MAJOR: An Efficient Technique for Mutation Analysis in a Java Compiler

René Just[1] and Gregory M. Kapfhammer[2]

[1]Department of Applied Information Processing, Ulm University
[2]Department of Computer Science, Allegheny College

Important Contributions

- Enhanced the Java 6 Standard Edition compiler
- Simple compiler options enable the mutation analysis
- Easily applicable in all Java development environments
- Effectively reduces mutant generation time to a minimum

Conditional Mutation

- Transforms the program's abstract syntax tree (AST)
- Encapsulates the mutations within conditional statements

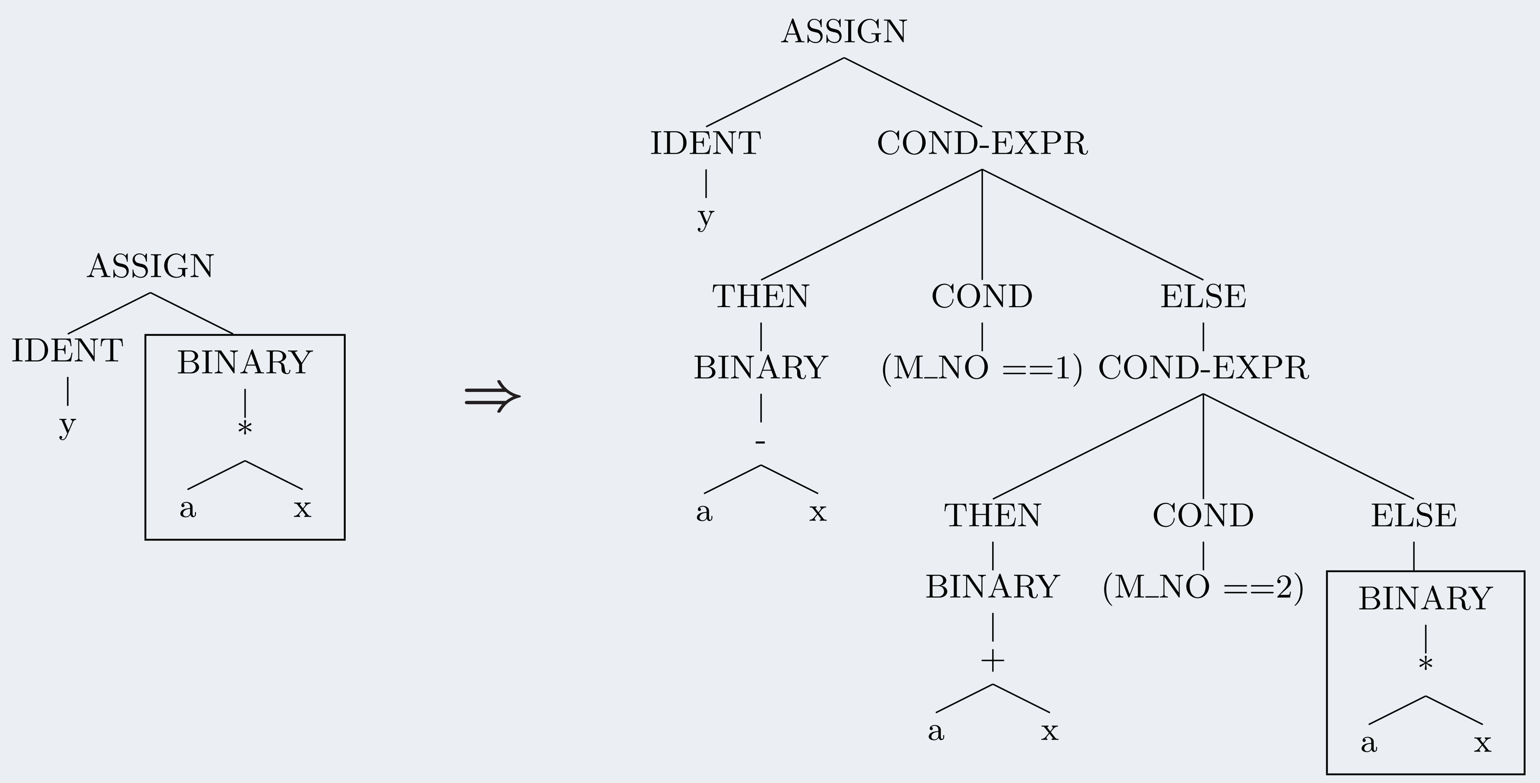

Figure: Multiple mutated binary expression as the right hand side of an assignment statement.

Supported Features

- Enables second and higher order mutation analysis
- Determination of mutation coverage by running the original code
- Configurable mutation operators by means of compiler options

Mutation Coverage

```
public int eval(int x){
    int a = 3, b = 1, y;

    y = (M_NO==1)? a - x:
        (M_NO==2)? a + x:
        (M_NO==3)? a % x:
        (M_NO==0 && COVER(1,3))?
       a * x : a * x; // original

    if(M_NO==4){
      y -= b;
    }else if(M_NO==0 && COVER(4,4)){
       y += b;
    }else{
       y += b; // original
    }

    return y;
}
```

Figure: Collecting coverage information.

- It is impossible to kill a mutant if it is not reached and executed
- Additional instrumentation determines the covered mutations
- Mutation coverage is only examined if the tests execute the original code
- An external driver efficiently records the covered mutations as ranges
- Only those mutants covered by a test case are executed

Performance Evaluation

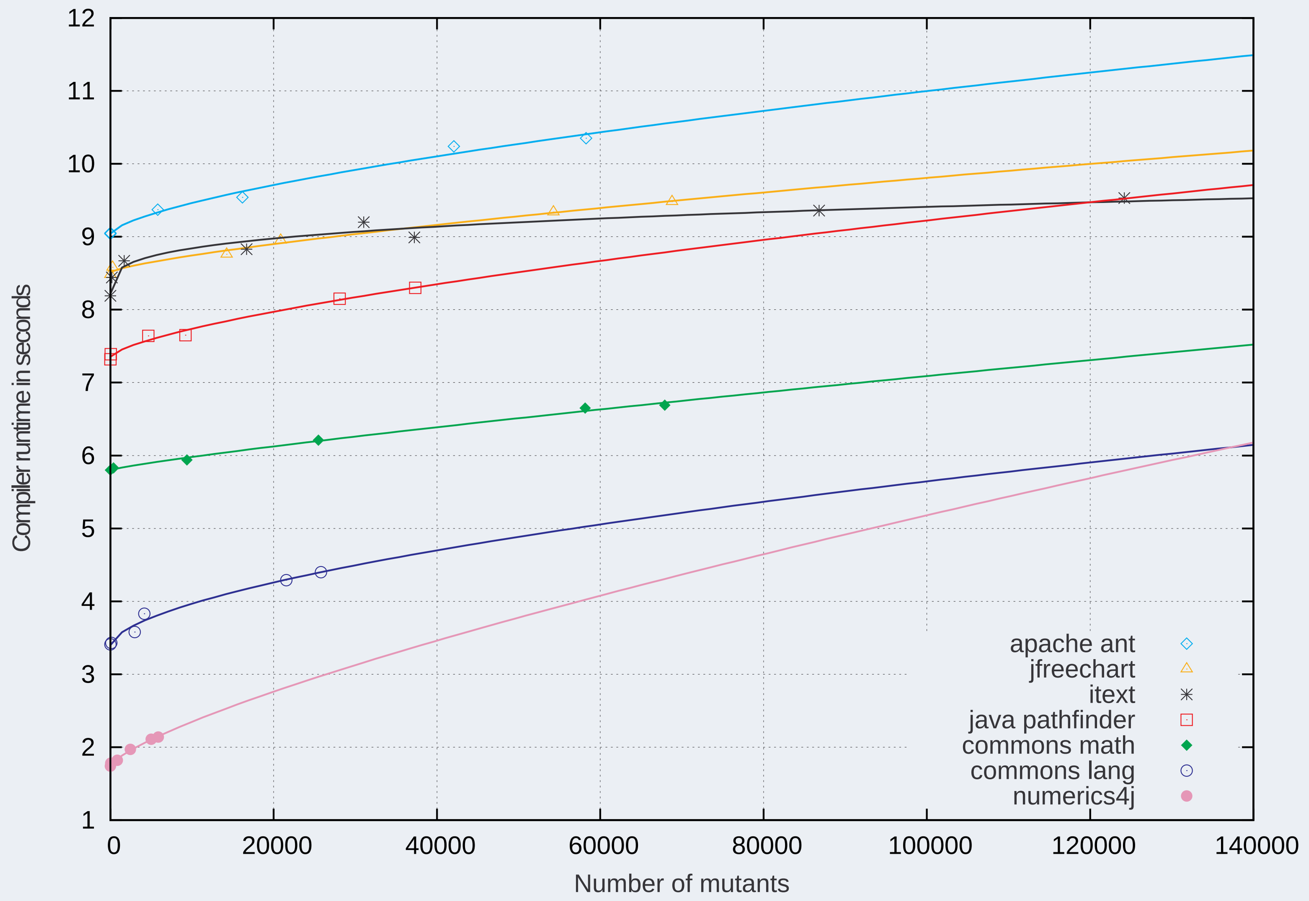

Figure: Compiler runtime to generate and compile the mutants for all of the projects.

Table: Time and space overhead for all of the investigated projects.

Application	*Mutants* generated	*Mutants* covered	*Mutants* killed	*Runtime of test suite*[1] original	*Runtime of test suite*[1] instrumented wcs[2]	*Runtime of test suite*[1] instrumented wcs+cov[3]	*Memory consumption*[1] original	*Memory consumption*[1] instrumented
aspectj	406,382	20,144	10,361	4.3	4.8	5.0	559	813
apache ant	60,258	28,118	21,084	331.0	335.0	346.0	237	293
jfreechart	68,782	29,485	12,788	15.0	18.0	23.0	220	303
itext	124,184	12,793	4,546	5.1	5.6	6.3	217	325
java pathfinder	37,331	8,918	4,434	17.0	22.0	29.0	182	217
commons math	67,895	54,326	44,084	67.0	83.0	98.0	153	225
commons lang	25,783	21,144	16,153	10.3	11.8	14.8	104	149
numerics4j	5,869	4,900	401	1.2	1.3	1.6	73	90

[1]Runtime in seconds and memory consumption of the compiler in megabytes [2]wcs: worst-case scenario [3]cov: coverage tracking enabled

- Time overhead for generating and compiling the mutants is negligible
- Inserting conditional statements leads to a minimal increase in space overhead
- Even for large projects, the method is applicable on commodity workstations

Implementation Details

- A separate package modularly extends the compiler
- Mutation operators configurable with enhanced `-X` options
- AST transformation implemented by means of the visitor pattern

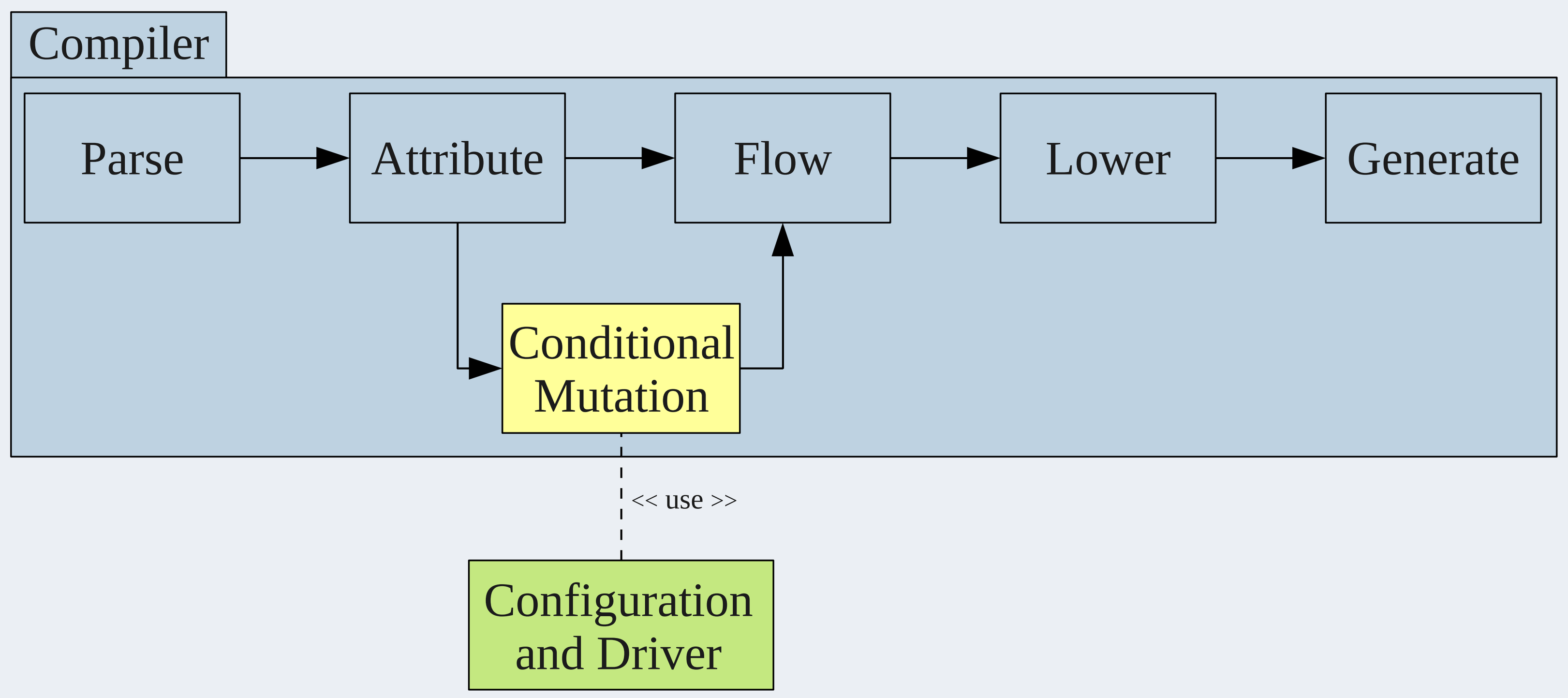

Figure: Integration of the conditional mutation approach into the compilation process.

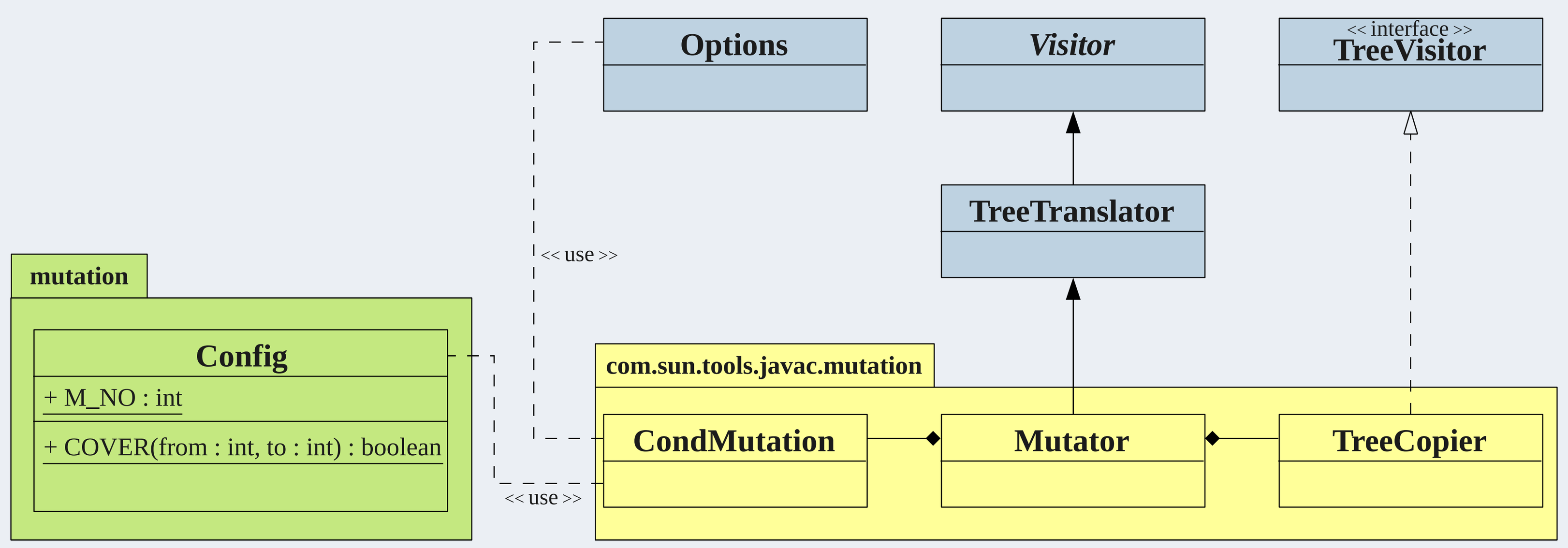

Figure: UML diagram of the implemented compiler classes and the external driver class.

Optimized Workflow

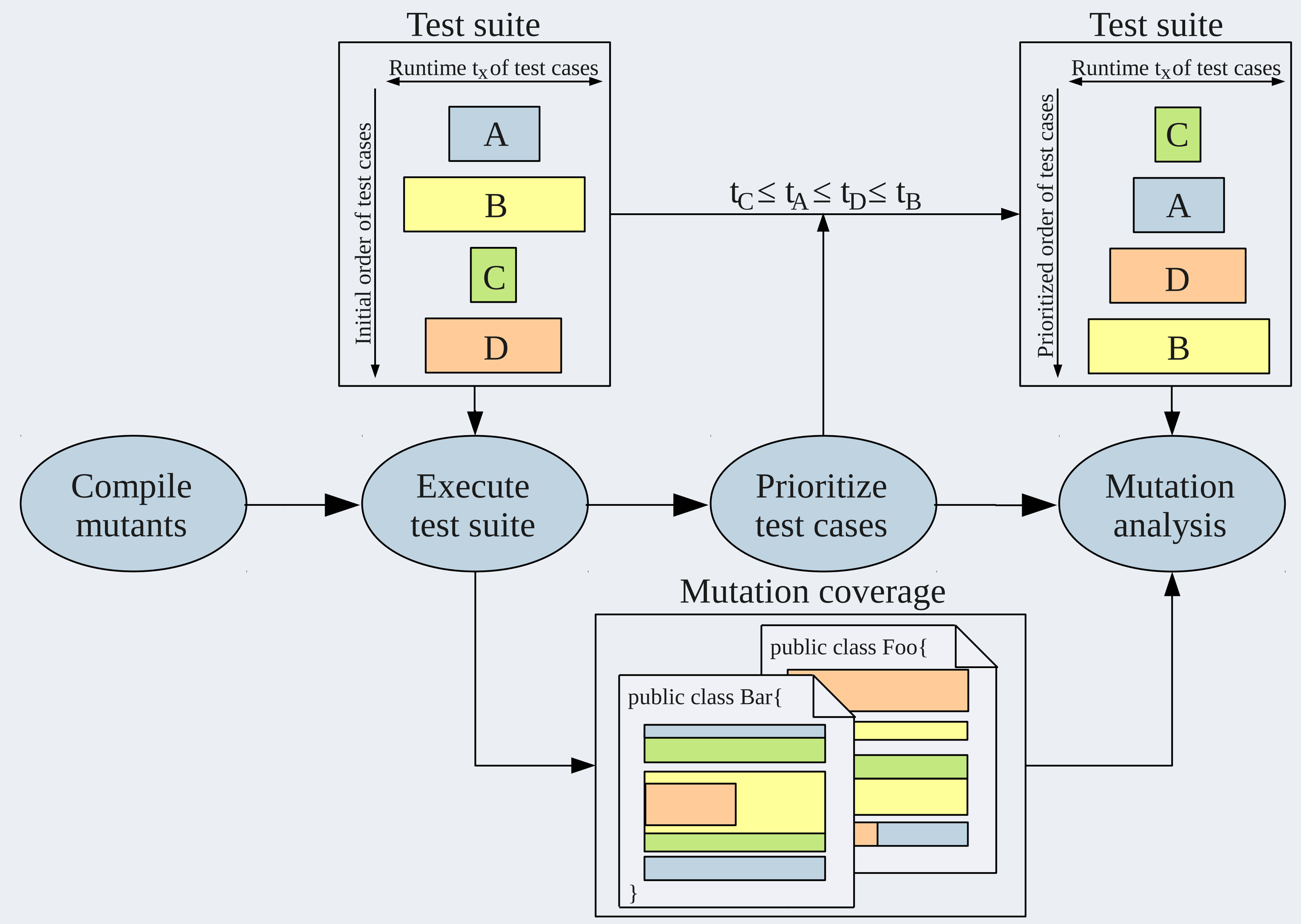

Figure: Minimizing the runtime of mutation analysis by means of test prioritization and mutation coverage.

Future Work

- Comparison of MAJOR with related techniques and tools such as muJava, Javalanche, and Jumble
- Further runtime optimizations by balancing the AST
- Implementation of several new mutation operators
- Domain specific language for specifying mutation operators
- Integration of conditional mutation into a C/C++ compiler