatically detected and most are resolved by precedence lists. Interaction resolution is done in a single program module (a feature package) instead of scattering into many program modules.

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 nonprocedural nature of the language and partly due to short hands, such as the keywords **all**, supported by the language. Consider the DoNotDisturb feature given in Figure 3. Its code will have to be duplicated many times if the feature is written in a procedural language.

We started to use FLX to produce useful code. Recently, we used it to develop the essential features of a call center built on top of the voice over IP platform, Skype. We are in the process of using FLX to rewrite the scheduler of the Linux kernel, and have started the development of an assertion based verifier for programs written in FLX.

A research version of the FLX compiler, example code, FAQ and other documents are available for download at http://www.openflx.org.

6. Acknowledgment

The author wishes to acknowledge the contribution of Lu Zhao and Yimeng Li in the development of the current version of the FLX first order SAT solver, and Lu Zhao for her research into the prior art of first order SAT solvers. The full scope of their contribution will appear in a forthcoming paper.

7. References

[1] Alexander, R., "The Real Costs of Aspect-Oriented Programming," IEEE Software, November/December, 2003.

[2] Ball, T., and Rajamani, S. K., "The SLAM Project: Debugging System Software via Static Analysis," Proceedings of Principles of Programming Languages, 2002.

[3] Baumgartner, P. and Tinelli, C., "The Model Evolution Calculus," The 19th International Conference on Automated Deduction, Volume 2741 of Lecture Notes in Artificial Intelligence (2003).

[4] Bernays, P. and Schonfinkel, M., "Zum Entscheidungsproblem der mathematischen Logik," *Mathematische Annalen 99*: 342-72, 1929.

[5] Bloomberg, J., "The Lego Model of SOA," ZapThink, December 11, 2006; www.zapthink.com/report.

[6] Bussler, C., "The Fractal Nature of Web Services," *Computer*, March 2007.

[7] Chavan, A. et. Al., "Resolving Feature Interaction with Precedence Lists in the Feature Language Extensions," Proceedings of 9th International Conference in Feature Interactions, September, 2007.

[8] Clarke, E. M., Grumberg, O., Peled, D. A., "Model Checking," The MIT Press, 1999.

[9] Church, A., "A Note on the Entscheidungsproblem," Journal of Symblic Logic, 1(1936).

[10] Corbett, J. C., et al, "Bandera: Extracting finite-state models from Java source code," In Proc. 22nd International Conference on Software Engineering (ICSE), June 2000.

[11] Filho, F., C. Rubira, and A. Garcia, "A Quantitative Study on the Aspectization of Exception Handling," Proceedings of ECOOP Workshop on Exception Handling in OO Systems, July, 2005.

[12] Godefroid, P., "Model Checking for Programming Languages using VeriSoft," Proceedings of POPL 1997.

[13] Harr, J.A., E.S. Hoover, and R.B. Smith, Organization fo the No. 1 ESS Stored Program. The Bell System Technical Journal, 1964.

[14] Holzmann, G. J., "The SPIN Model Checker: Primer and Reference Manual," Addison-Wesley Professional, Septmeber, 2003.

[15] Kiczales, G., et al., "An Overview of AspectJ," Proceedings of European Conference on Object Oriented Programming (ECOOP 2001), Springer-Verlag, 2001.

[16] Leung, W.-H., "Method to Add New Software Features without Modifying Existing Code," United States patent application, August, 2002.

[17] Leung W. H., "Program Entanglement, Feature Interaction and the Feature Language Extensions," *Computer Networks*, February, 2007 issue 2.

[18] 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.

[19] McMillan, K. L., "Applying SAT Methods in Unbounded Symbolic Model Checking," Proceedings of 14th International Conference on Computer Aided Verification, July, 2002.

[20] Moskewicz, M. W., et al, "Chaff: Engineering an Efficient SAT Solver," Proceedings of the 38th Design Automation Conference (DAC 2001).

[21] Murphy, G.C. et. Al., "Evaluating Emerging Software Development Technologies: Lesson Learned from Assessing Aspect-Oriented Programming," IEEE Transactions on Software Engineering 25 (4) 1999.

[21] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill, "CMC: A pragmatic approach to model checking real code," Usenix Association, OSDI 2002.

[22] Plaisted, D. and Zhu, Y., "Ordered Semantic Hyper Linking," Proceeding of 14th National Conference on Artificial Intelligence, 1997.

[23] Postel, J., Transmission Control Protocol, RFC 793, Sept. 1981. http://www.refceditor.org/rfc793.txt

[24] http://en.wikipedia.org/wiki/Partial_order

[25] Zhao, L., "A First Order Satisfiability Solver for the Feature Language Extensions," M.S. these, ECE Department, IIT, May, 2006.

[26] Lindstrom, P. et. al., "Model Checking Real Time Java Using JavaPathfinder", Proceedings of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA), October, 2005.