Mutant Operators for Testing Concurrent Java Programs

Márcio Delamaro
Universidade Estadual de Maringá (UEM)
delamaro@din.uem.br

Mauro Pezzè
Università degli Studi di Milano
pezze@disco.unimib.it

Auri M. R. Vincenzi
José Carlos Maldonado
Universidade de São Paulo (USP)
{jcmaldon,auri}@icmc.sc.usp.br

Abstract

Mutation testing is a fault-based testing technique that has been widely studied in the last decades. One reason for the interest of the scientific community in mutation testing is its flexibility. It can be applied to programs at unit and integration testing levels, as well as to software specifications written in a variety of different languages. A fundamental issue to make mutation testing work for a given language or environment is the set of mutant operators used to create the mutants. In this paper a set o mutant operator is proposed for the Java programming language, and more specifically, aiming at exercising aspects of concurrency and synchronization of that language.

1 Introduction

Testing is a crucial activity in the software lifecycle. It is expensive and time consuming. For this reason much effort has been spent on developing techniques and tools to support the testing activity. An important result of research is the definition techniques and criteria to drive the generation of test sets that can suitably exercise a program.

Mutation testing is a fault based test technique. It uses a set of "rules" called **mutant operators** to create programs slightly different from the program under test. These programs are called **mutants**. The goal of mutation testing is the generation of a test set that distinguishes the behavior of the mutants from the original program. The ratio of distinguished mutants (also called **dead** or **killed** mutants), over the total number of mutants, measures the adequacy of the test set.

According to the **coupling effect** hypothesis[14], test cases that distinguish simple faults injected in the original program to create the mutants should also be able to reveal faults that can be obtained as a composition of simple faults.

Thus, mutant operators can be seen as representing common faults usually found in software.

Mutant operators depend on the language in which the artifact to be tested is described. So far mutation testing has been applied to programs written in several programming languages, like Fortran [5], C [3] and Java [11]. and to formal specifications written using Finite State Machines [7], Petri Nets [8], Statecharts [6] and Estelle [20, 18]. This flexibility derives from the fact that mutation testing requires only an executable model that transforms inputs into observable outputs that can be compare against the results produced by the mutants.

Mutant operators are designed referring to the experience of using the target language and of the most common faults. In the past mutant operators were designed based experts' knowledge. Recently, Kim et al. [10] have proposed the use of a technique named "Hazard and Operability Studies" (HAZOP) to systematically derive mutant operators and they applied it to the Java language. Although the resulting operators do not significantly differ from past works, the proposed methodology is an important step towards a more rigorous discipline in the creation of mutant operators. The technique is based on two main concepts. It first identifies in the grammar of the taget language (Java in this case) the points subject to mutation and then defines the set of mutations these points based on suuitable "Guide Words".

In general, a comprehensive set of mutant operators, that well cover the structures of a given language tends to produce too many mutants even for small programs. This problem has been tackled by selecting subsets of mutants according to different strategies [21, 16, 13] like randomly selecting a percentage of the generated mutants or choosing a subset of the mutant operators based on some known characteristic like cost or tendency to generate equivalent mutants.

The test of concurrent programs is complicated by the nondeterminism of their execution, that in the case of mutation testing, makes it difficult to analyze the behavior of the mutants. Since a program in this context can present different correct results, the fact that the behavior of a mutant differs from the result of the original program for a test case t does not mean that t has revealed the fault in the mutant. The discrepancy may result from the non determinism of the program and not from the fault seeded in the mutant.

Two different approaches to apply mutation testing to concurrent programs have been proposed in the literature [17, 19]. Both have ADA as their target language and are presented with more details in the next section. Unfortunately the approaches do not extend straightforwardly to other languages, such as Java, that are not based on the Ada rendezvous model. A pilot study that used a pre-existent mutant operator set confirmed the need of mutant operators specifically designed for the Java model of concurrency and synchronization [2].

This paper describes part of the work that aims at applying mutation testing to Java concurrent programs. Herein we describe the design of mutant operators. In Section 2 we overview previous work on mutation testing for concurrent programs, and in particular the work of Silva-Barradas [19] that is closely related to our goals. In Section 3 we briefly explains the mechanisms used for synchronization and concurrency in Java. In Section 4 we present the set of mutant operators designed to exercise those mechanisms. The conclusions and ongoing works related to this paper are presented in Section 5.

2 Mutation Testing for Concurrent Programs

When testing a sequential program, one can rely on the fact that there exists only one correct output for a given test data. The statements are executed in a determined order that does not change from one execution to another, if the input is the same. This is not true for concurrent programs. The parallel or concurrent execution of several (deterministic) processes may result in different outputs, depending on the order of execution of the different processes.

This is particularly troublesome for mutation testing. When a mutant M is executed with a test case t, it should be killed if its behaviour is incorrect, according to a given specification. In the case of a sequential program this is equivalent to say that the results of the execution of M with t differs from the results of the execution of the original program P with t. In the case of concurrent programs we should compare the set of all possible results of M with t against the set of all possible results of P with t and then consider M distinguished if they are not the same.

Unfortunately, in general it is not possible to know the set of all possible results of a concurrent program for a given input. Offut et al. [17] proposed a technique that calculates an approximated set Ω, defined as a subset of all possible results of $P(t)$. This set is calculated by executing $P(t)$ several times. The mutant M is distinguished if it produces at least one result $M(t) \notin \Omega$.

Another approach, called "Behavior Mutation Analysis" (BMA), is defined by Silva-Barradas [19]. A suitable instrumentation of an ADA program P can be used to register the sequence of execution of its tasks. The recorded sequence can be used to reproduce the same behavior in a subsequent execution of $P(t)$. Incorporating this synchronization sequence as part of a test case, it is expected that $M(t, s) = P(t, s)$, i.e., the result of the execution of the mutant M with t reproducing the same synchronization sequence should be the same of the original program. The BMA technique considers the mutant distinguished if one of the two conditions is verified: 1) $M(t, s) \neq P(t, s)$; or 2) M cannot reproduce the synchronization sequence s. The second case may occur, for example, if the mutation eliminates a statement that when executed adds an event to the synchronization sequence, e.g., a entry call. In this case, BMA considers that the mutant has an incorrect behavior (deadlock or abnormal termination) and thus should be distinguished.

In a pilot study [2] we tried this approach using existing operators [4] adapted to Java and we could derive some important observations that are guiding the current development of the technique for the Java language.

- A mutant can be distinguished by either producing a different output or because it cannot reproduce the same synchronization sequence of the original program. In general a mutant that can be distinguished by one reason cannot be distinguished by the other. In a few cases though it is possible to find mutants that can be distinguished from the original program both because producing a different output for a test case t_1 and because not able to reproduce the required synchronization sequence for a different test case t_2. Although in general mutants should be killed if distinguished in either ways, for such mutants it would be useful to require both cases for killing them, since the different test cases can reveal different faults;
- The number of equivalent mutants, i.e., mutants that cannot be distinguished by either reasons, is very low. In the pilot study only 2 out of the 144 mutants

```
public class myclass {

    public synchronized void myMethod()
    {
        doSomething();
    }

    public void myOtherMethod()
    {
        doSomeOtherThing();
    }
}
```

Figure 1: Example of synchronized method

created were equivalents;

- It is harder to analyze mutants when concurrency is involved, mainly when it is necessary to induce certain specific synchronization sequence to distinguish them. A useful help in this matter would be the construction of a tool that could generate different synchronization sequences to test the program. We already know that a tool is necessary to record the synchronization sequences of the test cases so they can be reproduced in the mutant. In the ongoing work of developing such tool we are studying ways to create valid variants of an original synchronization sequence. Such variants can be useful for testing the program and killing the mutants.
- The set of mutant operators, based on the Interface Mutation operators [4], used in the pilot study is not sufficient. Although we could not formally evaluate the quality of the test set obtained – because no other criterion exists (in our best knowledge) against which we could evaluate it – we could intuitively note that some characteristics (either functional and structural) related to concurrency where not exercised and an existing bug could not be revealed.

In the next section we summarize the features of the Java language related to concurrency and synchronization and that conduced to the mutant operator set described in Section 4.

3 Overview of Java Concurrency Mechanisms

The basic mechanism for thread synchronization in Java are synchronized methods and blocks. Every object in Java has a **monitor** associated to it. This monitor is used to guarantee that only one thread at time has access to the object. Figure 1 shows an example of synchronized method. In a program with two or more threads sharing an object X of type `myClass`, only one at a time can enter `myMethod` using that object. The access to `myOtherMethod` does not have such restriction. So a thread T_1 can execute, for instance, `myMethod` on object X concurrently with thread T_2, executing `myOtherThread` on the same object X.

If a given thread T_1 is executing a synchronized method on an object X and another thread T_2 tries to enter a synchronized method on the same object the

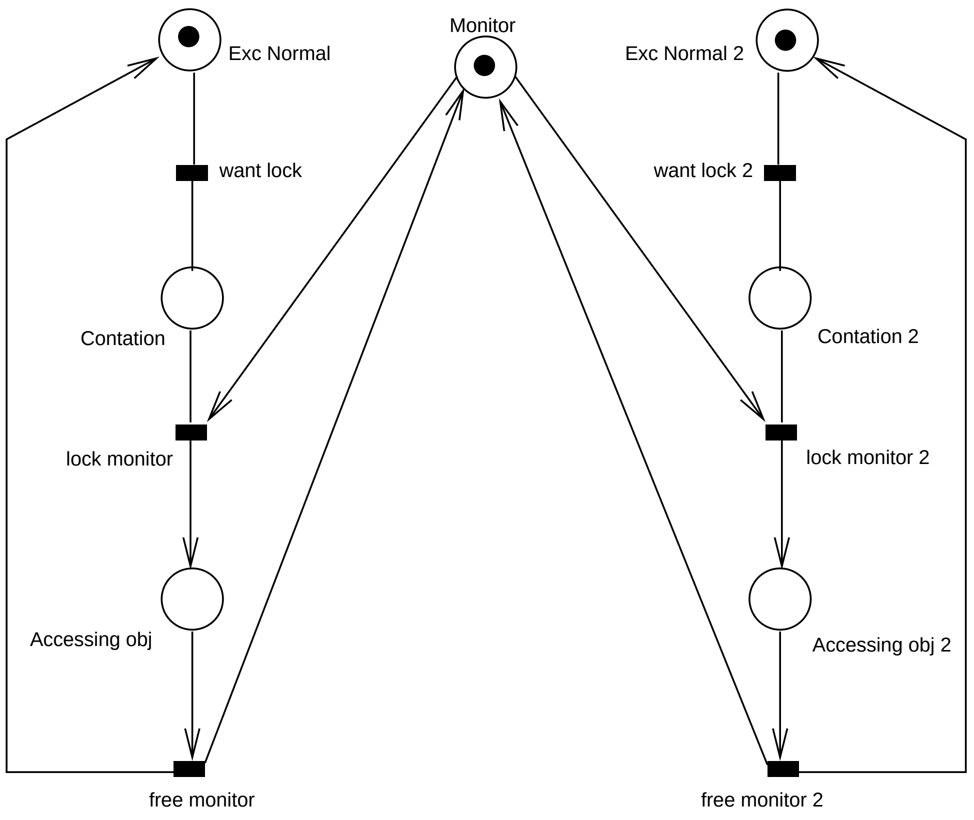

Figure 2: Basic model of 2 threads sharing an object

system blocks T_2 until T_1 terminates executing the synchronized method. Then T_2 can execute, or more precisely, it can compete to get access to the object, since other threads may also have been trying to access the same object within a synchronized method.

When a thread T_1 gains access to a synchronized method of an object X we say that T_1 owns or has locked X's monitor. A static method can be synchronized. In this case entering the method locks the monitor associated to the `Class` object associated to the class where the method is defined.

Figure 2 shows a simple Petri Net model of two threads that share an object X. The places named `Exec Normal` represent the states of each thread when they are executing non synchronized code w.r.t. the object of interest. A thread may want to execute a synchronized method on X and then it reaches a contention state, where it will lock the monitor, if it is available. On succeeding, the token is removed from the monitor place and is holden until the thread finishes the synchronized method and releases the monitor.

Synchronized blocks are similar to synchronized methods but protected code is restricted only to a piece of a method and the object on which the lock is executed is explicitly declared. For example, in the code in Figure 3, the thread must obtain the monitor of `myObject` before executing `doSomeOtherThing`, in method `myOtherMethod`.

Every object created in the JVM has associated a **wait set** which allows a thread in possess of a monitor to temporarily release that monitor until an event occurs. Class `Object` defines a `wait` method that releases the monitor of the object used in

```
public class myclass {
myOtherClass myObject = new myOtherClass();

    public synchronized void myMethod()
    {
        doSomething();
    }

    public void myOtherMethod()
    {
        synchronized (myObject)
        {
           doSomeOtherThing();
        }
    }
}
```

Figure 3: Example of synchronized block

the call and insert the current thread in the object wait set, temporarily blocking its execution. The definition of method `wait` explicitly states that the thread that calls the `wait` must be holding the monitor of the object, otherwise an exception is thrown. Since every class in Java inherits from class `Object`, every object created in Java has a method `wait` (declared final in the class `Object`).

Class `Object` also declares a method `notify` which wakes up a thread waiting in a wait set. When this method is called on an object X one of the threads is randomly removed from the wait set, becoming ready to execute. This does not mean that it will be executed immediately. First because the thread that calls `notify` must be in possess of the object monitor, thus the thread removed from the wait set can execute only after the current thread releases the lock. Moreover, even after the current thread has released the lock, there is no guarantee that that specific thread will be the one that acquires the monitor because other threads may be in contention for the same monitor.

The thread removed from the wait set will resume from the point immediately after the call to `wait` and its state returns to the same it had at that time, i.e., in possess of the monitor and with the same lock count it had on that monitor. For example, if it had locked the monitor twice, it reacquires the monitor with lock count equals two. Figure 4 shows the example of two threads that share an object, including the call of method `wait`. Note that the model becomes significantly more complex than that shown in Figure 2.

The `wait` method has a variant that accepts as argument a timeout value. The semantics of this call is similar to the one described above, i.e., the thread is inserted in the wait set until a `notify` removes it. In addition, if it is not removed by a `notify` within the specified timeout, it is removed “by itself”. The `notify` method has a variant `notifyAll` that, when called on object X, removes all the threads from the wait set of object X.

Other methods in the API complete the resources of the language for concurrency. Among them, the methods in the class `Thread`, responsible for the creation and management of new threads.

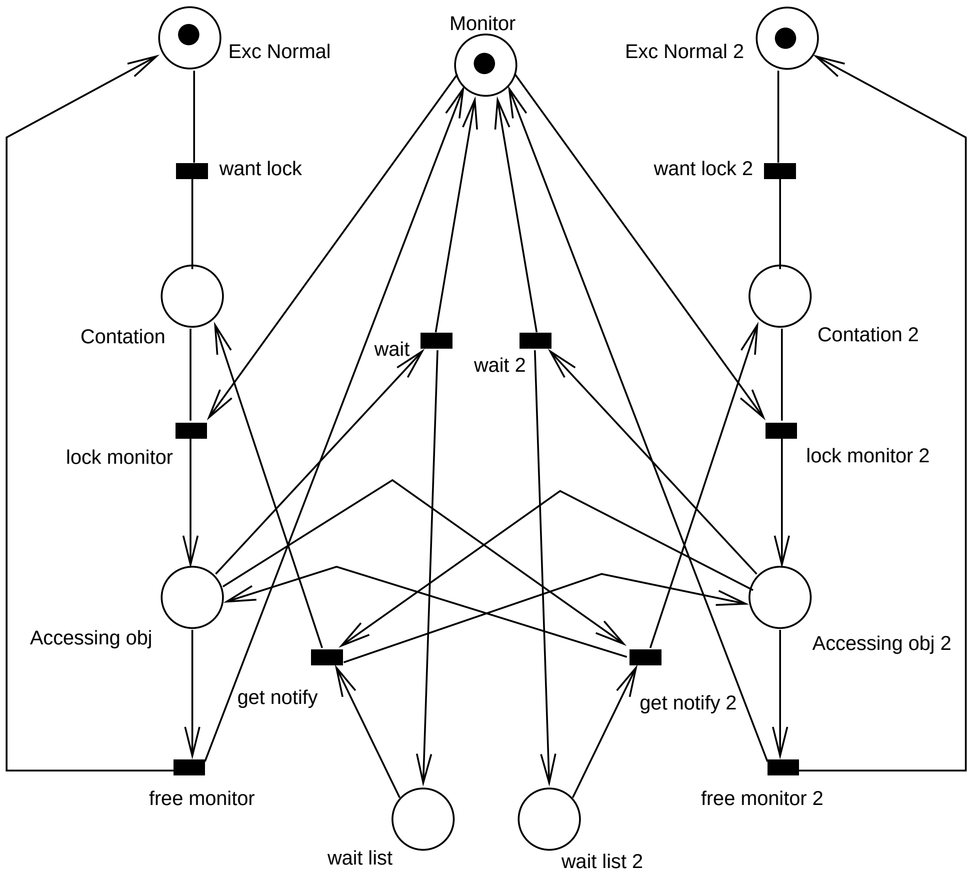

Figure 4: Model of two threads sharing an object, including wait sets

4 Mutant Operators for Java

In this section we present a set of mutant operators designed to exercise the concurrency and synchronization aspects of Java programs. First we identify the main structures related to concurrency and then we derive mutant operators to exercise such structures. One essential aspect taken in consideration is that the practical applicability on mutation testing heavily depends on the number of mutants generated by the approach.

Based on the overview presented in Section 3 we identified four groups of structures of Java related to concurrency:

- Monitor lock code;
- Methods related to wait set manipulation that are defined in the Java core API;
- Use of synchronized methods;
- Use of other methods related to synchronization and concurrency.

Each of these classes are analyzed and mutation operator to exercise them are proposed in the following sections.

```
        synchronized public void append (Element x)
        {
            while (cont >= buffer.length)
                wait();
            buffer[nextSpot] = x;
            nextSpot = (nextSpot + 1) % buffer.length;
//          nextSpot = (nextSpot * 1) % buffer.length;  <<== mutation
            cont++;
            notify();
        }
```

Figure 5: A synchronized method and a possible mutant

4.1 Monitor locks

The class of monitor locks include synchronized methods and statements. A method is declared synchronized adding the keyword `synchronized` in front of its declaration, for example

```
synchronized public foo(int k, String s)
{
    ...
}
```

The first mutant operator identified for this construc is the deletion of the keyword `synchronized`, turning the method to "normal", regardless of the monitor acquisition. This will require test cases that exercise the concurrent use of each synchronized method.

The objective of a mutation is to create a program with behavior that slightly differs fro the behavior of the original program. In the particular case of synchronized methods, we want a method that when called will produce a different behavior related to the access to a shared object. This could be achieved by mutating any statement in the body of the method. We could use mutant operators similar to those previously defined for unit testing [1, 12, 11] to obtain the discrepant behavior. For example, the code in Figure 5 shows a simple method that inserts an element in a circular shared buffer, used in a producer-consumer system, and a possible mutant.

If on one hand this mutant may help to exercise important features in the program, forcing for example a test case that uses a buffer larger than one element, on the other hand all such mutants would make the test impractical. First because the number of mutants would grow to an undesirable level. Second, because these "fine grained" mutants would change the state of the program in such a way that would make very hard for the tester to analyze them. In a pilot study [2], the use of Interface Mutation operators [4] confirmed these two conjectures. To avoid these problems we define only one type of mutation to be applied in the body of a synchronized method, i.e., statement deletion, as defined in [1].

This operator removes one statement at each time. In this way we reduce the number of mutants and create mutants that are easier to be analyzed because they represent more significant and understandable changes in the code. Using the definition of Offut and Hayes [15], we have mutants with a larger syntactic difference from to the original program and as side effect it is expected also a larger semantic difference.

A synchronized statement is defined in [9] as

SynchronizedStatement: **synchronized** (*Expression*) *Block*

The two mutant operators defined for synchronized methods can be applied. The first removes the declaration of the block as synchronized and the second deletes a statement inside the block. We can also apply operator "move brace up/down" as defined by Agrawal et al. [1] to the synchronized block. This operator moves the closing brace one statement "up" moving the last statement out of the block or one statement "down" including in the block the first statement declared outside. We can apply a similar operator to the beginning of the block, excluding the first statement or inserting the immediately previous one. These operators generate only few mutants (at most four per synchronized block) and are a more subtle version of the statement deletion operator. Test cases that distinguish such mutants might be useful to verify the limits that a synchronized statement must have.

The semantic of a synchronized statement indicates that its synchronization expression should evaluate to an object, i.e., an instance of `java.lang.Object` or one of its subclasses. To guarantee that the programmer used the correct expression, i.e., that the synchronization is being done in the right object we can replace the synchronization expression by other objects accessible in that scope. So we define four mutant operators:

- replace the synchronization expression with a local variable;
- replace the synchronization expression with a formal parameter;
- replace the synchronization expression with a static field; and
- replace the synchronization expression with an instance field;

4.2 Wait sets

In this section we present the operators designed to exercise another main feature of the Java synchronization mechanism: the wait sets. Basically, the wait sets are managed by two types of methods: 1) those that ask the system to block a thread inserting it in a wait set (`wait` and timed `wait` methods); and 2) those that ask the system to remove threads from a wait set (`notify` and `notifyAll`).

The mutant operators designed to deal with the use of the `wait` methods are:

- Delete a call to `wait`. In this case the thread that should be blocked continues executing. Test cases that kill this kind of mutants show the relevance of each `wait` call. This may be particularly important for timed `wait` that may require very specific test cases to be effectively used.
- Replace a timed `wait` with a non-timed.
- Replace a non-timed `wait` with a timed.
- Increment and decrement time in a timed `wait`. These last three operators exercise timing aspects of a program.

For exercising the use of `notify` methods we define:

- Delete a call to `notify` or `notifyAll`. This operator changes the set of threads available for execution at a given point of the execution. Test cases must be provided in order to show that each of such calls is really necessary.
- Replace a call to `notify` with a call to `notifyAll`.
- Replace a call to `notifyAll` with a call to `notify`. These two operators might be useful for testing synchronization aspects because test cases that distinguish them may require the execution of very specific sequences of `wait` and `notify` statements, otherwise difficult to exercise.

4.3 Synchronized method calls

The operators in this class aim at exercising the interactions with methods declared as synchronized as they are a main feature of the Java language, for what concern synchronization. Note that these operators differ from those defined in Section 4.1 because those are related to the declaration of synchronized methods and the ones defined here are related to the use of such methods, i.e., the invocation of such methods.

The type of interactions we want to exercise here are similar to those exercised by the Interface Mutation approach [4]. Interface Mutation defines a set of mutant operators that focus on the interaction between two units, e.g., two functions in the C language. Here we want to specifically exercise the use of synchronized methods, changing the points where they are invoked. Based on the set of Interface Mutation operators we define the following operators:

- Replace arguments with constants. Each argument is replaced with constants defined according to the type of the argument. For example `int` arguments can be replaced with {0, 1, -1, Integer.MAX_VALUE, Integer.MIN_VALUE} where MAX_VALUE and MIN_VALUE are respectively the largest positive integer and the smallest negative integer as defined by the Java API. The sets of required constants for each type in the language is shown in Table 1;
- Delete the call. If the method returns a value that is used in an expression then the call is replaced with constants of the type of the returned value;
- Switch arguments of compatible types;
- Change method signature. Change the method call by removing one of the arguments or adding a new argument, changing the call to match the declaration of another method with the same name but with different signature, if such method exists;
- Insert arithmetic negation before an argument. Invert the sign of an argument by inserting an arithmetic negation, if allowed by the type of the argument. If the argument is an expression, only the whole expression is negated, not subexpressions;
- Insert logical negation before an argument;
- Insert bit negation before an argument;
- Increment and decrement an argument;

- Change target object. Use another type-compatible object to make the call. We can subdivide this mutation depending on which object is used to replace the original one. So we can have: 1) replace with local variable; 2) replace with formal parameter; 3) replace with static field; 4) replace with instance field.

Table 1: Set of constants for operator "Replace arguments with constants"

Type	Constants
int	0, 1, -1, Integer.MIN_VALUE, Integer.MAX_VALUE
short	0, 1, -1, Short.MIN_VALUE, Short.MAX_VALUE
byte	0, 1, -1, Byte.MIN_VALUE, Byte.MAX_VALUE
long	0, 1, -1, Long.MIN_VALUE, Long.MAX_VALUE
char	1, Character.MIN_VALUE, Character.MAX_VALUE
boolean	true, false
float	0.0, 1.0, -1.0, Float.MIN_VALUE, Float.MAX_VALUE
double	0.0, 1.0, -1.0, Double.MIN_VALUE, Double.MAX_VALUE
String	"", null
other objects	null

4.4 Other synchronization methods

Other methods that are not part of the "core" mechanisms for synchronization in Java have been defined in the Java API and are important for the implementation of concurrent programs. To these methods we can apply the same mutations defined in the last section, changing their behavior by mutating the calls to them.

The methods we identify in this class of mutations are: `Thread.interrupt`, `Thread.join`, `Thread.sleep`, `Thread.start` and `Thread.yeld`.

Note that some operators, when applied to specific structures in a program can generate the same mutation, or more precisely, mutants with the same behavior. This is the case for example of deleting a statement in a synchronized block and deleting a `wait` call. If the first is applied, the second does not have to be applied. This is not uncommon in the definition of mutant operators and allows one to use an incremental approach to choose the mutant operators.

Much of the mutant operator set can be parameterized according to the the implementor's or even the tester's will. For example: the value to add/subtract on timed `wait`; the set of constants to use in the replacements described in Section 4.3; the methods subject to mutation described in Section 4.4.

Table 2 summarizes the set of mutant operators.

5 Conclusion and Future Work

This paper discussed the use of mutation testing for concurrent Java programs. Mutation testing has been shown to be an effective way of testing software in terms of fault revealing capacity and it has been widely explored. One advantage of mutation testing is its flexibility in the sense that it can be applied to several scopes, e.g, unit and integration testing in different languages [17, 4], test of object oriented

Table 2: Proposed set of mutant operators

Operator name	Meaning
DelSync	Removes the synchronized attribute from a method declaration or removes a synchronized statement
DelStat	Deletes a statement in a synchronized method or block
MoveBrace	Moves `{` and `}` up and down
ReplSyncObject	Replaces synchronization object
DelWait	Deletes a call to the `wait` method
ReplWait	Replaces a call to a timed wait by a call to a non-timed `wait` and vice-versa
IncrDecrWait	Increment and decrement argument for timed `wait`
DelNotify	Deletes a call to `notify` or `notifyAll`
ReplNotify	Replaces a call to `notify` by a call to `notifyAll` and vice-versa.
ReplArg	Replaces argument with constant
DelSyncCall	Deletes a call to a synchronized method
SwitchArg	Switch arguments in a call to synchronized method
ReplMeth	Uses method with same name and other signature
InsNegArg	Inserts unary (negation) operators in an argument
ReplTargObj	Replaces the object in a call to synchronized method

programs [11] and test of formal specifications using different languages [7, 8, 6, 20]. To adapt the criterion to these different scenarios it is necessary to identify the characteristics one wants to exercise on that specific scope and to develop mutant operators for it.

This paper focuses on the definition of mutant operators to exercise the features of the Java language related to concurrency and synchronization. Mutant operators for other concurrent environments have been defined [17, 19], but due to the differences between those environments and the Java environment, new mutant operators are required to address concurrency in Java. A pilot study [2] has been conducted using well know operators, not specifically designed to this goal, and confirmed this necessity. The mutant operators herein presented tries to cover most of the concurrency- and synchronization-related features of the Java environment and still be cost effective, restricting the number of operators and mutants that they can generate.

The application of mutation testing to concurrent programs introduces additional problems. In particular it is necessary to deal with the intrinsic nondeterminism derived from concurrency. The execution of a program with a given input can produce different correct behaviors. This constitutes a problem in comparing the results of executing mutants with the results of the original program. A tool that supports mutation testing in this scenario must provide a way to assure that two executions of a program with the same test case lead to the same behavior. Currently we are working on developing the tools necessary to experiment the results described in this paper. The first tool applies the mutant operators described in this paper to automatically generate mutants. The second aims at recording executions of the original program to execute the same synchronization sequences on its mutants.

6 Acknowledgements

The authors would like to thank CNPq, CAPES and FAPESP for partially supporting this project.

References

[1] H. Agrawal, R. A. DeMillo, R. Hataway, Wm. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of Mutant Operators for C Programming Language. Tech Report SERC-TR41-P, Software Engineering Research Center, Purdue University, March 1989.

[2] M. Delamaro, M. Pezzè, A. M. R. Vincenzi, and J. C. Maldonado. Applying Mutation Testing to Multi-threaded JAVA Programs. Tech Report, *www.din.uem.br/˜delamaro/papers/relat.ps.gz*, 2001.

[3] M. E. Delamaro. *Proteum: Um Ambiente de Teste Baseado Na Análise de Mutantes.* Master thesis, SCE-ICMSC-USP, São Carlos - SP, October 1993.

[4] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface Mutation: An Approach for Integration Testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.

[5] R. A. DeMillo and A. J. Offutt. Constraint Based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[6] S. C. P F. Fabbri. *A Análise de Mutantes No Contexto de Sistemas Reativos: Uma Contribuição Para O Estabelecimento de Estratégias de Teste e Validação.* Doctoral dissertation, IFSC - USP, São Carlos - SP, October 1996.

[7] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Mutation Analysis Testing for Finite State Machines. In *Proceedings of the 5th International Symposium on Software Reliability Engineering (ISSRE)*, pages 220–229, Monterey - CA, November 1994.

[8] S. C. P. F. Fabbri, J. C. Maldonado, P. C. Masiero, and M. E. Delamaro. Mutation Analisys Applied to Validate Specifications Based on Petri Nets. In *Proceeding of the 8th IFIP Conference on Formal Descriptions Techniques for Distribute Systems and Communication Protocols*, pages 329–337, Montreal, October 1995.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Tha Java Language Specification.* Java Series. Addison-Wesley, 2nd edition, June 2000.

[10] S. Kim, J. A. Clark, and J. McDermid. The Rigorous Generation of Java Mutation Operators Using HAZOP. In *Proceedings of the 12th International Conference on Software & Systems Engineering and their Applications (ICSSEA'99)*, December 1999.

[11] S. Kim, J. A. Clark, and J. McDermid. Class Mutation: Mutation Testing for Object-Oriented Programs. In *Proceedings of the FMES*, October 2000.

[12] K. N. King and A. J. Offutt. A Fortran Language System for Mutation Based Software Testing. *Software-Practice and Experience*, 21(7):685–718, July 1991.

[13] A. P. Mathur and W. E. Wong. Evaluation of the Cost of Alternate Mutation Strategies. In *Proceedings of the 7th Brazilian Symposium on Software Engineering*, pages 320–335, Rio de Janeiro, RJ, Brazil, October 1993.

[14] A. J. Offutt. Coupling Effect: Fact or Fiction. In *Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification (ISSTA'89)*, pages 131–140, Key West, FL, December 1989.

[15] A. J. Offutt and J. H. Hayes. A Semantic Model of Program Faults. In *Proceedings of the International Symposium on Software Testing and Analisys and Verification (ISSTA'96)*, San Diego, CA, 1996.

[16] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, 1996.

[17] A. J. Offutt, J.M. Voas, and J. Payne. Mutation Operators for ADA. Technical Report ISSE-TR-96-09, Department of ISSE, George Mason University, Fairfax, VA, March 1996.

[18] R. L. Probert and F. Guo. Mutation Testing of Protocols: Principles and Preliminary Experimental Results. In *IFIP TC6 – Third International Workshop on Protocol Test Systems*, pages 57–76. North-Holland, 1991.

[19] S. Silva-Barradas. *Mutation Analysis of Concurrent Software.* Phd thesis, Department of Eletronic and Informatics, Polythecnic of Milan, Milan, Italy, 1997.

[20] S. R. S. Souza, J. C. Maldonado, S. C. P. F. Fabbri, and W. Lopes de Souza. Mutation Testing Applied to Estelle Specifications. In *33rd Hawaii Internacional Conference on System Sciences, Mini-Tracks: Distributed Systems Testing*, Maui, Hawai, January 2000. (Accepted for publication in a special issue on Distributed Systems Testing of the Software Quality Journal).

[21] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa, and M. E. Delamaro. Unit and Integration Testing for C Programs Using Mutation-based Criteria. *Journal of Software Testing, Validation and Reliability*, 2001. to appear.