An Exception Monitoring System for Java[*]

Heejung Ohe and Byeong-Mo Chang

Department of Computer Science
Sookmyung Women's University
Seoul 140-742, Korea
lutino@sookmyung.ac.kr, chang@sookmyung.ac.kr

Abstract. Exception mechanism is important for the development of robust programs to make sure that exceptions are handled appropriately at run-time. In this paper, we develop a dynamic exception monitoring system, which can trace handling and propagation of thrown exceptions in real-time. With this tool, programmers can examine exception handling process in more details and handle exceptions more effectively. Programmers can also trace only interesting exceptions by selecting options before execution. It can also provides profile information after execution, which summarizes exception handling in each method during execution. To reduce performance overhead, we implement the system based on code inlining, and presents some experimental results.

Keywords: Java, exception propagation, exception analysis.

1 Introduction

Exception handling is important in modern software development because it can support the development of robust programs with reliable error detection, and fast error handling. Java provides facilities to allow the programmer to define, throw and catch exceptional conditions. Because uncaught exceptions will abort the program's execution, it is important for the development of robust programs to make sure that exceptions are handled appropriately at run-time. However, it is not easy for programmers to trace and handle exceptions effectively.

A number of static exception analyses including our previous works have been proposed based on static analysis framework [2, 3, 9, 10]. They approximate all possible exceptions, and don't consider runtime exceptions usually. The static analysis information is usually used to check that all uncaught exceptions are specified in the method header. They, however, are not able to provide exact information on how exceptions are thrown, caught and propagated at runtime.

To assist developing robust software, we need a tool to trace or monitor raised exceptions effectively during execution. For example, J2ME Wireless Tool Kit(WTK), a dynamic analysis tool, provides just the names of exceptions, whenever exceptions are thrown, but it cannot trace how thrown exceptions

---

are handled and propagated. In addition, J2ME WTK is too slow when tracing exceptions, because it relies on JVMPI. Programmers cannot trace propagation and handling of exceptions with this tool. To develop reliable and robust Java programs, programmers need a more powerful tool, which can trace exception propagation and exception handling during execution.

In this paper, we develop a dynamic exception monitoring system, which can trace how thrown exceptions(including runtime exceptions) are handled and propagated in real-time. Programmers can examine exception handling process in more details and handle exceptions more effectively. Moreover, programmers can trace only interesting exceptions by selecting options before execution. It also provides profile information after execution, which summarizes exception handling in each method during execution.

To reduce performance overhead, we design the system based on code inlining. Input programs are transformed by inlining codes so as to trace only interesting exceptions according to user options. The transformed programs produce trace information during execution, and profile information after execution. We implement the event monitoring system in Java based on Barat [1], which is a front-end for a Java compiler. We also present some experimental results, which can show the effectiveness of the system.

The rest of this paper is organized as follows. The next section gives preliminaries on exceptions. Section 3 describes overall design of the system. Section 4 describes the implementation of the system, and Section 5 presents some experiments. Section 6 concludes this paper and discusses further research topics.

2   Preliminaries

Like normal objects, exceptions can be defined by classes, instantiated, assigned to variables, passed as parameters, etc. Exception facilities in Java allow the programmer to define, throw and catch exceptional conditions. The `throw` statement `throw` e_0 evaluates e_0 first, and then throws the exception object. The `try` statement `try` S_1 `catch` (c x) S_2 evaluates S_1 first. If the statement S_1 executes normally without thrown exception, the `try-catch` statement executes normally. If an exception is thrown from S_1 and its class is covered by c then the handler expression S_2 is evaluated with the exception object bound to x. If the thrown exception is not covered by class c then the thrown exception continues to propagate back along the call chain until it meets another handler. The programmers have to declare in a method definition any exception class whose exceptions may escape from its body. The formal semantics of Java was proposed in [5] with exception throwing, propagation and handling taken into consideration.

Let's consider a simple example in Figure 1, which shows exception propagation. The thrown exception `E1` from the method `m2` is propagated through `m2` and `m1`, and caught by the `try-catch` in the main method. The exception `E2` may be thrown from the method `m3`. If it is thrown, then it is propagated until the `main` method and not caught. The method `m3` also has a recursive call

```
class Demo{
  public static void main(String[] args ) throws E2
  {
    try {
        m1( );
      } catch (E1 x) {  ; }
      . . .
      m3( );
  }

  void m1( ) throws E1{
     m2( );
  }

  void m2( ) throws E1{
     if (...) throw new E1();
  }

  void m3( ) throws E2 {
     if (...) throw new E2();
     if (...) m3( );
  }
}
```

**Fig. 1.** An example program for exception propagation

to itself, so that the thrown exception E2 may be propagated back through the recursive calls.

Because uncaught exceptions will abort the program's execution, it is important for the development of robust programs to check that exceptions are handled appropriately at run-time.

3   Design considerations

There are several ways that monitors can mediate all application operations. A traditional reference monitor is implemented by halting execution before certain machine instructions and invoking the reference monitor with the instruction as input. An alternate implementation, not limited by hardware support, runs applications inside an interpreter like JVM that executes the application code and invokes a reference monitor before each instruction. JVMPI(JVM Profiler Interface) follows this approach. However, this approach has unacceptable performance overhead [6], since a cost is incurred on every executed instruction. The third option inlines reference monitors in the target software. This approach is shown to overcome the limitations of traditional reference monitors, yet exhibits reasonable performance [6].

An inlined reference monitor is obtained by modifying an application to include the functionality of a reference monitor. As in Figure 2, IRMs are inserted into applications by a rewriter or transformer that reads a target application and a policy, and produces a secured application, whose execution monitors its execution. The inlining approach is shown to be efficient for monitoring Java programs in [6].
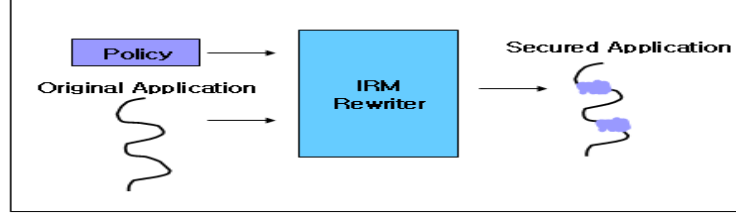
Fig. 2. Inlined monitor [6]

We follow the inlining approach to design a dynamic exception monitoring system for efficiency. We take the following things into consideration in the design.

The first one is to provide users with options to select interesting exceptions. By selecting options before execution, users can focus on interesting exceptions and methods by tracing only interesting exceptions in real-time. This option can also contribute in reducing performance overhead, because it makes the system to trace only interesting exceptions instead of all exceptions. The second one is to provide a profile option to produce profile information after execution, which summarizes exception throwing and handling during execution. The third one is to reduce performance overhead. We try to reduce performance overhead by inlining code instead of using JVMPI. An input program P is transformed into a program P' by inlinig codes so as to trace only interesting exceptions according to user options. The transformed program P' will trace how thrown exceptions are handled and propagated during execution, and give profile information on exception handling after execution.

Overall architecture of the system is shown in Figure 3. This system consists of four steps as in Figure 3. The function of each step is as follows:

The first step extracts exception-related constructs by static analysis. This static information is used to give users options. The second step is option selection, where users can select interesting exceptions and methods using the static exception information. Users can trace only interesting exceptions and methods by selecting options in this step. The third step is a transformer, which transforms an input program P into a program P' by inlining codes so as to trace only interesting exceptions according to user options. The fourth step is to compile and execute the transformed program P'. It is to be executed on Java 2 SDK or J2ME WTK.
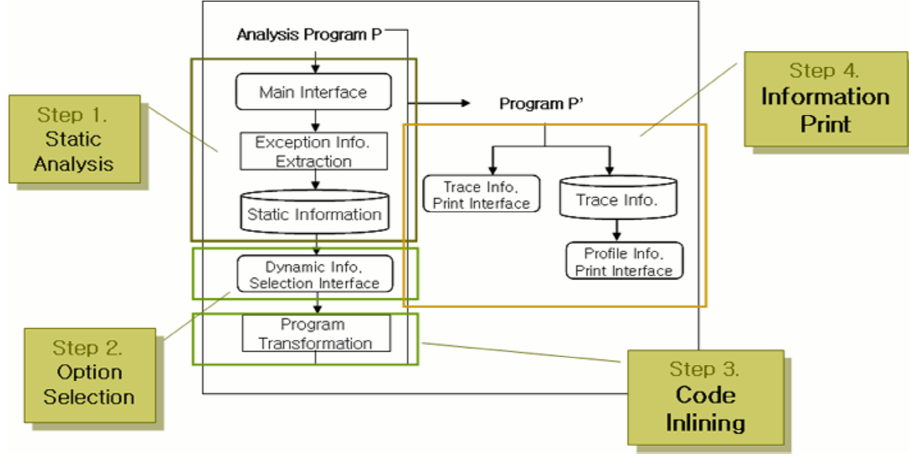
Fig. 3. System architecture

4 Implementation

The exception monitoring system is implemented in Java based on Barat [1], which is a front-end for a Java compiler. Barat builds an abstract syntax tree for an input Java program and enriches it with type and name analysis information. It also provides interfaces for traversing abstract syntax trees, based on
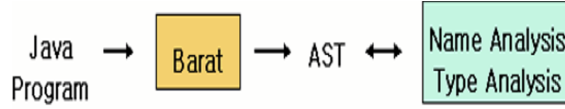

Fig. 4. Architecture of Barat

visitor design pattern in [7]. We can traverse AST nodes and do some actions or operations at visiting each node using a visitor, which is a tree traverse routine based on design patterns. Barat provides several visitors as basic visitors: `DescendingVisitor` which traverses every AST node in depth-first order and `OuputVisitor` which outputs input programs by traversing AST nodes. We can develop a static analyzer by implementing visitors to do necessary actions or operations at visiting AST nodes by extending basic visitors [1].

As described in Figure 3, our system consists of four steps. We implement the first three steps. The last step is a real execution on Java SDK.

A main window for selecting options is shown in Figure 5, which shows all files in the package of a benchmark program *Check* from specjvm98. After users select files from the package, the window displays a list of exceptions, handlers and methods based on the static analysis information. Then, users can select
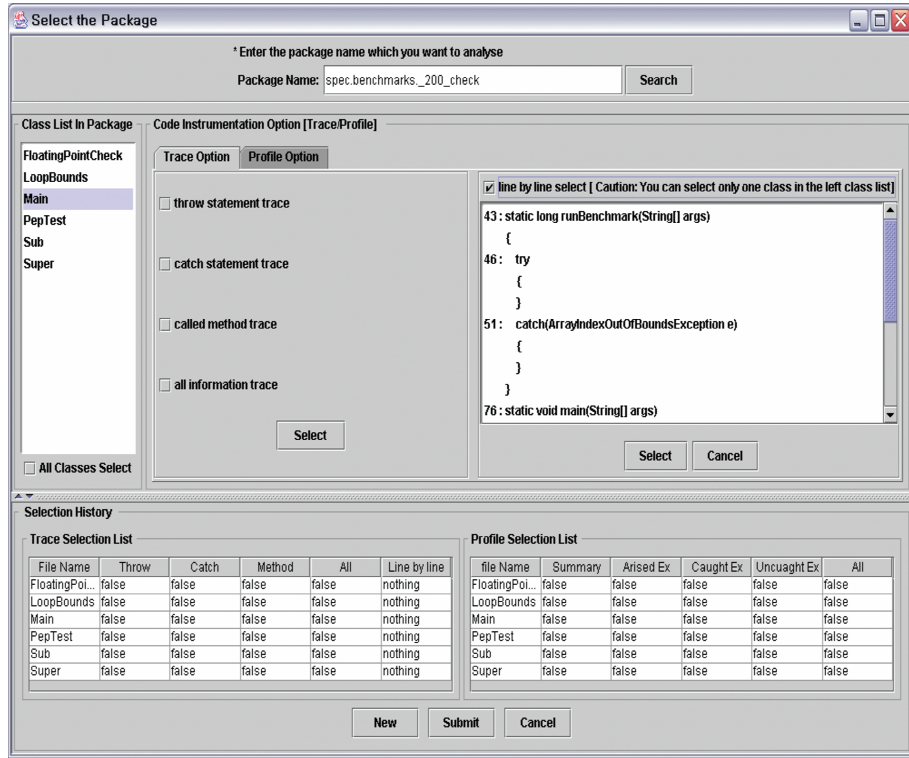
Fig. 5. Menu window

only interesting exceptions, handlers and methods among them. By selecting options, users can get only interesting trace information and focus on interesting exceptions when debugging.

To provide users with options, we implement a static analyzer to extract exception-related constructs by extending `DescendingVisitor`. It extracts static information about possible exceptions and methods by analyzing input programs statically. In particular, it extracts static program constructs on exception raising, handling and methods.

We implement a program transformer called `TransformVisitor` by extending `OuputVisitor`, which transforms an input program P into a program P' by inlining codes so as to trace handling and propagation of thrown exceptions according to selected options. Figure 6 shows overall structure of the program transformer.

A fragment of the transformed `Check` program is displayed in Figure 7. This figure shows `main` method and `syncMethod2` which is called from the main method. The codes in the box are inlined codes by the transformer. The transformed program is to be executed automatically in Java 2 SDK. This transformed code traces how thrown exceptions are handled and propagated in real-time dur-

```
Class TransformVisitor extends OutputVisitor{
    visitThrow{
    // Print the location of thrown exception and its type
    }
    visitTry{
    // Print the location of try statement
    }
    visitCatch{
    // Print the location of catch statement, the type of caught exception
    // call printStackTrace() method
    }
    visitMethod{
    // Print the method information and record exception propagation via this method
    }
}
```

Fig. 6. Structure of TransformVisitor

ing execution. In addition, it can also profile exception handling of each method during execution and shows the number of thrown, caught, and uncaught exceptions for each method after execution.

5   Experiments

We have implemented the system with SDK 1.4.2 on Pentium 4 processor and Window XP. We first tested it with *Check* from specjvm98, which is a program to check that the JVM fits the requirements of the benchmark.

When we execute the transformed program, we can trace handling and propagation of thrown exceptions (including runtime exceptions) as in Figure 8. It shows the location and exception type when an exception is thrown. It also shows the propagated path when an propagated exception is caught by a `catch`-clause. For example, the propagation path of `ArithmeticException`, when it is caught at line 659, is shown in Figure 8 When the program terminates, it profiles the thrown, caught, and propagated exceptions of each method. Figure 9 shows the names and numbers of thrown, caught and propagated exceptions of each method. It also shows the name of exceptions, which is specified by `throws` clause at method headers. For example, `ArithmeticException` is specified at `PepTest.syncMethod2`.

We have experimented the system with five Java benchmark programs. The first one is the small server connecting program *ServerStuff*. The second one is *Linpack* benchmark, which is to solve a dense system of linear equations. The third one is *Check* from specjvm98. The fourth one is *Jess* from specjvm98, which is an expert system shell based on NASA's CLIPS program. The last one is *Rex* from gnu, which matches a regular expression against strings.

```
28    synchronized int syncMethod2(int y) throws ArithmeticException {
29
30         pi.print_methodtrace("syncTest","syncMethod2","27","synchronized null syncMethod2(int y) throws Arithme
31         pi.methodcollect("syncTest","syncMethod2","|ArithmeticException|");
32    this.x = this.x + y;
33
34    if (this.x == 99){
35         pi.print_throwtrace("syncTest","syncMethod2","30","new ArithmeticException");
36         pi.set_ThrowInfo("syncTest" , "syncMethod2" , "30" ,"new ArithmeticException");
37         throw (new ArithmeticException("fisk"));
38    }
39
40    return this.x;
41  }
42
43  public static void main(String[] args) {
44         try{
45
46         pi.print_methodtrace("syncTest","main","34","static void main(String[] args)");
47         pi.methodcollect("syncTest","main","|");
48    syncTest sy = new syncTest();
49    int xx = sy.syncMethod(4);
50    xx = sy.syncMethod2(4);
51         }catch(Exception apple)  {
52             apple.printStackTrace();
53         }
54         finally{
55             pi.print_Profile();
56         }
57  }
```
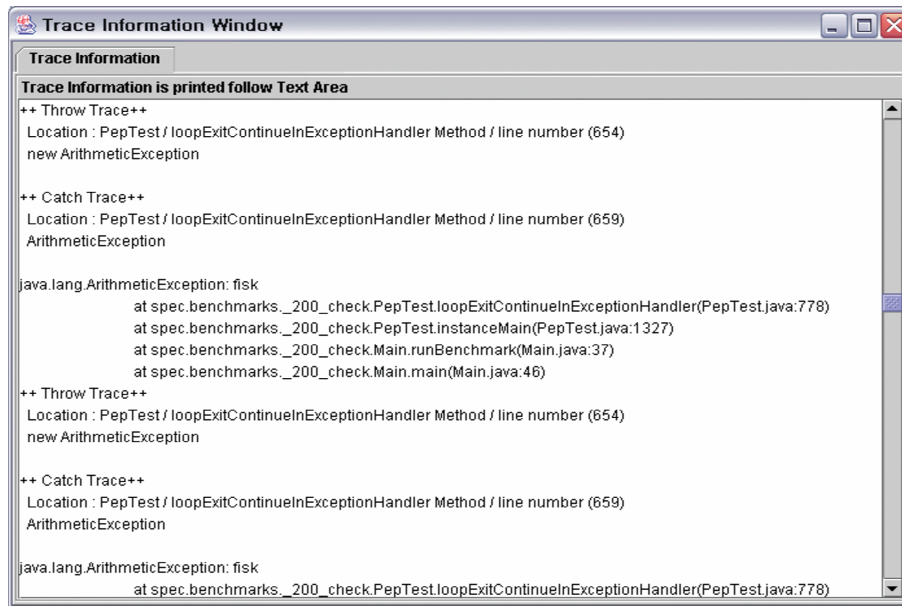
Fig. 7. Transformed program

Fig. 8. Trace of Check program

Profile of thrown exception

** Profile Information **

Class.Method	Thrown exceptions (no.)	Caught exceptions (no.)	Propagated exceptions (no.)	Specifications
PepTest.testExc7		class java.lang.ArithmeticException(1)	class java.lang.ArithmeticException(2),	
LoopBounds.execute				
PepTest.testExc6		class java.lang.ArithmeticException(1)	class java.lang.ArithmeticException(1),	
PepTest.testExc5	exc(1)	class java.lang.ArithmeticException(2)		
	ArithmeticException(1)			
Main.checkSubclassing				
PepTest.throwArithmetic...	ArithmeticException(2)			
PepTest.testExc4	RuntimeException(1)	class java.lang.RuntimeException(1)		
Super.getName				
PepTest.instanceMain				
PepTest.testExc3	ArithmeticException(1)	class java.lang.ArithmeticException(1)		
LoopBounds2.size				
PepTest.testExc2	RuntimeException(1)	class java.lang.RuntimeException(1)		
PepTest.isPrime				
Main.runBenchmark		class java.lang.ArrayIndexOutOfBoundsExceptio...		
PepTest.testExc1	ArithmeticException(1)	class java.lang.ArrayIndexOutOfBoundsExceptio...		
		class java.lang.ArithmeticException(1)		
LoopBounds.size				
PepTest.testHiddenField				
PepTest.Verify				
PepTest.testLookupSwitch				
PepTest.testArray		class java.lang.ArrayIndexOutOfBoundsExceptio...		
⋮				
LoopBounds.run				
PepTest.checkRemainde...				
PepTest.checkRemD				
Main.main				
PepTest.testBitOps				
StringAndInt.clone				
LoopBounds.constraintAt				
PepTest.syncMethod2				ArithmeticException
total methods (62)	total thrown exceptions(107)	total caught exceptions(122)	total propagated exceptions(9)	total specifications (1)

Fig. 9. Profile of Check program

Table 1 first shows the numbers of lines of benchmark programs before and after inlining, and then shows the number of thrown exceptions, the number of caught exceptions, and the number of propagated exceptions for each benchmark. If an exception is propagated, it is counted as propagated exceptions at every method, through which it is propagated back. So a propagated exception can be counted multiply.

The listed figures of *Jess* represents propagation of a thrown exception when an input with wrong syntax is given. The listed figures of *Rex* also represents propagation of a thrown exception when a wrong option is given. The listed figures of *Check* represents the many numbers of thrown, handled and propagated exceptions.

6 Related works

In [10, 11], the usage patterns of exception-handling constructs in Java programs were studied to show that exception-handling constructs are used frequently in Java programs and more accurate exception flow information is necessary.

Programs	Lines(before)	Lines(after)	Throw	Caught	Propagated
ServerStuff	71	104	2	2	4
Linpack	1057	1103	1	1	17
Check	1817	1898	107	122	9
Jess	542	574	1	1	10
Rex	3198	3308	1	1	6

Table 1. Experiments with benchmark programs

Exception analyses have been studied actively based on static analysis framework [2, 3, 12, 9]. Static exception analyses analyze input programs before execution and provide approximate information about all possible uncaught exceptions of each method. In Java[8], the JDK compiler ensures, by an intraprocedural analysis, that clients of a method either handle the exceptions declared by that method, or explicitly specify them at method header. In [9], a tool called Jex was developed to analyze uncaught exceptions in Java. It can extract the uncaught exceptions in Java programs, and generate views of the exception structure.

In our previous work [2, 12], we proposed interprocedural exception analysis that estimates uncaught exceptions independently of programmers's specified exceptions. We compared our analysis with JDK-style analysis by experiments on realistic Java programs. We also have shown that our analysis can detect uncaught exceptions, unnecessary `catch` and `throws` clauses effectively.

Static analysis techniques, however, cannot provide information about actual execution. So, dynamic analysis techniques have also been studied to provide information about actual execution [4, 13, 14]. Several dynamic analysis tools are developed for Java including J2ME Wireless Toolkit [14] and AdaptJ [13]. Recent J2ME Wireless Toolkit can trace method calls, exceptions and class loading as well as memory usage using JVMPI. However, it provides just the names of exceptions whenever exceptions are thrown. Moreover, JVMPI imposes heavy burden on performance overhead, which makes execution speed too slow. It is hard to trace interesting parts of programs effectively, because all codes including libraries are included in the trace. AdaptJ don't provide any exception-related information during execution.

Our current work differs from the previous static works in that the previous works focus on estimating uncaught exceptions rather than providing information on the propagation paths of thrown exceptions. Our monitoring system can trace in real-time how thrown exceptions including runtime exceptions are handled and propagated during execution. This trace function has not been supported by any dynamic systems yet.

7   Conclusion

We have developed a dynamic exception monitoring system, which can help programmers trace and handle exceptions effectively. Using this system, program-

mers can examine exception handling process in more details by tracing only interesting exceptions, and can handle exceptions more effectively. To reduce performance overhead, we have designed the system based on inlined reference monitor. We are extending this system in two directions. The first one is to visualize exception trace and profile information, which can give more insights to programmers. The second one is to adapt this system to J2ME programs, which are widely used in mobile environment.

References

1. B. Bokowski, Andre Spiegel. Barat A Front-End for Java. Technical Report B-98-09 December 1998.
2. B.-M. Chang, J. Jo, K. Yi, and K. Choe, Interprocedural Exception Analysis for Java, *Proceedings of ACM Symposium on Applied Computing*, pp 620-625, Mar. 2001.
3. J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, Efficient and precise modeling of exceptions for analysis of Java programs, *Proceedings of '99 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, September 1999, pp. 21-31.
4. B. Dufour, K. Driesen, L. Hendren and C. Verbrugge. Dynamic Metrics for Java. *Proceedings of ACM OOPSLA '03*, October, 2003, Anaheim, CA.
5. S. Drossopoulou, and T. Valkevych, Java type soundness revisited. Techical Report, Imperial College, November 1999. Also available from: http://www-doc.ic.ac.uk/ scd.
6. U. Erlingsson, *The inlined reference monitor approach to secure policy enforcement*, Ph.D thesis, Cornell University, January 2004.
7. E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns:Elements of Reusable Object-Oriented Software*, Addison-Wesley,1995.
8. J. Gosling, B. Joy, and G. Steele, *The Java Programming Language Specification*, Addison-Wesley,1996.
9. M. P. Robillard and G. C. Murphy, Analyzing exception flow in Java programs, in *Proc. of '99 European Software Engineering Conference and ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 322-337.
10. B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah, A static study of Java exceptions using JESP, Tech. Rep. DCS-TR-403, Rutgers University, Nov. 1999.
11. S. Sinha and M. Harrold, Analysis and testing of programs with exception-handling constructs, *IEEE Transations on Software Engineering* 26(9) (2000).
12. K. Yi and B.-M. Chang Exception analysis for Java, ECOOP Workshop on Formal Techniques for Java Programs , June 1999, Lisbon, Portugal.
13. AdaptJ:A Dynamic Application Profiling Toolkit for Java, http://www.sable.mcgill.ca/ bdufou1/AdaptJ
14. Sun Microsystems, J2ME Wireless Toolkit, http://java.sun.com.