Run-Time Assertion Checking with Énfasis

Verificación de Aseveraciones a Tiempo de Ejecución con Énfasis

José Oscar Olmedo Aguirre and Ulises Juárez Martínez
Department of Electrical Engineering, CINVESTAV – IPN
México 07360, D. F.
oolmedo@cs.cinvestav.mx, ujuarez@computacion.cs.cinvestav.mx

Article received on July 31, 2008; accepted on December 04, 2008

Abstract
Local variables are fundamental to describe and implement computer algorithms and to specify some of their properties such as correctness, termination and performance. In this paper, we address the run-time assertion checking problem involving local variables in Java programs using Énfasis. Énfasis is a novel Aspect-Oriented Programming (AOP) language that introduces a join point model for crosscutting on local variables and path expressions to select sets of join points. The contribution of this work consists on showing that run-time assertion checking is more effective in Énfasis by its greater expressive power than in other approaches such as AspectJ and the interface specification language JML. The greater expressiveness of Énfasis assertions arises from the finer granularity of crosscutting, defined at the expression and statement level in contrast to the coarser granularity defined at the method invocation level of AspectJ. Énfasis approach characterizes not only by its non-invasiveness at the source code level, modularity, uniformity and generality are also improved in handling orthogonal crosscutting concerns.
Keywords: Program assertion, formal grammars, local variable crosscutting, local variable pointcut, aspect-oriented programming.

Resumen
Las variables locales son fundamentales para describir e implementar algoritmos y para especificar algunas de sus propiedades tales como correctitud, terminación y desempeño. En este artículo se discute la verificación de aseveraciones a tiempo de ejecución en programas Java utilizando Énfasis. Énfasis es un lenguaje orientado a aspectos que incorpora un modelo de puntos de unión para aplicar corte sobre variables locales y expresiones de rutas para seleccionar conjuntos de puntos de unión. La contribución de este trabajo es mostrar que la verificación de aseveraciones a tiempo de ejecución es más efectiva en Énfasis por su gran poder expresivo respecto a otros enfoques como AspectJ y JML. La expresividad de las aseveraciones en Énfasis se debe a la granularidad fina de los cortes que se definen a nivel de expresiones y sentencias, a diferencia de otras granularidades más gruesas definidas a nivel de invocaciones de métodos como en AspectJ. El enfoque de Énfasis no solo destaca por su no invasividad del código fuente, modularidad, uniformidad y generalidad, sino también por el manejo ortogonal de incumbencias de corte.
Palabras clave: Aserciones de programas, gramáticas formales, corte en variables locales, puntos de corte en variables locales, programación orientada a aspectos.

1 Introduction

Run-time enforcement of formal program specifications leads to the construction of more reliable software. A formal specification is an abstract description of the intended purpose and behavior of a software system. It is defined according to logical or mathematical models that capture unambiguously the most relevant system properties. In practice, formal specifications, called *assertions*, are conditions introduced at specific points in the program where they are expected to hold. After the notable success of the model checking techniques applied to the software verification problem, run-time enforcement has recently acquired more attention over other theorem-proving and compilation-time verification methods. The importance of assertions for the programming practice is manifold: (1) by enforcing the application of systematic and rigorous programming development methods, (2) by showing that the program implementation follows from the program specification, (3) by bringing in automated programming assistants that help to check the program specifications, and (4) by producing an unambiguous, precise and high-quality documentation.

Assertion specifications were originated from the work of Floyd (1976) and Hoare (1969) and introduced in Object-Oriented Programming by Meyer (1991) as the *design by contract* metaphor. In this metaphor, the implementer and the user have agreed (through their code) a contract in which the user obliges himself in invoking the implementer services only if his request satisfy the contract (pre-)conditions, and in return the implementer will provide the service to the client as stipulated in the contract (post-)conditions. In providing his services, the implementer develops his behavior in a way that ensures that some guarantees (invariants) of reliability are always met. These principles ensure not only that at any moment the implementer's code correctly attends the incoming requests, but also that his stable and well-behaved performance guarantees that the contract responsibilities will be always met.

However and in spite of their importance, assertion verification is very often neglected and considered only when some program defects are found. From this perspective, program verification is considered as run-time assertion enforcement, a crosscutting concern not considered very often as part of the program design and implementation. Crosscutting concerns are not part of the core program functionality because they appear scattered and tangled across the entire system code. Modularizing crosscutting concerns then becomes fundamental to isolate the assertion from the program points in which they appear and from the condition implementation to be executed.

Historically assertion checking has been approached in a variety of forms ranging in sophistication from simple tracing statements to program annotations, among others. Tracing statements give complete control to the programmer to retrieve information during the program execution. However they require modifying the source program for each concern, a code invasive strategy that leads to cumbersome versioning management problems. In contrast, program annotations are relatively easier to manage by pre-processing tools, but nowadays annotations still do not provide the means to inspect the program state with enough detail because they can be only applied at very restricted parts of the program text.

1.1. Aspect-oriented programming

Aspect-Oriented Programming (AOP) [Kiczales, 1997] is a new programming paradigm originated from the need to modularize non-invasive crosscutting behavior. AOP brings in new levels of modularity to separate design concerns to better the construction and maintainability of software systems. AOP focuses in a kind of modularization that is difficult to conceptualize through standard Object-Oriented Programming (OOP) languages and techniques. This kind of modularization generalizes the simple method invocation pattern of OOP to more complex patterns that enable the execution of reactive code when a specific pattern of runtime situations is detected [Cilia, et. al., 2003]. Among the more common situations are field access, method call and execution, and exception throwing. Many crosscutting concerns are associated to the occurrence of these situations.

Behavioral pattern descriptions and aspectual behavior are grouped in aspects [Kiczales, 2001]. An *aspect* is a modular unit for implementing crosscutting concerns that consists of pointcut descriptors, advice definitions and member class introductions. Its organization is similar to a Java class containing fields, constructors and methods. A *join point* is a well-defined program segment that represents a crosscutting concern and upon which are implemented mechanisms to interleave the advice code during program execution. Among the more common join points are those for field access, constructors, method call and execution, and exception handling. A *pointcut* is an abstract description of a join point pattern that is observed during program execution. The pointcut specification detects those matching join points that satisfy the pointcut specification according to the run-time values of classes, objects, methods, fields, parameters, and local variables. An *advice* dynamically modifies the behavior of the class methods that it advices by crosscutting. An advice specification defines code intended to be executed when a matching join point is detected during and interleaved with the program execution. Execution of advice code must be ensured by an AOP system whenever a join point is encountered that match the pointcut specification for that advice. Interleaved execution of advice code with the program code is called *aspect weaving*. Aspect weaving can be approached *statically* (either at compile-time or load-time) and *dynamically* (at runtime) [Hilsdale and Hugunin, 2004; Nicoara and Alonso, 2005].

There are three basic types of advice, according to their relative inter-leavings when they execute. When a matching join point is detected, the *advice before* is executed before the join point code is executed. If the join point refers to a method call, the advice is executed before the method starts executing. When a matching join point is detected, the *advice after* is executed after the join point code is executed. If the join point refers to a method call, the advice is executed immediately after the method terminates its execution. There are in turn two variants of after advice in relation to either its normal or abnormal termination. In the latter case, the advice code is executed only when the exception thrown is of the same type. Finally, when a matching join point is detected, the advice code is executed instead of the join point code. If the join point refers to a method call, the around advice may proceed to execute the original method code as part of the new behavior by calling to the `proceed()` method. An *introduction* can be used to add new fields, constructors or methods into classes and interfaces. An introduction changes the static type structure of the class or the interface.

AspectJ can be considered as an aspect-oriented extension of Java implemented on it [Kiczales, et. al., 2001; Harbulot and Gurd, 2005]. AspectJ interleaves aspect code with the base code at compile-time. AspectJ supports both forms of crosscutting: static crosscutting that adds functionality to existing classes and dynamic crosscutting that provides additional functionality at the join points. In AspectJ, coarse-grained join points are supported but fine-grained join points are only supported in method invocations.

1.2. AOP and assertion checking

Assertion checking can be approached from the AOP programming language perspective. It can be considered as a detection and reaction behavioral mechanism. Detection refers to identifying the joinpoints in the base code where the assertion is expected to hold and reaction to executing the assertion checking aspectual code. Detection and reaction mechanisms can vary from non-invasive to invasive code inserted in the base code. While non-invasive mechanisms are always preferred, non-invasive mechanisms at bytecode-level render highly inefficient due to the need of incessantly observing any change in the program state to detect the points of interest. Instead non-invasive source-level mechanisms can be used to retrieve and pass contextual information coming from the program points to the assertion code.

For example, verifying that a sorting method produces an array with its elements arranged in ascending order is a crosscutting concern about the method correctness. Thus, instead of inserting the ascending order checking code at the end of every sorting method, it is possible to obtain the same effect if a call to a sorting method is detected, and then execute the assertion checking code, upon the completion of the method execution. In this work, we are interested not only in specifying assertions at the program points where they are expected to hold, but also in describing this aspectual behavior in a non-invasive manner, with no modification of the original source code.

This strategy could be applied to the `insertionSort()` method of class `SortingAndSearching` shown in Figure 1.

```
public class SortingAndSearching {
  public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
      int j, v = a[i];
      for (j = i-1; 0 <= j && v <= a[j]; j--)
        a[j+1] = a[j];
      a[j+1] = v;
    }
  }
...
```

Fig. 1. SortingAndSearching java class with method insertionSort()

In that follows, we analyze this problem by weaving aspects in Énfasis [Juárez and Olmedo, 2008], a recent proposal of a non-invasive AOP language with fine-grain crosscutting that includes crosscutting on local variables. The joinpoint model of Énfasis describes the exact position of the joinpoint in the source program where the aspectual code is intended to be inserted no matter how deeply nested the joinpoint could be defined.

For the run-time assertion checking problem, the approach adopted in Énfasis is more effective by its finer granularity and greater expressive power than in other approaches like those adopted in AspectJ and in the interface specification language JML, as discussed next.

The coarse crosscutting granularity of AspectJ precludes the precise placement of assertions before or after of any kind of statement. For example, termination condition of a **`for`** statement must be placed immediately after the statement to check that the condition became false upon the loop completion, but AspectJ coarse granularity is unable to describe this join point of interest. In contrast, the finer granularity model of Énfasis makes possible to define a joinpoint before or after any statement including **`for`** statements. Overcoming coarse granularity is important because assertions need to be defined arbitrarily depth in recursive compound statements. Furthermore, AspectJ neglects the importance of local variables in its joinpoint model, leaving out the possibility of expressing assertion conditions involving local variables which are the most common kind of assertions in program assertion verification.

JML [Leavens et al., 1999] is an interface behavioral specification language (BISL) that contains a formal notation to introduce the DbC approach [Meyer, 1991] in Java. JML uses lightweight and heavyweight annotations to describe the expected behavior of the methods of a class. A BISL describes both the details of a module's interface with clients and its behavior from the client's point of view [Hoare, 1969]. By using a BISL, it is possible to formally specify both the behavior and the exact interface of Java program modules, which is an essential step towards their formal verification. As JML is introduced into the source program by annotations, it suffers from the same disadvantages, among annotation name conflicts, incomplete pass of information from the source text to the assertion tool, and difficulties in reading and maintaining unrelated annotations. Énfasis solve this problem by:

- keeping encapsulated aspect name components,
- giving the programmer complete control on the source program information, and
- maintaining unrelated annotations separated from the source program and from each other.

Being an assertion checking tool, JML outperforms Énfasis in its program verification capabilities. Nonetheless, by equipping Énfasis with appropriate language compilers and interpreters, automated theorem provers, model checkers, static program analyzers and other programming language tools, it may reproduce closer the outstanding JML results.

However, there are a number of disadvantages that remains to be further solved in Énfasis too:

- Decoupling assertions from base program reduces readability and programming tools are still required to deal with this problem
- Dependencies on the lexical and syntactical structure of the base program and the Énfasis descriptions.

This paper is organized as follows. In Section 2 the assertion checking problem is stated and analyzed in relation to the design decision of embedding assertions directly into source code. In Section 3 it is explained how assertions can be expressed as aspects. The programming model is presented in Section 4 and the crosscutting programming model in Section 5. In Section 6 the relationship between weaving mechanism and assertion verification is given. Related work and conclusions are shown in Section 7 and 8 respectively.

2 The Run-Time Assertion Checking Problem

Originally, Java did not provide native support for assertions until version 1.4 [Gosling, et. al., 2005]. However, embedding assertion directly into the program code may be inconvenient because changing the assertion condition may alter the program behavior. The assertion describes some program property that is satisfied at a specific point within the program text. In consequence, assertion checking requires specifying both the assertion condition and the point in the program text where it holds. In that follows, a number of solutions to this problem are presented, beginning with a program annotated with assertions in a tailored assertion language, and ending with the original source program along with the corresponding assertions placed separately in a *descriptor program* that includes linking information that relates the assertions to the points where they hold in the source program.

2.1. Assertion specification

Design by contract (DbC) [Meyer, 1991] is a programming approach to design software components that must implement behavioral interface specifications (BISL). DbC assigns responsibilities to both sides of a method call interaction, the client's caller-code and the implementer's called-code, following a business contract metaphor. As explained before, the client's code must guarantee that the pre-conditions specified in the interface hold before the method is called. In return the implementer's code ensures that the corresponding post-conditions specified in the interface hold after the method call. The DbC is a refinement from the original concept of Hoare triple, where pre- and post-conditions provide a very precise description of both the implementer's code and the client's call. The Hoare triple for the `insertionSort()` method could be written as follows:

```
              {a != null && a.length > 1}
                 insertionSort(int[] a)
      {A int k; 0 <= k && k < n-1 ==> a[k] <= a[k+1]}
```

The Hoare triple affirms that if a client's call to `insertionSort(int[] a)` provides a non-null array `a` as parameter with more than one element, the implementer's code ensures that the elements of the array will be sorted in ascending order after the method terminates. The precondition `{a != null && a.length > 1}` and the postcondition `{A int k; 0 <= k && k < n-1 ==> a[k] <= a[k+1]}` are written in a tailored assertion specification language that extends Java with additional operators, connectives and quantifiers. An introduction to this language and to its semantics is given in section 4, but for now some notation and intuition will be sufficient to state the assertion checking problem.

The fundamental concepts of pre- and post-conditions were introduced by Hoare (1969) based on the earlier work of Floyd (1967). Hoare introduced these concepts in the form a logical system (also known as Floyd-Hoare logic) to reason about the correctness of computer programs. The *correctness of a program* is asserted when it is correct with respect to the specification of its pre- and post-conditions. Pre- and post-conditions are related to a well-formed program J in a *Hoare triple* $\{P\}J\{Q\}$. *Partial correctness* affirms that Hoare triple $\{P\}J\{Q\}$ is true if pre-condition P holds in a state σ where program J starts its execution, and assuming its termination, the program reaches a state σ' where post-condition Q holds. *Total correctness* is more stringent than partial correctness and requires in addition proving program termination. An *invariant* condition I holds before and after the execution of program J, occurring in both pre- and postconditions $\{P \wedge I\}J\{Q \wedge I\}$. A *loop-invariant* is a condition that holds through the successive iterations of an iterative statement. A deterministic program J always reaches exactly one state σ' for the same initial state σ.

In Figure 2, assertions have been introduced in the source program in a form that closely follows the proposed by Hoare. The assertions are written as comments, surrounded by brackets and preceded by a label (beginning with A), in the program points where they are expected to hold. For example, `A1` labels the method precondition, whereas `A6` the postcondition. Intuitively, a label represents the location within the method code that marks a point of interest; its precise meaning for the model will be introduced in section 3. Notice, that there are other labels in Figure 2, some of them beginning with `G` (getter/reader) and other with `S` (setter/writer) to designate the points of interest for local variables when they respectively appear in the left or in the right hand side of an assignment. For example, `G1` and `G4` are the points where local variable `i` is accessed for reading, whereas `S6` and `S7` are the points where element `a[j+1]` is accessed for writing (depending on the value of `j`). For reasons of readability not all labels were annotated.

The assertions of Figure 2 describe method correctness as a crosscutting concern, upon which may be added some others concerns later on, among program termination and time-complexity estimation to name a few. However, for the sake of simplicity we restrict ourselves to partial correctness only, though the results and conclusions can be extended to other concerns in a similar way. Briefly, insertion sort requires a non-null array as input parameter with more than one element (line 2.a) and ensures that their elements will be sorted in ascending order upon its completion (line 8.a). The method splits the input array `a` in sorted and unsorted parts. The sorted part has elements with index ranging from `0` to `i-1`, whereas the unsorted part has elements with index ranging from `i` to `n-1`.

```
02  public static void insertionSort(int[] a) {
02a   //A1: {a != null && a.length > 1}
03    for (/*S1:*/int i = 1; /*G1:*/i < /*G2:*/a.length; /*S2:*/i++) {
04      int j, /*S3:*/v = /*G3:*/a[/*G4:*/i];
04a     //A2: {A int k; 0 <= k && k < i ==> a[k] <= a[k+1]}
05      for (/*S4:*/j = i-1; 0 <= j && v <= a[j]; /*S5:*/j--)
06        /*S6:*/a[j+1] = a[j];
06a     //A3: {j == -1 ==> v < a[j+1]}
06b     //A4: {0 <= j && j < i ==> a[j] <= v && v < a[j+1]}
07      /*S7:*/a[j+1] = v;
07a     //A5: {A int k; 0 <= k && k < i+1 ==> a[k] <= a[k+1]}
08    }
08a   //A6: {A int k; 0 <= k && k < n-1 ==> a[k] <= a[k+1]}
09  }
```

Fig. 2. Method insertionSort() with embedded assertions

The sorted part satisfies the ascending order condition given by `{A int k; 0 <= k && k < i ==> a[k] <= a[k+1]}` (line 4a). This is actually a loop invariant that holds before (line 4a) and after (line 8a) of the outer loop body (from lines 4 to 8). The method reduces the unsorted part by removing its first element, and increases the sorted part by inserting the element at its correct position `j`. The position is scanned through the sorted part by the inner loop (lines 5 and 6). The insertion position occurs either at the beginning with `v < a[j+1]` if `j == -1` (assertion at line 6a), or at any other position whenever `a[j] <= v < a[j+1]` if `0 <= j < i` (assertion at line 6b), and in any case `v == a[j]` (after executing line 4). The invariant condition of insertion sort states that the sorted part remains ordered after inserting the element. The invariant condition holds because the greater elements are moved upwards, keeping their relative positions with each other, and then placing the element in the correct position with respect to its neighbors, either at the beginning `{j == -1 ==> v < a[j+1]}` (line 6a) or somewhere else `{0 <= j && j < i ==> a[j] <= v && v < a[j+1]}` (line 6b).

Assertions as annotated comments are intended to better the understanding about the program behavior by the programmer. However,

- Introducing comments is still a source code invasiveness practice in the sense that new assertions may affect program reuse and evolution.
- Consistency between the source base program and the assertion specifications is difficult to maintain.
- Numerous, diverse, long and complex assertions (not always needed) tends to obscure program readability.
- Annotated comments are not code.

Assertions should be implemented in the base programming language to produce a version that checks in run-time the assertion specifications.

2.2. Assertion checking

The most common strategy for implementing assertions follows the property of producing no effect if the assertion is valid and of producing an abnormal program termination if the assertion is invalid.

{*P*} `-> skip` if *P* holds
{*P*} `-> throw new Error("...");` if *P* fails

The first property uses pseudo-statement **`skip`** to denote the statement that does nothing (in Java for most contexts it corresponds to the semicolon). Both properties can be implemented using **`if`** statements whose condition is the negation of the assertion and whose protected statement is, among possibly others, the throwing of an error. Exceptions to this rule are quantifications that are examined later on.

Implications are not part of the Java logical connectives but it can be implemented in terms of negation and disjunction, as shown in Table 1. Recall that Java connectives evaluate subexpressions from left to right by shortcutting, stopping the evaluation as soon as the result is determined. Therefore in order to produce consistent results, side-effects in assertion evaluation must be avoided.

The simplest form of universal and existential quantification is also shown in Table 1. In this work, the quantified variables must be declared in a finite subset of an ordered domain, upon which are defined successor (and predecessor) functions that obtain the next element according to the order relation defined in the domain. Thus the schema provided in Table 1 can be easily extended to include variables ranging in enumerations and in any type of the Java collections (among lists, dictionaries, and maps). Java generic type system allows considering elements not only of predefined types but also of user-defined types, requiring only be finite in number. Quantification binds the quantified variable to all values ranging in the finite subset of the domain using the successor function, from the first to the last element, checking the validity of the assertion for each element. Universal quantification **`A`** fails if there is at least one element for which the assertion failed and succeeds otherwise. Existential quantification **`E`** succeeds if there is at least one element for which the assertion succeeds and fails otherwise. Other quantifiers like **`min`**, **`max`**, and **`sum`**, may be defined to obtain, respectively, the minimum, maximum and sum of all elements of an array, enumeration or collection. However, in our tailored assertion language we restrict to universal and existential quantification only.

Quantification should be wisely used because it takes a linear time on the number of elements of the collection to evaluate it. In the `insertionSort()` example, testing the condition that a subarray is in ascending order may be prohibitive to evaluate if the assertion is placed inside the outer loop due to the quadratic time cost of evaluating all the assertions even if the input array is already ordered.

Table 1. A simple implementation scheme for some basic assertions

Assertion	Implementation schema	Description
`//{P}`	`{ if (!P) throw new Error("...");` `}`	**If *P* holds, program continues after this point; otherwise, program aborts**
`//{P ==> Q}`	`{ if (P && !Q)` `throw new Error("...");` `}`	**If *P* holds, then *Q* holds**
`//{A int x;` **`a <= x && x < b`** **`==> C(x)}`**	`{ int x;` `for (x = a; x < b && C(x); x++)` `;` `if (a < b && x != b)` `throw new Error("...");` `}`	**_C_(_x_) holds for all integer _x_ in non-empty interval [_a_,_b_) ($a < b$) such that $a \le b \wedge x < b$ and also holds in any empty interval ($a \ge b$)**
`//{E int x;` **`a <= x && x < b`** **`&& C(x)}`**	**`{ int x;`** **`for (x = a; x < b && C(x); x++)`** **`;`** **`if (a >= b || x == b)`** **`throw new Error("...");`** **`}`**	**_C_(_x_) holds for some integer _x_ in non-empty interval [_a_,_b_) ($a < b$) such that $a \le b \wedge x < b$**

The implementation schema for the assertion language is summarized in Table 1. All assertion implementation schemas are translated into one or more Java statements, being the last one an **`if`** statement that throws an error when the assertion condition fails. For universally quantified assertions, a **`for`** statement makes quantified variable *x* to take all its values in interval $a \le x \wedge x < b$, checking the condition for each value of *x*. If the quantified variable reaches the greatest value $b-1$ in the interval, then the assertion is true. A similar schema is used in existentially quantified assertions, though it differs in that the quantified variable must not exceed the greatest value in the interval. Note that for empty intervals, universally quantified assertions are true, whereas existentially quantified assertions are false.

Following the implementation schema presented in Table 1, the new version of the `insertionSort()` method shown in Figure 3 replaces the assertion specification by the implementation code exactly at the same position of the program text.

In this example, the class has been extended with a number of public boolean methods that encapsulates the implementation of assertions. Otherwise, the assertion code would be inserted line by line into the `insertionSort()` code, turning difficult to discern between the method code and the assertion code. Inserted methods `pre_insertionSort()`, `post_insertionSort()`, `ascendingOrderCheck()` and `validPositionCheck()` are implementations of the assertion specifications given in Figure 2. The relatively complex condition of the **`if`** statement in method `validPositionCheck()` could be simplified applying the equivalences of logical connectives, but it has been written just following the translation schema of Table 1.

```
02  public static void insertionSort(int[] a) {
02a   pre_insertionSort(a);
03    for (int i = 1; i < a.length; i++) {
04      int j, v = a[i];
05      for (j = i-1; 0 <= j && v <= a[j]; j--)
06        a[j+1] = a[j];
06a     validPositionCheck(a,i,j,v);
07      a[j+1] = v;
07a     ascendingOrderCheck(a,i+1);
08    }
08a   post_insertionSort(a);
09  }
10  public static boolean pre_insertionSort(int[] a) {
11    if (a == null && a.length > 1)
12      throw new Error(“Null array or array with less than two elements”);
13    return true;
14  }
15  public static boolean post_insertionSort(int[] a) {
16    return ascendingOrderCheck(a,a.length);
17  }
18  public static boolean ascendingOrderCheck(int[] a, int n) {
19    //{A int k; 0 <= k && k < n-1 ==> a[k] <= a[k+1]}
20    int i;
21    for (i = 0; i < n-1 && a[i] <= a[i+1]; i++)
22      ;
23    if (0 <= n-1 && i != n-1)
24      throw new Error("Not in ascending order");
25    return true;
26  }
27  public static boolean validPositionCheck(int[] a, int i, int j, int v) {
28    //{j == -1 ==> v < a[j+1]}
29    //{0 <= j && j < i ==> a[j] <= v && v < a[j+1]}
30    if (!((!(j+1 == 0) || v < a[j+1]) || !(0 <= j && j < i) || v < a[j+1]))
31      throw new Error("Not a valid position "+j+" to insert "+v);
32    return true;
33  }
```

Fig. 3. Implementation of assertions by Java boolean methods and its use

These methods definitions and invocations show a straightforward manner of implementing the assertion checking code. The implementation illustrates the approach adopted in this work of inserting the externally defined assertion code into specific points within the base method code. These specific points of interest, marked by the labels shown in Figure 2, still need to be externally specified without modifying the base code if non-intrusiveness is expected. Both specifications, the assertion code and its insertion position, are given in a separated descriptor program that transforms the original base program into a run-time assertion checked program. Figure 3 shows the resulting method code after applying our program transformation technique.

Among the advantages of this approach:

- Modularity, uniformity and generality are improved in handling orthogonal crosscutting concerns.
- For many simple assertions, the corresponding implementations are relatively easy to build.
- Programming tools can be used to generate the assertion checking code from the assertion specification.

However, there are a number of disadvantages that remains in this approach too:

- Correct implementations are generally difficult to meet and the implementation may still need to be verified.
- Value-passing parameter mechanism of Java is in general not adequate for implementing assertions as boolean methods; instead, macro expansion-passing mechanism is more appropriate (including hygienic macro handling)
- Single returned value in methods is insufficient to handle the multiple assignments of values to variables.

Though these problems are not approached in this work, our proposal still offers the advantages discussed before over previous works.

3 Assertions as Aspects

Though AspectJ [Kickzales, et. Al., 2001; Hilsdale and Hugunin, 2004] is widely agreed as the most representative and influential AOP language developed so far, we will no discuss it here. A brief comparison with Énfasis is discussed in section 7 on the related work.

3.1. Aspects as Assertions in Énfasis

Énfasis is a novel AOP language that contributes in several respects to approach the run-time assertion checking problem in: (1) avoiding invasiveness at source code level, (2) defining a join point model for fine crosscutting granularity, (3) providing a path expression language to describe the pointcuts of interest, and (4) introducing a join point model for local variables.

Fine crosscutting granularity is defined not only for method invocations but also for statements and expressions [Rho, et. al., 2006; Eaddy and Aho, 2006]. Run-time assertion checking requires getting access to all the local variables that appear in the assertion. However, as assertions appear scattered through the method body, in order to deal with this crosscutting concern it is necessary to identify the join points of interest by providing a pointcut description of them. Énfasis determines a basic set of join points for a local variable by specifying a path in the abstract syntax tree of the method body to the method fragment in which the variable exists along with some of the assertions that use it. A path expression describes the traversal in the abstract syntax tree required to reach a set of join points of interest for a local variable within the method code. A path expression consists of a pattern and a path. A pattern selects the package, the class, and the method by means of simple regular expressions. A path selects the method fragment where the local variable occurs. A pointcut descriptor is a complete path expression.

For example in Figure 3, denoting each join point by the labels of Figure 2, as shown by the excerpt given next:

```
02  public static void insertionSort(int[] a) {
02a   //A1: {a != null && a.length > 1}
03    for (/*S1:*/int i = 1; /*G1:*/i < /*G2:*/a.length; /*S2:*/i++) {
04      int j, /*S3:*/v = /*G3:*/a[/*G4:*/i];
04a     //A2: {A int k; 0 <= k && k < i ==> a[k] <= a[k+1]}
05      for (/*S4:*/j = i-1; 0 <= j && v <= a[j]; /*S5:*/j--)
06        /*S6:*/a[j+1] = a[j];
06a     //A3: {j == -1 ==> v < a[j+1]}
06b     //A4: {0 <= j && j < i ==> a[j] <= v && v < a[j+1]}
07      /*S7:*/a[j+1] = v;
```

the path expression that contains only a pattern:

```
public static int SearchingAndSorting.insertionSort(int[] a)/*::setLocal(int *);
```

denotes the set {`S1,S2,...,S7`} of all join points (for any variable of type **`int`**) that appear anywhere in the method body of `insertionSort()` method of class `SearchingAndSorting`, whereas the path expression that contains both a pattern and a path:

```
public static int SearchingAndSorting.insertionSort(int[] a)
/for/do/for/do/assignment::setLocal(int i, a[]);
```

describes the set {`S6,S7`} of all join points where local integer variables `i` and `a[]`, appear in the left-hand side of the first assignment occurring in the body of the first inner **`for`** loop which in turn occurs in the body of the first outer **`for`** loop. Finally, the path expression:

```
public static int SearchingAndSorting.insertionSort(int[] a)
/for/condition::getLocal(int i, j, v, a[]);
```

describes the set {G1,G2} of join points appearing in the condition of the first outer **`for`** loop.

The matching join points are combined with a minimal set of composition operations defined on sets of join points like complement, union, and intersection. This model provides a rich pointcut description language to query for join points with fine granularity upon which can be defined advising code. In this respect, neither AspectJ-like languages nor Prolog-like query based languages join point query languages support this kind of crosscutting.

Assertions, as advice code, are inserted in the set of join points determined by the pointcut descriptor. As mentioned before, they can be inserted instead, before or after the instruction where the join point is located. Thus, for example, given the pointcut insert:

```
public static int SearchingAndSorting.insertionSort(int[] a)
/for/do/assignment::setLocal(int i, j, v, a[]);
```

that describes the set {S7} of join points that assign a value to element `a[j+1]`. By applying advice before and after:

```
before({S7}, validPositionCheck(a,i,j,v););
after({S7}, ascendingOrderCheck(a,i+1););
```

respectively, the following program fragment is obtained:

```
06a           validPositionCheck(a,i,j,v);
07               /*S7:*/a[j+1] = v;
07a           ascendingOrderCheck(a,i+1);
```

as shown in Figure 2.

The `Correctness` class that implements an aspect by extending the Aspect class is shown in Figure 3. This program is compiled separately from the `SearchingAndSorting` class and executed only after this class has been compiled into bytecodes. Énfasis weaves the assertions as advice code in to the base code of `insertionSort()` method. The result is the asserted version of the class shown in Figure 2.

Crosscutting on local variables in Énfasis introduces a notation to describe local variable pointcuts with name, type, occurrence number within a program fragment, and accessing instruction for reading or writing. Though refactoring on fields has been considered as an implementation option for local variable crosscutting, this option is unacceptable in recursive definitions because memory allocation code for fields is not reentrant. Besides, it leads to inefficient implementations both in terms of time and memory space. Local variables have a number of advantages over fields: 1) well-defined lexicographic scoping rules, 2) efficient memory handling through stack allocation, 3) optimized memory instructions, 4) well-suited for recursive definitions, and 5) improved expressiveness for assertion descriptions. This advantages leads to a considerable increase in both descriptive power and in efficiency of the generated code by avoiding refactoring.

```
public class Correctness extends Aspect {
public static void main(){
  String method = "public static int SearchingAndSorting.insertionSort(int[]a)";
  Pointcut body = new Pointcut(method+"/*;");
  Advice precondition = new Advice("pre_insertionSort(a);");
  Advice postcondition = new Advice("post_insertionSort(a);");
  Pointcut insert = new Pointcut(method+"/for/do/assignment::getLocal(int i, j, v,
  a[]);");
  Advice position_check = new Advice("validPositionCheck(a,i,j,v);");
  Advice ordering_check = new Advice("ascendingOrderCheck(a,i+1);");
  Weaver w = new Weaver();
  w.add(new Before(body, precondition));
  w.add(new After(body, postcondition));
  w.add(new Before(insert, position_check));
  w.add(new After(insert, ordering_check));
  w.weave();
}...
```

Fig. 4. Assertions in Énfasis. The Correctness aspect written in Énfasis for the `SearchingAndSorting` class

Nonetheless, the asserted version of the program still needs to be checked at run-time. Assuming that the programmer guarantees the correctness of the assertion code, it still remains to show that the inserted code correctly checks the original program. In order to do so, in the next section we introduce the structured operational semantics upon a Java subset that is interesting enough to test our approach. We will prove that a partially correct method if terminates, it is correct with respect to its assertions; otherwise, it will terminate by throwing an error.

4 Base Programming Model

In the remaining part of this section, the notion of assertion evaluation is formalized in order to decide on the validity of assertions. The abstract syntax of deterministic programs is generated from the following grammar rules of the Java subset considered here:

$$J ::= \texttt{error} \mid \texttt{skip} \mid x = E\texttt{;} \mid a[E_1] = E_2\texttt{;} \mid \texttt{if}(B)\, J_1\, \texttt{else}\, J_2 \mid \texttt{while}(B)\, J \mid J_1\, J_2 \mid \{J\} \mid m(E_1,...,E_2)$$

A *program fragment* or simply a *program J* consists of statements among assignments on simple variables *x* and array variables *a*, selective instructions, iterative instructions, sequential composition, grouping of instructions enclosed by brackets called *block*, and method calls. Method declaration is not considered because this work focuses in method definition. Class declaration is not included because class invariants are not considered here.

Arithmetic expressions *E* consist of integer constants, integer variables, and compound expressions formed by composition of arithmetic expressions with usual arithmetic operators, according to the following abstract syntax:

$$E ::= \perp \mid c \mid x \mid a[E] \mid E_1 + E_2 \mid E_1 \times E_2 \mid \cdots$$

where *E* (along with its decorations) denotes the usual arithmetic expressions including constants, variables, elements of array variables, and compound arithmetic expressions. Boolean expressions *B* consist of the boolean constants, simple boolean expressions formed by the comparison of arithmetic expressions with the usual relational operators of equality and inequality and by compound boolean expressions formed by the composition of boolean expressions with the logical connectives of negation, conjunction, and disjunction, according to the following abstract syntax:

$$B ::= \perp \mid \texttt{true} \mid \texttt{false} \mid E_1 = E_2 \mid E_1 < E_2 \mid \cdots \mid \texttt{!}B \mid B_1\, \texttt{\&\&}\, B_2 \mid B_1\, \texttt{||}\, B_2$$

The *assertion language* is defined according to the following abstract syntax:

$$A ::= B \mid A ==> B \mid \mathtt{\textbf{A int}}\ x.A \mid \mathtt{\textbf{E int}}\ x.A$$

Assertions use either program variables or logical variables and include all boolean expressions written using the common relational operators. Logical variables, known as *model variables*, are not declared in the program but are required in the assertion language to improve its expressiveness. All quantified expressions are *closed*, i.e. there are no unbound free logical variables.
A *value of type T* is an element in carrier set D_T of *T*. An *array of type T* is a finite partial function $\mathbf{N} \to D_T$ from natural numbers to values ranking on carrier set D_T of *T*. For example, integer type **`int`** denotes the set $D_{\mathtt{int}}$ of integers $\{0, \pm 1, \pm 2, ...\}$ that can be represented by the JVM, whereas Boolean type **`boolean`** denotes the set $D_{\mathtt{boolean}}$ of logical constants $\{\mathtt{\textbf{true}}, \mathtt{\textbf{false}}\}$.

A *program state* is a partial function $V_T \to D_T + A_T \to \mathbf{N} \to D_T$ defined on variable names V_T to values on D_T and array variable names A_T to arrays on $\mathbf{N} \to D_T$. The $dom(f) = \{x \mid x \mapsto e \in f\}$ of partial function *f* is the set of values where the function is defined. The values of the array are indexed and the index ranks in $dom(a)$, i.e. from 0 to $a.length - 1$ where $a.length$ denotes the number of elements in array *a*. As an example, program state $\sigma = \{x \mapsto 0, y \mapsto 1, a \mapsto \{0 \mapsto 2, 1 \mapsto 4, 2 \mapsto 6\}\}$ comprises variable *x* bound to value 8, variable *y* bound to value 1, and array variable *a* bound to array $\{0 \mapsto 2, 1 \mapsto 4, 2 \mapsto 6\}$ of length 3. The *value* $\sigma(x)$ of variable *x* in state σ is either defined or undefined. The value $\sigma(x) = e$ of variable *x* is defined if exists some value *e* of the same type bound to *x* in σ (i.e. $x \mapsto e \in \sigma$), and undefined, written $\sigma(x) = \perp$, otherwise. The value of element *i* of array *a*, written $a[i]$, is defined $\sigma(a[i]) = \sigma(a)(\sigma(i))$ if *i* is in rank (i.e. in $dom(a)$). The values of *x* and *a*, are respectively $\sigma(x) = 0$, $\sigma(a) = \{0 \mapsto 1, 1 \mapsto 4, 2 \mapsto 6\}$ and $\sigma(a[x]) = \sigma(a)(\sigma(x)) = \{0 \mapsto 2, 1 \mapsto 4, 2 \mapsto 6\}(0) = 2$. A *change in program state* $\sigma\{x \mapsto e\}$ results in a new state after updating the previous one by assigning new value *e* to variable *x* of the same type such that $(\sigma\{x \mapsto e\})(x) = e$. Note that a valuation of a variable always takes its last assigned value (i.e. $(\sigma\{x \mapsto e\}\{x \mapsto e'\})(x) = e'$) and also that a valuation looks for the variable definition in the program state (i.e. $(\sigma\{x \mapsto e\}\{y \mapsto e'\})(x) = e$).

The *value* (*evaluation*) of arithmetic and Boolean expressions is defined by induction on the structure of the abstract syntax of expressions:

$$\begin{array}{rcl}
\perp \sigma & = & \perp \\
c\ \sigma & = & c \\
x\ \sigma & = & e \ \textbf{if}\ \sigma(x) = e \\
x\ \sigma & = & \perp \ \textbf{if}\ x \notin dom(\sigma) \\
a[E]\ \sigma & = & \sigma(a)(E\ a) \\
a[E]\ \sigma & = & \perp \ \textbf{if}\ a \notin dom(\sigma) \\
 & & \quad \textbf{or if}\ a \in dom(\sigma)\ \textbf{and}\ E\ \sigma \notin dom(a) \\
(E_1 + E_2)\ \sigma & = & E_1\ \sigma + E_2\ \sigma \\
(E_1 \times E_2)\ \sigma & = & E_1\ \sigma \times E_2\ \sigma \\
(E_1 = E_2)\ \sigma & = & E_1\ \sigma = E_2\ \sigma \\
(E_1 < E_2)\ \sigma & = & E_1\ \sigma < E_2\ \sigma \\
(\mathbf{!}B)\ \sigma & = & (\neg B)\ \sigma \\
(B_1 \mathbf{\&\&} B_2)\ \sigma & = & B_1\ \sigma < B_2\ \sigma \\
(B_1 \mathbf{||} B_2)\ \sigma & = & B_1\ \sigma \vee B_2\ \sigma
\end{array}$$

The valuation of an expression is a compositional partial function whose value is obtained from the application of the denotation of the operator (relational or connective) to the values of the corresponding operands (subexpressions). The value of an expression may be undefined if there is an undefined variable occurring in the expression. In case of an element of an array, the value is undefined if the index is out of rank. Other operators may lead to undefined valuations as in the case of division by zero. The valuation of any unary or binary operation is undefined if one of its operands is undefined.

The *validity* relation $\sigma \models A$ of assertion A in state σ means that A holds in σ and is defined inductively on the structure of the abstract syntax of assertions.

$$\begin{array}{rl}
\bot \models \texttt{false} & \\
\sigma \models A & A\sigma = \texttt{true} \\
\sigma \models A_1 \texttt{==>} A_2 & A_1\sigma \Rightarrow A_2\sigma \\
\sigma \models \texttt{Aint}\, x.A & \forall n \in \mathbf{N}.\ \sigma\{x \mapsto n\} \models A \\
\sigma \models \texttt{Eint}\, x.A & \exists n \in \mathbf{N}.\ \sigma\{x \mapsto n\} \models A
\end{array}$$

Inconsistent state $\bot$ is an abnormal state that is reached by a program that does not terminate, making invalid any assertion if the state could be reached. Evaluation of A is well-defined when all variables in σ are defined. Assertion A is valid for state σ if value of A is true. Implication of assertions holds if the implication of the corresponding assertions holds. Quantification over integer variables has the usual interpretation. Universal quantification of an assertion is valid if the assertion is valid for all integer values that the quantified variable can take, whereas existential quantification is valid if there is some integer value that the quantified variable can take for which the assertion is valid.

Assertion evaluation can be defined using the *structural operational semantics* [Reynolds, 1998] by defining a transition relation over pairs of program constructs and program states, as follows:

$$\langle J, \sigma \rangle \rhd \langle J', \sigma' \rangle$$
$$\langle J, \sigma \rangle \rhd \sigma'$$

meaning that whenever program J starts executing at state σ, the program execution leads to either the program fragment $J¢$ that continues executing afterwards at state σ' or the program termination at state σ'.

$$\langle \texttt{skip}, \sigma \rangle \rhd \sigma$$
$$\langle \texttt{error}, \sigma \rangle \rhd \langle \texttt{error}, \sigma \rangle$$
$$\langle \texttt{error}, \sigma \rangle \rhd \bot$$

$$\frac{E\sigma \neq \bot}{\langle x = E\texttt{;}, \sigma \rangle \rhd \sigma\{x \mapsto E\sigma\}} \qquad \frac{E\sigma = \bot}{\langle x = E\texttt{;}, \sigma \rangle \rhd \langle \texttt{error}, \sigma \rangle}$$

$$\frac{E_1\sigma, E\sigma \neq \bot}{\langle a[E_1] = E_2\texttt{;}, \sigma \rangle \rhd \sigma(a)\{E_1\sigma \mapsto E_2\sigma\}} \qquad \frac{E_1\sigma = \bot \vee E\sigma = \bot}{\langle a[E_1] = E_2\texttt{;}, \sigma \rangle \rhd \langle \texttt{error}, \sigma \rangle}$$

$$\frac{B\sigma = \texttt{true}}{\langle \texttt{if}(B)\, J_1\ \texttt{else}\ J_2, \sigma \rangle \rhd \langle J_1, \sigma \rangle}$$

$$\frac{B\sigma = \texttt{false}}{\langle \texttt{if}(B)\, J_1\ \texttt{else}\ J_2, \sigma \rangle \rhd \langle J_2, \sigma \rangle}$$

$$\frac{B\sigma = \bot}{\langle \texttt{if}(B)\, J_1\ \texttt{else}\ J_2, \sigma \rangle \rhd \langle \texttt{error}, \sigma \rangle}$$

$$\frac{B\sigma = \mathtt{true}}{\langle \mathtt{while}(B)\,J,\,\sigma\rangle \rhd \langle J\ \mathtt{while}(B)\,J,\,\sigma\rangle}$$

$$\frac{B\sigma = \mathtt{false}}{\langle \mathtt{while}(B)\,J,\,\sigma\rangle \rhd \langle \mathtt{skip},\,\sigma\rangle}$$

$$\frac{B\sigma = \bot}{\langle \mathtt{while}(B)\,J,\,\sigma\rangle \rhd \langle \mathtt{error},\,\sigma\rangle}$$

$$\frac{\langle J_1,\sigma\rangle \rhd \langle \mathtt{error},\,\sigma\rangle}{\langle J_1\ J_2,\sigma\rangle \rhd \langle \mathtt{error},\,\sigma\rangle}$$

$$\frac{\langle J_1,\sigma\rangle \rhd \langle J_1',\,\sigma'\rangle}{\langle J_1\ J_2,\sigma\rangle \rhd \langle J_1'\ J_2,\,\sigma'\rangle} \qquad \frac{\langle J_1,\sigma\rangle \rhd \sigma'}{\langle J_1\ J_2,\sigma\rangle \rhd \langle J_2,\,\sigma'\rangle}$$

$$\langle \{J\},\sigma\rangle \rhd \langle J,\sigma\rangle$$

$$\frac{x \notin dom(\sigma)}{\langle \mathtt{int}\ x = E\mathtt{;}\,J,\,\sigma\rangle \rhd \langle J\ \mathtt{unbind}\ x\mathtt{;},\,\sigma\{x \mapsto E\sigma\}\rangle}$$

$$\frac{x \in dom(\sigma)}{\langle \mathtt{int}\ x = E\mathtt{;}\,J,\,\sigma\rangle \rhd \langle \mathtt{error},\,\sigma\rangle}$$

$$\frac{a \notin dom(\sigma)}{\langle \mathtt{int}\ a[\,] = \{E_0,\ldots,E_{n-1}\}\mathtt{;}\,J,\,\sigma\rangle \rhd \langle J\ \mathtt{unbind}\ a\mathtt{;},\,\sigma\{a \mapsto \{0 \mapsto E_0\sigma, 1 \mapsto E_1\sigma,\ldots,n-1 \mapsto E_{n-1}\sigma\}\}\rangle}$$

$$\frac{a \in dom(\sigma)}{\langle \mathtt{int}\ a[\,] = \{E_0,\ldots,E_{n-1}\}\mathtt{;}\,J,\,\sigma\rangle \rhd \langle \mathtt{error},\,\sigma\rangle}$$

$$\langle \mathtt{unbind}\ x\mathtt{;},\sigma\{x \mapsto E\}\rangle \rhd \sigma$$

$$\frac{\mathtt{void}\ m(x_1,\ldots,x_n)\{J\}\ \textit{declared in class}}{\langle m(E_1,\ldots,E_n),\,\sigma\rangle \rhd \langle J\ \mathtt{unbind}\ x_1\mathtt{;}\cdots\mathtt{;}\ \mathtt{unbind}\ x_n\mathtt{;},\,\sigma\{x_1 \mapsto E_1\sigma,\ldots,x_n \mapsto E_n\sigma\}\rangle}$$

Null program **`skip`** does nothing, maintaining program state unchanged. Error program **`error`** leads to a non terminating configuration, and if it could terminate, it would be in an inconsistent state (i.e. satisfying **`false`**). Assignment to simple variable $x = E\mathtt{;}$ evaluates expression E at the right-hand side and binds this value to variable x at the left-hand side, updating program state to $\sigma\{x \mapsto E\sigma\}$. Similarly assignment to an element of an array variable $a[E_1] = E_2\mathtt{;}$ evaluates expression E_2 at the right-hand side and binds this value to element of array a given by evaluating expression E_1. In this case, updating program state consists on updating array $\sigma(a)$ at element given by $\sigma(E_1)$. Selective program $\mathtt{if}(B)\,J_1\ \mathtt{else}\ J_2$ does not change program state, assuming that no side-effects arise when evaluating condition *B*. Program J_1 is selected if condition *B* is valid at s, otherwise J_2 is selected.

Iterative program $\mathtt{while}(B)\,J$ can be rewritten by unwinding to $\mathtt{if}(B)\{J\ \mathtt{while}(B)\ J\}\mathtt{else}\,\mathtt{skip;}$. It does not change program state, assuming that no side-effects are present when evaluating condition *B*. Iterative program unwinds program *J* while condition *B* is valid at current state; otherwise it behaves like **`skip`**. Sequence of programs $J_1\ J_2$ starts executing J_1 and upon its completion continues executing J_2. The execution of a block program behaves like the execution of its enclosed statement. Variable declaration, including array variable, introduces a new binding into the program state if the variable has not been declared before; otherwise it causes an error. Variable scope is provided by unbinding the variables declared at the end of its block by means of special statement **`unbind`**. Finally, method invocation behaves like its defining block at the same state but extended with the parameters passed.

Null program **`skip`** has not a unique notation in Java and depends on the context in which it occurs: it may be represented by a single semicolon (;) but it may also appear as an empty instruction, as for example, in the initialization and re-initialization parts of the **`for`** statement. Furthermore, there is a number of programming features not included in the Java subset that can still be captured by this notation and semantics. The family of increment and decrement operators can be translated to simple assignment statements with additions and substractions, respectively. Simple if statement $\mathtt{if}(B)\,J$ can be represented as $\mathtt{if}(B)\,J\;\mathtt{else}\,\mathtt{skip}$, representing **`skip`** as a semicolon. The iterative **`for`** statement $\mathtt{for}(E_1;\,E_2;\,E_3)\,J$ can be represented as $E_1\;\mathtt{while}(E_2)\left\{J\;E_3\right\}$. Throwing errors is considered as an abnormal program termination and behaves like **`error`** which leads to abnormal program termination. Method declaration and definition is not included in the transition relation because this work focuses on the body of the method.

The meaning of partial correctness of a Hoare triple can now be formally defined as follows:

$$\sigma \models \{P\}J\{Q\}$$
$$\text{if and only if}$$
$$\forall\sigma.\forall\sigma'.\left(\langle J,\sigma\rangle \rhd \sigma'\right) \wedge \sigma \models P \Rightarrow \sigma' \models Q$$

Hoare triple $\{P\}J\{Q\}$ is valid at state σ if and only if program J starts executing at σ where assertion P is valid, and then the program terminates at state σ' where assertion Q is valid.

Hoare triples can be deduced from the inference rules of the Floyd-Hoare logic. However, because this work is focused on assertion runtime checking, this logic does not need to be elaborated here. Automated proving of program correctness in the Floyd-Hoare requires the use of sophisticated automated reasoning tools assisted with effective theorem proving strategies. Nonetheless, some proof assistants can be adapted to read both the program and the assertions specified in Énfasis. Besides, model checking methods can also be adapted, but it requires an entire research by itself.

5 Crosscutting Programming Model

The programming model defines a join point for a local variable as the address of the instruction that get access to the variable. From this definition, a mechanism can be devised to retrieve run-time information of the local variable such as its name, type, location (slot) and accessing instruction.

This notion of join point also provides an operational interpretation based on the mapping between a pointcut descriptor and the join points (instruction addresses) that it specifies.

In this model, high level description of local variables pointcuts is achieved by introducing a source program notation for join points and by binding each join point to its corresponding accessing instruction address. The binding can be produced by visiting in post-order the abstract syntax tree (AST) of the source program and by visiting the bytecode program in a synchronized manner. The synchronization consists in pairing a program fragment (as a subtree in the AST) with the corresponding segment of the byte code generated by the compiler.

Binding join point labels to instruction addresses helps to maintain a high level perspective for the pointcut descriptor. Énfasis introduces a language for pointcut descriptors to recognize patterns of join points detected at run-time. The language incorporates some novel features like path and filter expressions to better the resolution in the join point selection mechanisms.

5.1. Pointcut Descriptor Syntax

The abstract syntax of the Java subset includes only a minimal extension that can be hidden during lexicographic analysis. The extension consists on decorating each local variable with a unique label that fix the position of a join point for the variable.

$$\begin{array}{lcl}
J & ::= & [N:]\,J \mid \cdots \\
E & ::= & c \mid [N:]\,x \mid [N:]\,a[E] \mid E_1 + E_2 \mid \cdots \\
B & ::= & \texttt{\textbf{true}} \mid \texttt{\textbf{false}} \mid E_1 < E_2 \mid B_1 \,\texttt{\textbf{\&\&}}\, B_2 \mid \cdots \\
V & ::= & H\,[\texttt{\textbf{/}}P][\texttt{\textbf{::}}F(L)][\texttt{\textbf{@}}N][\texttt{\textbf{\#}}\,N] \\
H & ::= & T\;Q([L]) \\
M & ::= & Q([A]) \\
T & ::= & T[\,] \mid \texttt{\textbf{boolean}} \mid \texttt{\textbf{byte}} \mid \texttt{\textbf{int}} \mid \texttt{\textbf{float}} \mid \cdots \\
L & ::= & T \mid T\;x \mid L_1\,,\,L_2 \\
A & ::= & E \mid A_1\,,\,A_2 \\
P & ::= & \texttt{\textbf{*}} \mid K \mid P\texttt{\textbf{/}}K \mid \cdots \\
K & ::= & C[n] \mid C \mid S \\
C & ::= & \texttt{\textbf{assign}} \mid \texttt{\textbf{if}} \mid \texttt{\textbf{while}} \mid \texttt{\textbf{call}} \mid \cdots \\
S & ::= & \texttt{\textbf{cond}} \mid \texttt{\textbf{then}} \mid \texttt{\textbf{else}} \mid \texttt{\textbf{do}} \mid \cdots \\
F & ::= & \texttt{\textbf{hasL}} \mid \texttt{\textbf{getL}} \mid \texttt{\textbf{setL}} \\
D & ::= & V \mid \texttt{\textbf{!}}D \mid D_1 \,\texttt{\textbf{\&\&}}\, D_2 \mid D_1 \,\texttt{\textbf{!!}}\, D_2
\end{array}$$

in the above grammar rules, x ranks over simple variable names, a ranks over array variable names, c and n ranks over the integer and natural numbers respectively, and N ranks over sets of ordered natural numbers. Syntactic category M describes a restricted form of method declaration comprising only the method name, type and parameter list. By simplicity types are restricted to either **`int`** or **`int`**`[]`. Program statements are as usual except that variables are possibly labeled. Thus, labels appear in the left-hand side of assignments an in any part of a program in which (arithmetic or **`boolean`**) expressions may occur. However, expressions that are not allowed in this version of Énfasis are those obtained from the increment and decrement operators due to their difficult side-effect semantics. In that follows, syntactical parenthesis $[\![P]\!]$ denote the ordered set of joint points in program fragment P (represented by its abstract syntax tree).

5.2. Pointcut Descriptor Semantics

Path expression is a novel feature introduced in Énfasis to address a finer granularity in pointcut descriptors. A path expression describes the sequence of subtree selections on an AST. Syntactic category P above describes the structure of a path as a sequence of constructors and selectors. A constructor in C corresponds to a symbolic name for each Java statement. A constructor can be followed by a skip index enclosed in brackets $C[i]$ intended to resume the traversing of the AST at the i-th sibling constructor C if any; otherwise it has no effect. A selector in S indicates the part of the statement to be chosen next. Thus, in any given level of the AST (i.e. a block), a constructor skips all subtrees whose root does not correspond to the same constructor. If a skip index is included in a constructor, then it also skips the number of subtrees rooted with the same constructor. The subtree selection process begins with the AST representing the entire code of a given method. For example, the path expression `SortAndSearch.insertionSort(`**`int`**`[] a, `**`int`**` n)`**`/while/do`** states that the first **`while`** statement should be selected and then its **`do`** part. By unfolding the method definition of `search()` for the method body, the subtree selection process can begin. The meaning of path expressions is given by the following equations.

$$
\begin{aligned}
[\![M]\!] &= [\![J]\!] \textbf{ if } M\{J\} \text{ is declared in class} \\
[\![J]\!]\texttt{/*/}P &= [\![J]\!] \\
[\![L:x=E\texttt{;}]\!]\texttt{/assign/}P &= \{L\} \\
[\![\texttt{if}\,(B)\,J_1\,\texttt{else}\,J_2]\!]\texttt{/if/cond/}P &= [\![B]\!] \\
[\![\texttt{if}\,(B)\,J_1\,\texttt{else}\,J_2]\!]\texttt{/if/then/}P &= [\![J_1]\!]\texttt{/}P \\
[\![\texttt{if}\,(B)\,J_1\,\texttt{else}\,J_2]\!]\texttt{/if/else/}P &= [\![J_2]\!]\texttt{/}P \\
[\![\texttt{while}\,(B)\,J]\!]\texttt{/while/cond/}P &= [\![B]\!] \\
[\![\texttt{while}\,(B)\,J]\!]\texttt{/while/do/}P &= [\![J]\!]\texttt{/}P \\
C[0] &= C \\
[\![J_1\;J_2]\!]\texttt{/}C[i]\texttt{/}P &= [\![J_2]\!]\texttt{/}C[i]\texttt{/}P \textbf{ if } cons(J_1) \neq C \\
[\![J_1\;J_2]\!]\texttt{/}C[i]\texttt{/}P &= [\![J_2]\!]\texttt{/}C[i-1]\texttt{/}P \textbf{ if } cons(J_1) = C \wedge i > 0 \\
[\![J_1\;J_2]\!]\texttt{/}C[0]\texttt{/}P &= [\![J_1]\!]\texttt{/}C\texttt{/}P \textbf{ if } cons(J_1) = C
\end{aligned}
$$

where intrinsic function *cons*() returns the name of the constructor placed at the root of the given abstract syntax subtree.

Filter expressions select only the join points that correspond to the local variable given and also for the kind of accessing instruction (for reading or writing). Syntactic structure of filters is given by category *F* and used in the context of category *V* of the abstract syntax. The meaning of a filter expression is given by the following equations.

$$
\begin{aligned}
[\![J]\!]\texttt{::hasL}(T\,x) &= [\![J]\!]\texttt{::setL}(T\,x) \cup \texttt{getL}(T\,x) \\
[\![L:x=E\texttt{;}]\!]\texttt{::setL}(T\,x) &= \texttt{if}\,v=x\,\texttt{then}\,\{L\}\,\texttt{else}\,\varnothing \\
[\![L:x=E\texttt{;}]\!]\texttt{::getL}(T\,x) &= [\![E]\!]\texttt{::getL}(T\,x) \\
[\![c]\!]\texttt{::getL}(T\,x) &= \varnothing \\
[\![L:x]\!]\texttt{::getL}(T\,x) &= \texttt{if}\,v=x\,\texttt{then}\,\{L\}\,\texttt{else}\,\varnothing \\
[\![E_1+E_2]\!]\texttt{::getL}(T\,x) &= [\![E_1]\!]\texttt{::getL}(T\,x) \cup [\![E_2]\!]\texttt{::getL}(T\,x) \\
[\![J_1\;J_2]\!]\texttt{::xL}(T\,x) &= [\![J_1]\!]\texttt{::xL}(T\,x) \cup [\![J_2]\!]\texttt{::xL}(T\,x) \\
[\![\texttt{if}\,(B)\,J_1\,\texttt{else}\,J_2]\!]\texttt{::xL}(T\,x) &= [\![B]\!]\texttt{::xL}(T\,x) \cup [\![J_1]\!]\texttt{::xL}(T\,x) \cup [\![J_2]\!]\texttt{::xL}(T\,x) \\
[\![\texttt{while}\,(B)\,J]\!]\texttt{::xL}(T\,x) &= [\![B]\!]\texttt{::xL}(T\,x) \cup [\![J]\!]\texttt{::xL}(T\,x)
\end{aligned}
$$

where **`xL`** stands for either **`setLocal`** or **`getLocal`**.

Selecting specific occurrences of a set of join points is a very useful operation that avoids having to decompose arithmetic and boolean expressions in all their fragments as it was done for statements. The programmer does not need to analyze the AST of an expression and only needs to count the number of occurrence of interest, following the order in that they appear in the source program. This approach considerably simplifies the notation of pointcut descriptors, avoiding verbose and unreadable path expressions.

The occurrence number selector `@` and the line number selector `#` are defined by the set comprehensions:

$$
[\![J]\!]\texttt{@}N = \left\{ j!!i \mid j \in J \wedge i \in N \right\}
$$

$$[\![J]\!]\,\#\,N \quad = \quad [\![J]\!]\,@\Big(\bigcup_{l\in N} OccsAtLine(l)\Big)$$

where notation $j!!i$ takes element i from ordered set j. The occurrence number selector `@` collects the set of all join points from the ordered set $[\![J]\!]$ occurring at the positions given by set N. The line number selector `#` firstly collects all the variables occurring in each line l of set N, by using the *OccsAtLine*() function, and then applies the occurrence number selector.

From the definition of join point also arise a natural definition of pointcut descriptor composition. Composition rules are given by category D in the grammar. Basic logic connectives like conjunction, disjunction and negation correspond to the basic operations of intersection, union and complement on ordered sets of join points. Complement of a set is obtained with respect to the complete set of join points in a method.

$$\begin{aligned} [\![D_1]\!]\,\mathbf{\&\&}\,[\![D_2]\!] &= [\![D_1]\!] \cap [\![D_2]\!] \\ [\![D_1]\!]\,\mathbf{||}\,[\![D_2]\!] &= [\![D_1]\!] \cup [\![D_2]\!] \\ \mathbf{!}[\![D]\!] &= [\![M]\!] \setminus [\![D]\!] \end{aligned}$$

These operations provide the means to compose complex pointcut descriptors from simpler ones.

5.3 Advice Semantics

The set of join points, once defined, identify the points in the source program where the aspectual code will be inserted. The only two basic advice mechanisms considered here are defined as:

$$\begin{aligned} \texttt{before}(L:J,A) &= \{A\ L:J\} \\ \texttt{after}(L:J,A) &= \{L:J\ A\} \end{aligned}$$

where notation L: J denotes source instruction J labeled with join point location L, and aspectual code A. Each advice structure introduces a block to group both the source and the aspectual code in order to extend the scope in which the former originally appeared. It is assumed that no free local variables are introduced by the advice code to avoid the name capturing problem. Besides, if any local variable is declared, its scope is lexically restricted to the scope of the advice as it often occurs in control variables of **`for`** loops. In case of assertions, the advice code must not cause any side-effect, ensuring that the program state is maintained with no change.

6 Proving Program Correctness in Énfasis

In this section, we show that weaving aspects in Énfasis for the assertion verification problem produces a semantically equivalent program with assertion handling.

ASSUMPTION. Assertion implementation does not modify program behavior if assertion holds; otherwise, it throws an error.

1. $\langle I[\![P]\!],\sigma\rangle \triangleright \sigma$ if $\sigma \models P$
2. $\langle I[\![P]\!],\sigma\rangle \triangleright \langle \texttt{error},\sigma\rangle$ if $\sigma \models \neg P$

Proposition 1 establishes that Énfasis does not change original program behavior if assertions are considered special instructions with no effect (like comments).

PROPOSITION 1. Given Hoare triple $\{P\}L:J\{Q\}$ and p is the path to join point label L of J, weaving advices $\texttt{before}(p/J,P)$ and $\texttt{after}(p/J,Q)$ into the source program correctly generates the Hoare triple.

PROOF. The proposition follows immediately from the definition of advices **`before`** and **`after`** when instruction J is at the top level in the block in which the instruction appears. However, in case of that J is not in a block but the sole instruction, then it must be changed by semantically equivalent block instruction $\{J\}$, and then proceed as before. QED

Proposition 2 is similar to the previous one except by the fact that assertions are correctly implemented by aspectual code.

PROPOSITION 2. Given $I[\![P]\!]$ and $I[\![Q]\!]$ correct implementations of precondition P and postcondition Q of $L : J$, and p the path to label L, weaving advices $\mathtt{before}(p/J, I[\![P]\!])$ and $\mathtt{after}(p/J, I[\![Q]\!])$ into the source program produces instruction $\{I[\![P]\!]\, L : J\ I[\![Q]\!]\}$ that correctly check the assertions for J.

PROOF. From the previous proof, advices **`before`** and **`after`** interlace a correctly generated Hoare triple. Then it remains to show that sequential instruction $\{I[\![P]\!]\, L : J\ I[\![Q]\!]\}$ behaves like $L : J$ only if precondition P holds; otherwise, the behavior departs from the original one by throwing an error. Therefore, we want to show that:

1. $\langle \{I[\![P]\!]\, L : J\ I[\![Q]\!]\}, \sigma \rangle \triangleright \sigma'$ if $\langle J, \sigma \rangle \triangleright \sigma'$, $\langle I[\![P]\!], \sigma \rangle \triangleright \sigma$ and $\langle I[\![Q]\!], \sigma' \rangle \triangleright \sigma'$, and
2. $\langle \{I[\![P]\!]\, L : J\ I[\![Q]\!]\}, \sigma \rangle \triangleright \langle \mathtt{error}, \sigma \rangle$ if $\langle I[\![P]\!], \sigma \rangle \triangleright \langle \mathtt{error}, \sigma \rangle$

For case 1, from the operational semantics of the base language:

1.1 $\langle \{I[\![P]\!]\, L : J\ I[\![Q]\!]\}, \sigma \rangle \triangleright \langle I[\![P]\!]\, L : J\ I[\![Q]\!], \sigma \rangle$ by the block rule

1.2 $\langle I[\![P]\!]\, L : J\ I[\![Q]\!], \sigma \rangle \triangleright \langle L : J\ I[\![Q]\!], \sigma \rangle$ by assumption $\langle I[\![P]\!], \sigma \rangle \triangleright \sigma$ and by sequential composition

1.3 $\langle L : J\ I[\![Q]\!], \sigma \rangle = \langle J\ I[\![Q]\!], \sigma \rangle \triangleright \langle I[\![Q]\!], \sigma' \rangle$ by assumption $\langle J, \sigma \rangle \triangleright \sigma'$ and by sequential composition

1.4 $\langle I[\![Q]\!], \sigma' \rangle \triangleright \sigma'$ by assumption

For case 2, the conclusion follows immediately from sequential composition if the precondition throws an error:

2.1 $\langle \{I[\![P]\!]\, L : J\ I[\![Q]\!]\}, \sigma \rangle \triangleright \langle \mathtt{error}, \sigma \rangle$ by $\langle I[\![P]\!], \sigma \rangle \triangleright \langle \mathtt{error}, \sigma \rangle$ and by sequential composition

By introducing error throwing in assertion verification, the program behavior is changed to avoid either returning an arbitrary value from a computation that does not produce a correct one or modifying (corrupting) application files, or even crashing the system.

Advices before and after have been considered only because they provide the required properties to implement pre- and postconditions. Semantic descriptions of other types of advice mechanism (i.e. around, cflow, etc.) have been not considered here.

7 Related Work

Different researches have noted that a join point model with support for local variable crosscutting is necessary. For instance, Usui and Chiba (2005) present Bugdel, a specialized tool for debugging based on aspect-orientation. They extend thisJoinPoint variable with specific information about liner numbers and local variables visible at the join point. In [Rho, et. al., 2006] the authors explore a minimal language design based on fine-grained pointcuts to express structures of the base language through logic-meta variables. This approach is able to express new kind of pointcuts like **`setL()`** and **`getL()`** for local variables. However, with this approach is possible to get substitutions of expressions where sentences are expected and vice-versa. Each join point model extensions address different approaches to solve a particular problem in regard to local variables. Generalizing local variable crosscutting on other application areas may require a better design as it was attempted in Énfasis.

Despite its expressive power, pointcut expressions in this approach tend to be very lengthy and unreadable when join points of interest are selected within deeply nested statements. This is due to the use of logical meta-variables and predicate-like pointcuts that brings in much of the programming style of logic programming. LogicAJ2 [Rho, et. al., 2006] introduces the concept of fine-grained genericity for aspect languages. Based on static analysis and logic-meta variables is possible to construct any new pointcut designators such as `getL` and `setL` for local variables. However, those designators are not capable for selecting specific join points for local variables neither show source code information.

Bugdel [Usui and Chiba, 2005] is an aspect-oriented system designed exclusively for debugging. It provides two pointcut designators: `line(line number)` and `allLines(method name)`. `thisJoinPoint` variable is extended with line and variables to provide information about local variables available at a specific line of source code. In this sense, Énfasis take a step further with a more precise pattern matching mechanism to select local variables and with a join point model for many application areas. Josh [Chiba and Nakagawa, 2004] uses all capabilities of Javassist [Chiba, 2000; Chiba and Nishizawa, 2003] to implements and AspectJ-like language with an extensible pointcut language. The extensions are limited to Javassist capabilities, thus it does not support local variable crosscutting. Although PROSE [Nicoara and Alonso, 2005] was designed to show the potential of dynamic AOP, it is related to Énfasis. Both define aspects in the same base language but PROSE uses nested classes notation.

As mentioned in section 1.2, JML [Leavens et al., 1999] is a BISL (Behavioral Interface Specification Language) specially designed to deal with the assertion checking problem. Unfortunately, because JML are used as annotations introduced in the source code, this approach has the same disadvantages when an application requires that multiple kinds of disparate annotations being used all together though they maybe unrelated, turning the program difficult to read and maintain. Énfasis solve this problem by keeping separated unrelated annotations from the source program and from each other. Unfortunately, this higher degree of modularity comes with a price, called the fragile pointcut problem.

According to Kellens, et. al. (2006), the *fragile pointcut problem* (FPP) occurs in aspect-oriented systems when pointcuts unintentionally capture or miss particular join points as a consequence of their *fragility* with respect to seemingly safe modifications to the base program. Pointcut descriptions as defined in Énfasis depend on the lexical and syntactical structure of the base program because they are based on the AST. Any substantial modification on the base program may turn invalid the path on the AST to the joinpoints of interest. Thus the joinpoints are tightly coupled with the program structure that changes quite often during program evolution, demanding consistent changes in the pointcut description. Though this is a general problem in aspect-oriented programming, in Énfasis seems to be worsen because they depend not only in the variable and method names but also in the method structure. A possible solution to this problem that needs to be further investigated is to consider only stable versions of both base and aspect programs by introducing a versioning identifier in their package name that can be used in the pointcut descriptor as part of the qualified class name. By linking them by means of a versioning identifier, pointcuts may be more specific to indicate upon which base programs they may apply.

Classical pointcut definitions suffer the same FPP problem, but they are related to less frequently changed pieces of code, such as field members or method names. In this respect, a better idea to deal with the FPP problem in Énfasis consists in a combination of both approaches: (1) by placing abstract annotations within the source program with no relation to any use or application and with the sole purpose of indicating the joinpoints of interest, and (2) by keeping apart the Énfasis descriptions that use these abstract annotations. In this regard, the separation of cutpoint descriptions from advice code in AOP programming may be an interesting venue of research to be further investigated.

8 Conclusions

The main contribution of this work is on defining crosscutting on local variables by means of a simple operational interpretation of join points based on the addresses (labels/join points) where the load/store instructions are located. The path and filter expression approach used in Énfasis simplifies the writing of pointcut descriptions and offers an alternative to the current trend of using Prolog-like query languages to define the pointcut semantics of AOP languages.

Unfortunately, Énfasis pointcut descriptions depend on the lexical and syntactical structure of the base program becoming tightly coupled with the program code. A fragile pointcut problem arises when method code changes during program evolution, demanding consistent changes in the pointcut definition. Though this is a general problem in aspect-oriented programming, in Énfasis seems to be worsen because they depend not only in the variable and method names but also in the method structure. A possible solution to this problem that needs to be further investigated is to consider only stable versions of both base and aspect programs by introducing a versioning identifier in their package name. By linking them by means of a versioning identifier, pointcuts may be more specific to indicate upon which base classes they may apply.

It is also convenient to extend the pointcut language and model to improve its expressive power. So far, pointcuts are anonymous leaving out the possibility to the user of defining its own pointcuts. Finally, the Énfasis programming model enjoys several algebraic properties that need to be investigated to establish a mathematical foundation for the model. As a consequence, it could be possible to state and prove useful properties that can be used to transform programs for either verification or optimization purposes.

This paper presented Énfasis, a formal join point model for local variable crosscutting and its implementation in Java. The model is based on set theory where each element represents an insertion point to instrument (weave) Java classes. The model allows: 1) pointcut composition, 2) signature patterns to choose specific variables, and 3) a join point model to generalized local variable crosscutting for different application areas.

9 References

1 **Chiba, S. (2000).** Load-time structural reflection in Java. *Object Oriented Programming ECOOP 2000. Lecture Notes in Computer Science*, 1850, 313–336.

2 **Chiba, S. & Nishizawa, M. (2003).** An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. *Generative Programming and Component Engineering GPCE'03*. *Lecture Notes in Computer Science*, 2830, 364–376.

3 **Cilia, M., Haupt, M., Mezini, M. & Buchmann, A. (2003).** The Convergence of AOP and Active Databases: Towards Reactive Middleware. *Second International Conference on Generative Programming and Component Engineering GPCE'03. Lecture Notes in Computer Science*, 2830, 169-188.

4 **Eaddy, M. & Aho, A. (2006).** Statement Annotations for Fine-Grained Advising. *Workshop on Reflection, AOP and Meta-Data for Software Evolution*, RAM-SE'06 – ECOOP'06, Nantes, France, 89-99.

5 **Floyd, R. W. (1967).** Assigning Meaning to Programs. In J. T. Schwartz (Ed.), *Symposium on Applied Mathematics*, New York, USA, 19, 19-32.

6 **Gosling, J., Joy, B., Steele, Jr. G. L. & Bracha, G. (2005).** *The Java(TM) Language Specification. 3rd ed.* New Jersey: Prentice Hall.

7 **Harbulot, B. & Gurd, J. R. A. (2006).** A Join Point for Loops in AspectJ. *Aspect-Oriented Software Development AOSD'06*, Bonn, Germany, 63-74.

8 **Hilsdale, E. & Hugunin, J. (2004).** Advice Weaving in AspectJ. *3rd International Conference on Aspect-Oriented Software Development AOSD'04*, Lancaster, UK, 26-35.

9 **Hoare, C. A. R. (1969).** An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10), 576-585.

10 **Juárez-Martínez, U. & Olmedo-Aguirre, J. O. (2008).** Énfasis: A Model for Local Variable Crosscutting. *23rd Annual ACM Symposium on Applied Computing SAC 2008*. Ceará, Brazil, 261-265.

11 **Kellens, A., Gybels, K., Brichau, J. & Merrs, K. (2006).** A Model-driven Pointcut Language for more Robust Pointcuts. *Workshop on Software Engineering Properties of Languages and Aspect Technologies SPLAT'06*, Bonn, Germany, 57-63.

12 **Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. G. (2001).** An Overview of AspectJ. *European Conference on Object–Oriented Programming. Lecture Notes in Computer Science*, 2072, 327-353.

13 **Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J. M. & Irwin, J. (1997).** Aspect-Oriented Programming, *ECOOP'97*. *Lecture Notes in Computer Science*, 1241, 220-242.

14 **Leavens, G. T., Baker, A. L. & Ruby, C. (1999).** JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe & W. Harvey (Eds.), *Behavioral Specifications of Business and Systems* (175-188). Massachusetts: Kluwer Academic Publishers.

15 **Meyer, B. (1991).** Applying "Design by Contract". In D. Mandrioli & B. Meyer (Eds.), *Advances in Object-Oriented Software Engineering* (1-50). New Jersey: Prentice Hall.

16 **Nicoara, A. & Alonso, G. (2005).** Dynamic AOP with PROSE. *International Workshop on Adaptive and Self-Managing Enterprise Applications ASIMEA'05*. Porto, Portugal, 125-138.

17 **Reynolds, J. C. (1998).** *Theories of Programming Languages*. Cambridge University Press.

18 **Rho, T., Kniesel, G. & Appeltauer, M. (2006).** Fine-Grained Generic Aspects. *Workshop on Reflection, AOP and Meta-Data for Software Evolution*, RAM-SE'06 – ECOOP'06, Nantes, France, 29-35.

19 **Usui, Y. & Chiba, S. (2005).** Bugdel: An Aspect-Oriented Debugging System. *12th Asia-Pacific Software Engineering Conference APSEC'05*, Taipei, Taiwan, 59-66.

***José Oscar Olmedo Aguirre** received his BSc degree in Physics and Mathematics from the National Polytechnic Institute (IPN) in 1987, his MSc degree in Electrical Engineering from Cinvestav-IPN in 1988 and his PhD in Computer Science from the University of Southampton developing a Coordination Model, Language and Architecture for Open Hypermedia Systems. His research interests include Programming Languages, Concurrency Theories and Models and Web-based Systems. He has been also involved in several projects in SHCP, Pemex, Telmex, Micrologica Aplicada, Camara de Diputados, among others. He is currently Research Professor in the Communications Group of the Electrical Engineering Department at Cinvestav-IPN.*

***Ulises Juárez Martínez** received his BSc degree in Metallurgical Engineering from the National Polytechnic Institute (IPN) in 1994, his MSc degree in Electrical Engineering from Cinvestav-IPN in 1999 and his PhD in Electrical Engineering from Cinvestav IPN in 2008 developing a Framework for Fine-Grained Aspect-Oriented Programming. His research interests include Aspect-Oriented Software Development, Software Engineering, Software Architectures and Compilers. He is currently Research Professor in the Software Engineering Group at Orizaba Institute of Technology, México.*