MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler

René Just[1], Franz Schweiggert[1], and Gregory M. Kapfhammer[2]

[1]Ulm University, Germany

[2]Allegheny College, USA

26th International Conference on Automated Software Engineering

Lawrence, Kansas, USA

November 6 - 12, 2011

ulm university universität ulm

ALLEGHENY COLLEGE

Overview of MAJOR

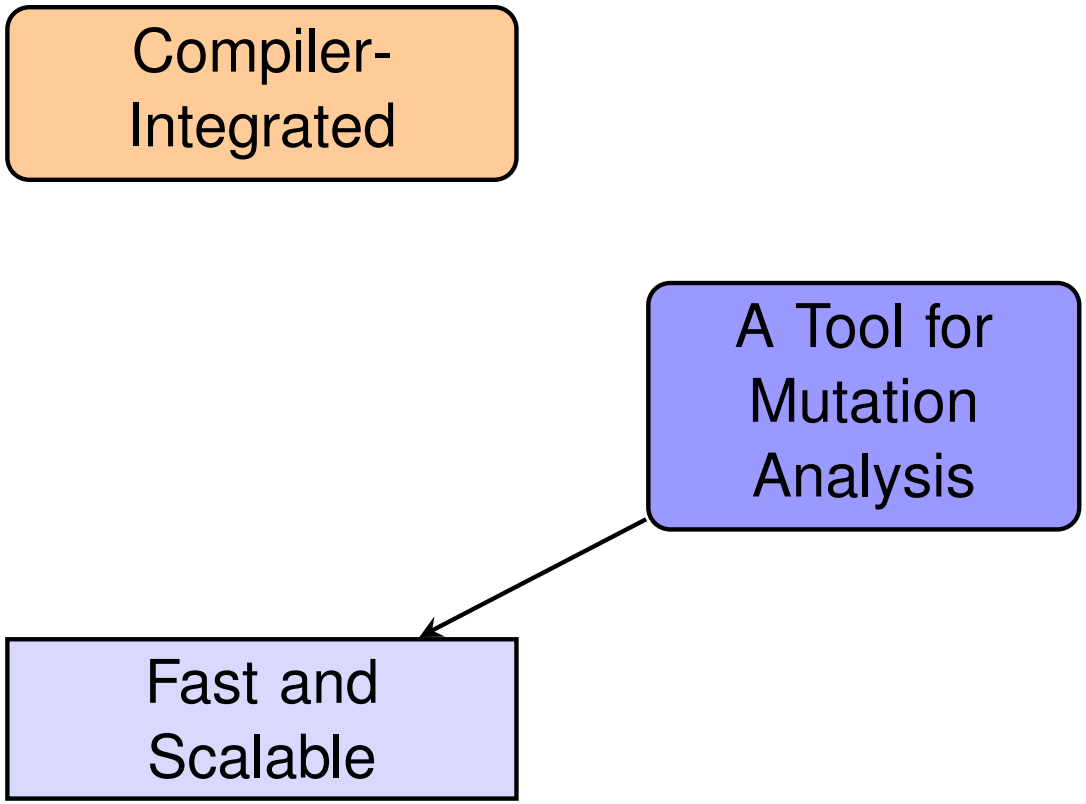
A Tool for Mutation Analysis

Overview of MAJOR

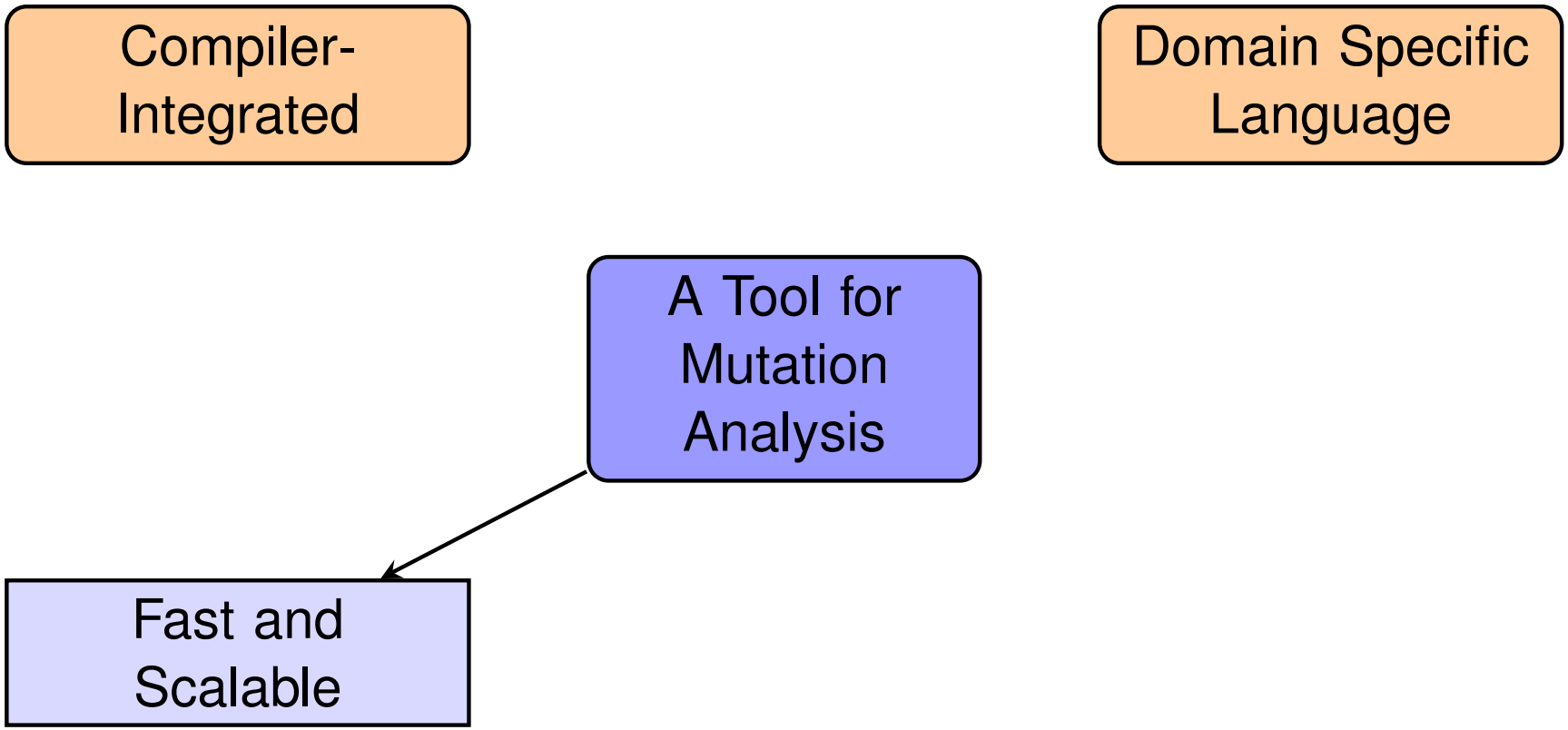
Compiler-Integrated

A Tool for Mutation Analysis

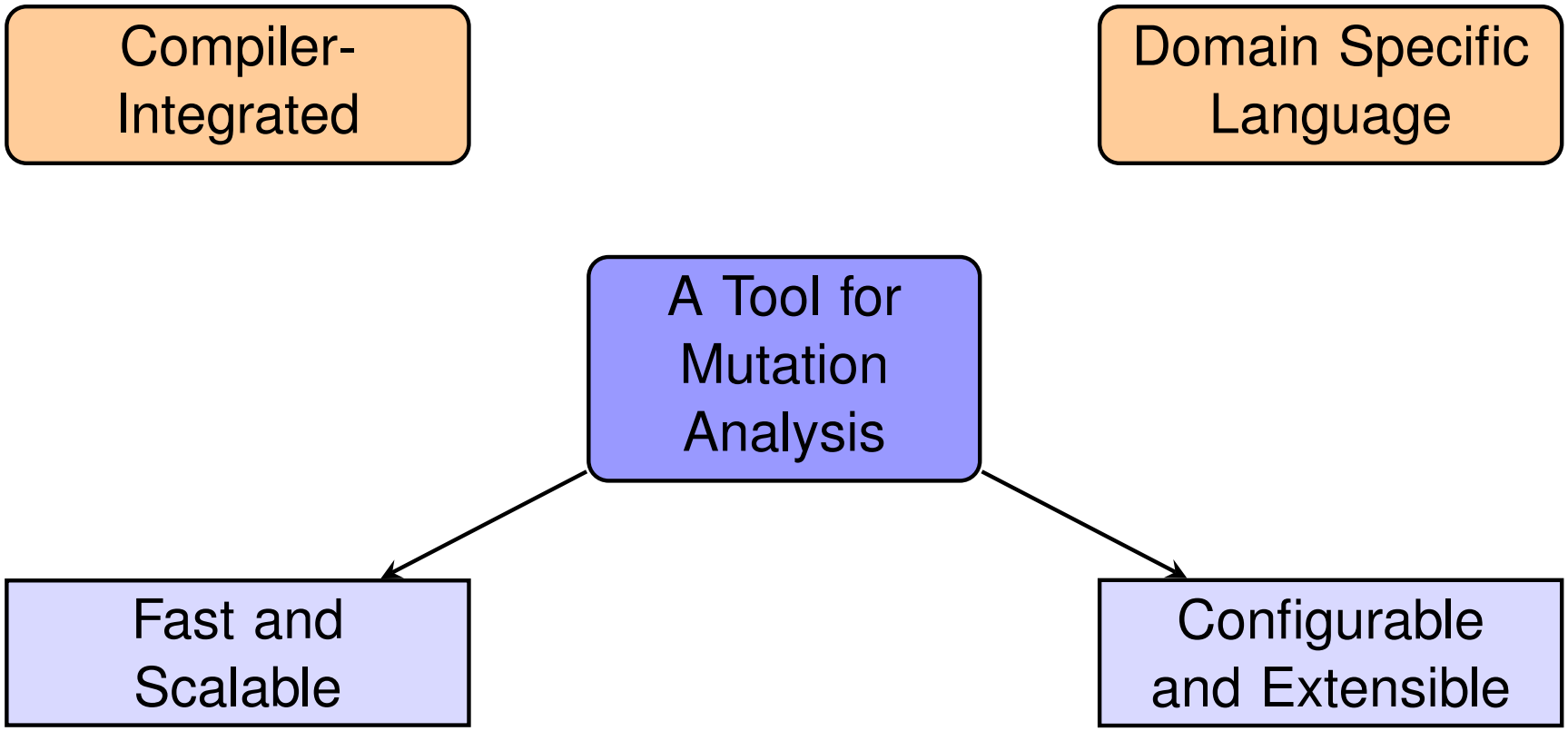
Overview of MAJOR



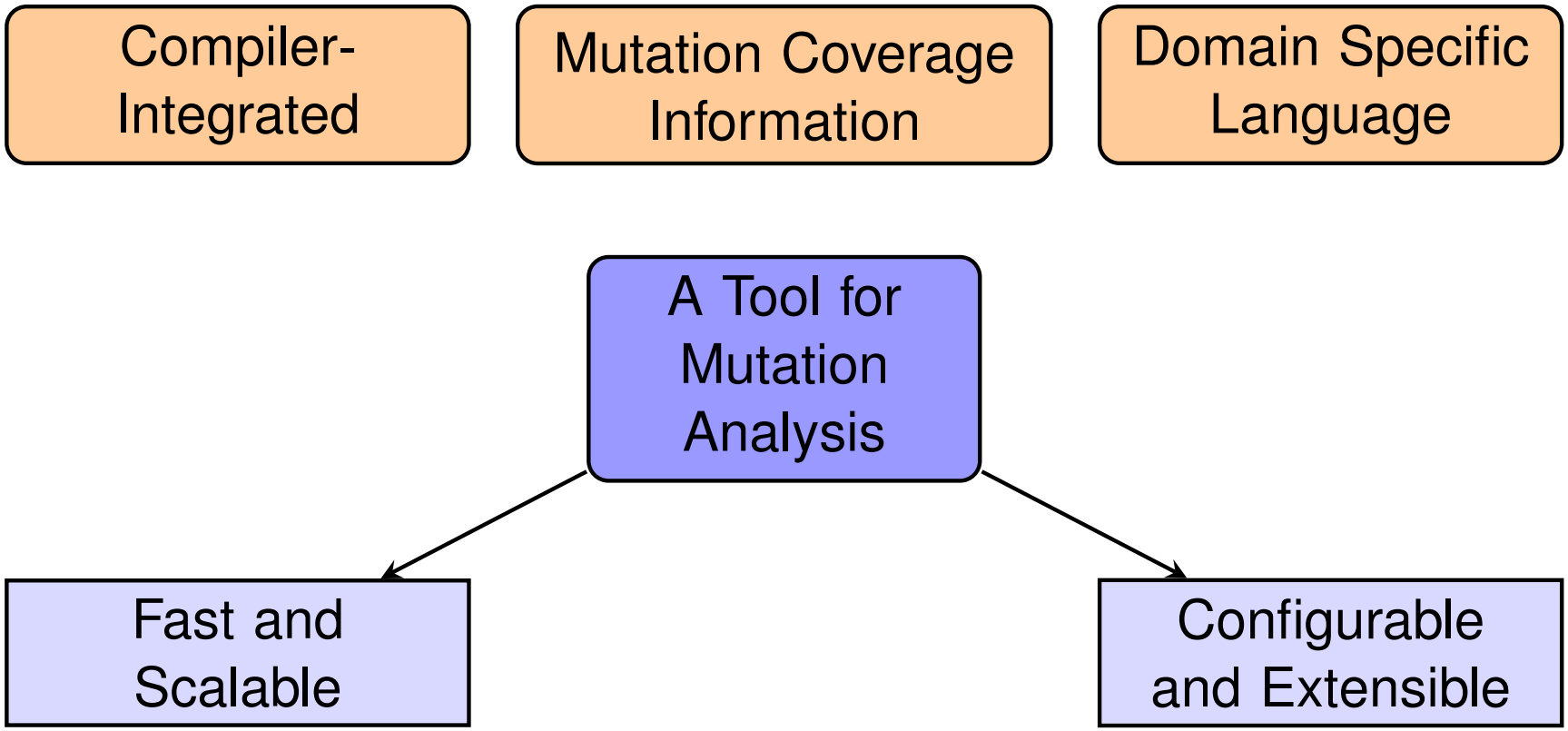
Overview of MAJOR



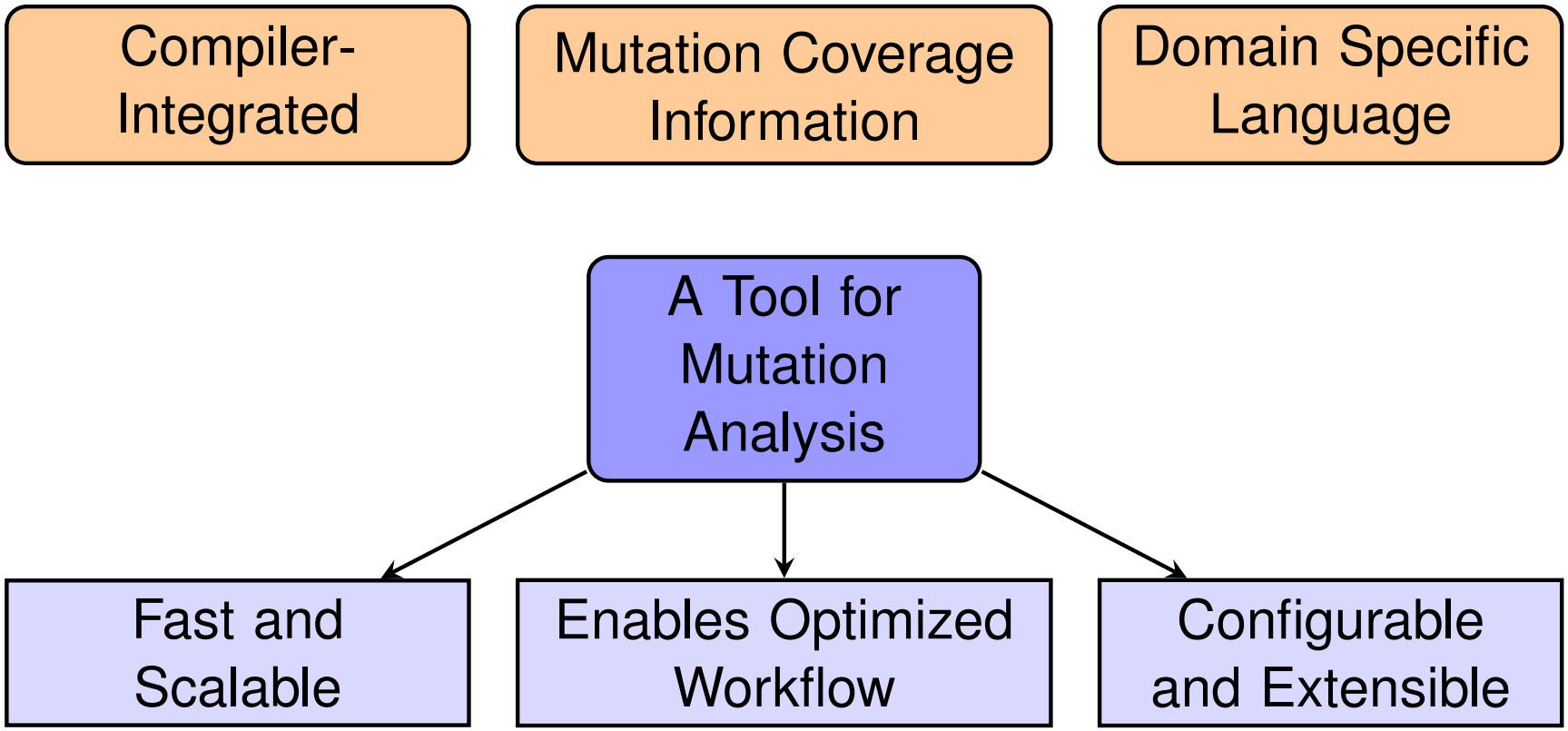
Overview of MAJOR



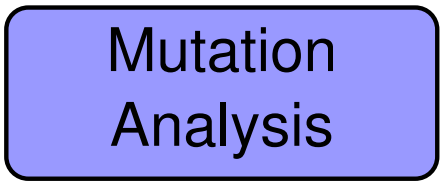
Overview of MAJOR



Overview of MAJOR



Overview of Mutation Analysis

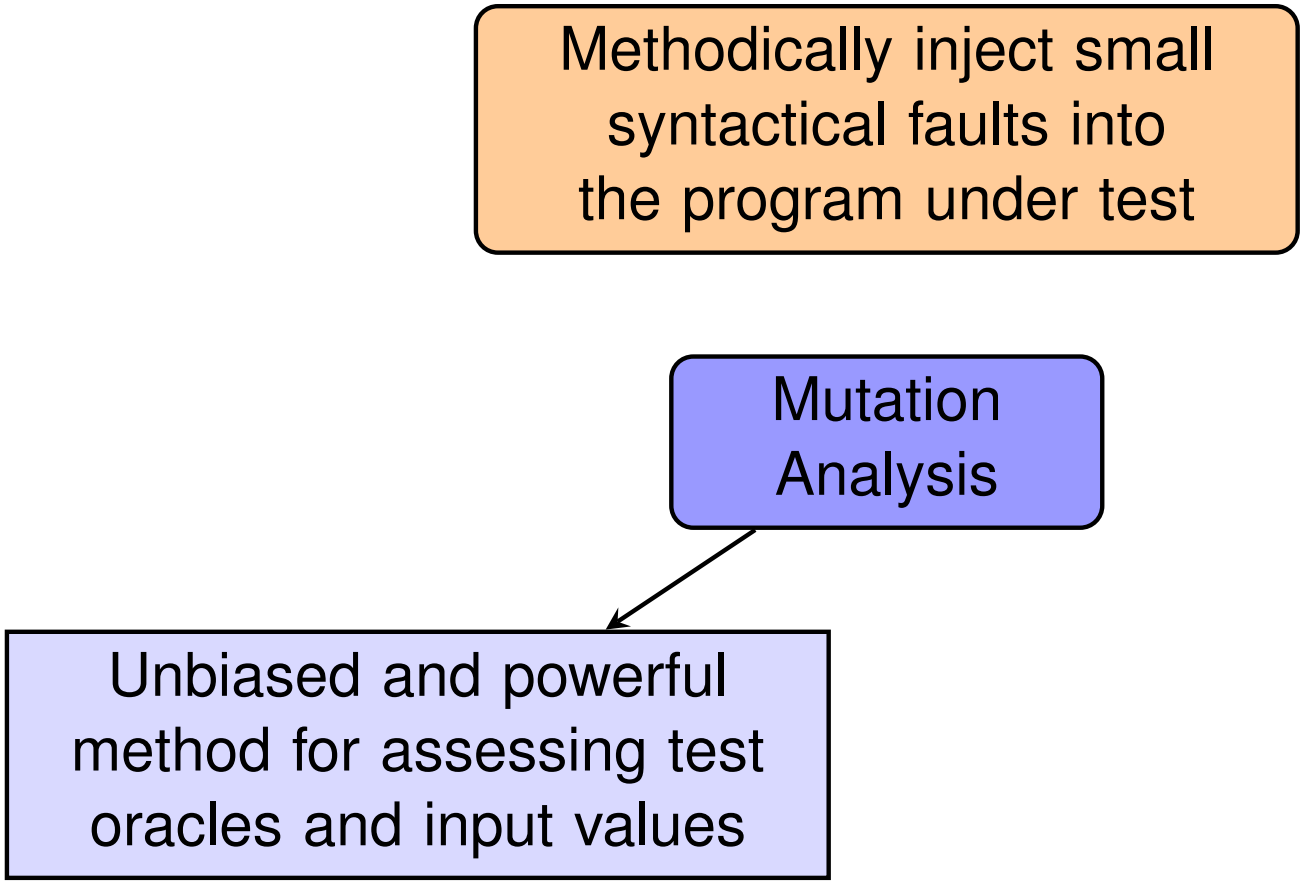


Overview of Mutation Analysis

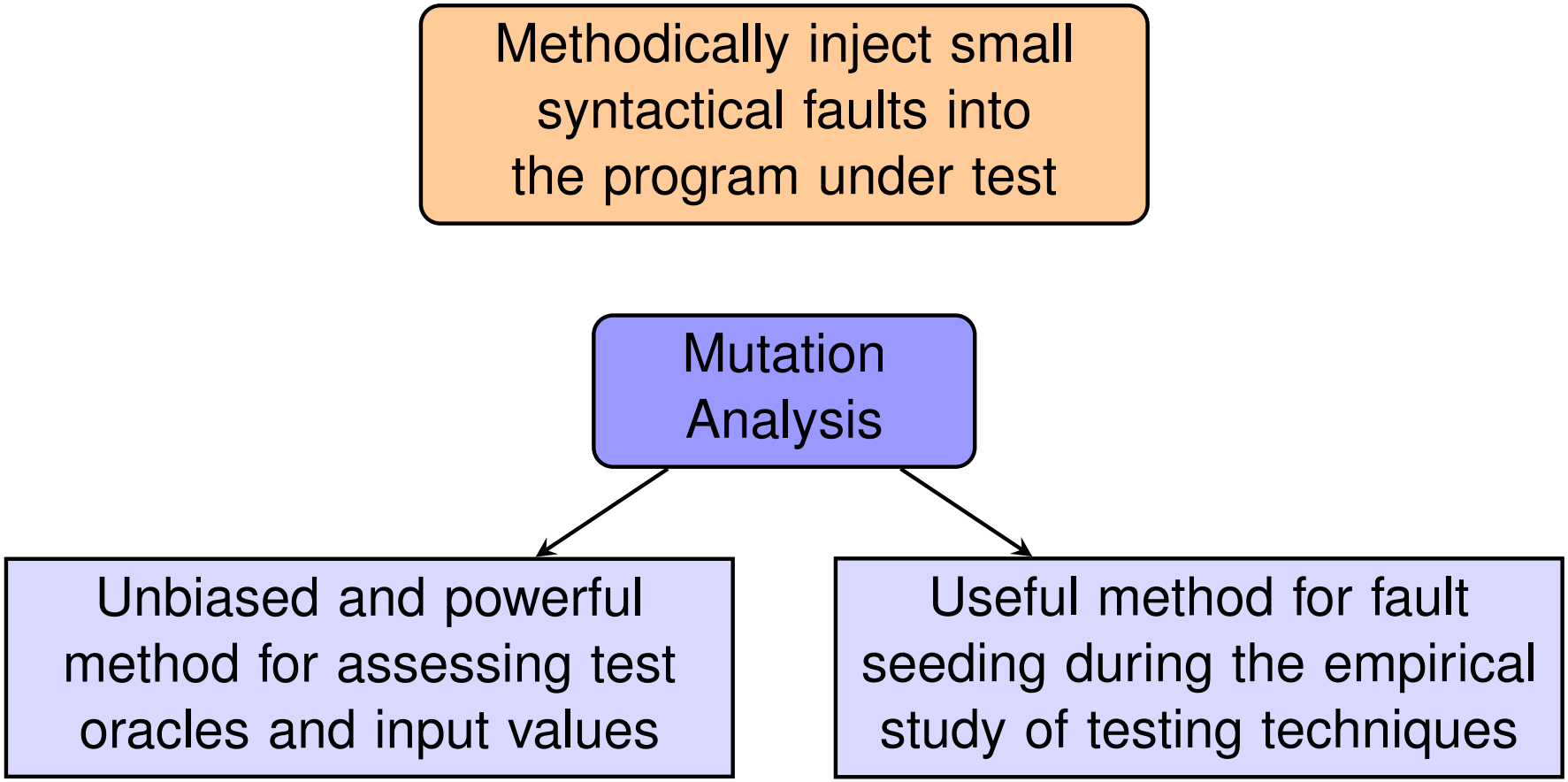
Methodically inject small syntactical faults into the program under test

Mutation Analysis

Overview of Mutation Analysis



Overview of Mutation Analysis



Overview of Mutation Analysis

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

    y = a * x;

    y += b;
    return y;
}

public int max(int a, int b){
   int max = a;

   if(b>a){
      max=b;
   }

   return max;
}
```

Overview of Mutation Analysis

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

    y = a * x;

    y += b;
    return y;
}

public int max(int a, int b){
   int max = a;

   if(b>a){
      max=b;
   }

   return max;
}
```

Overview of Mutation Analysis

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

    y = a * x;

    y += b;
    return y;
}

public int max(int a, int b){
   int max = a;

   if(b>a){
      max=b;
   }

   return max;
}
```

$\Longrightarrow$

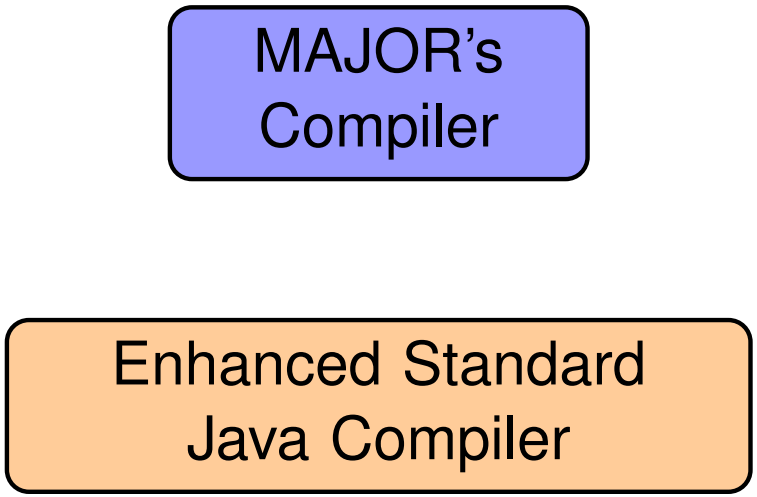
- `y = a - x;`
- `y = a + x;`
- `y = a / x;`

$\Longrightarrow$

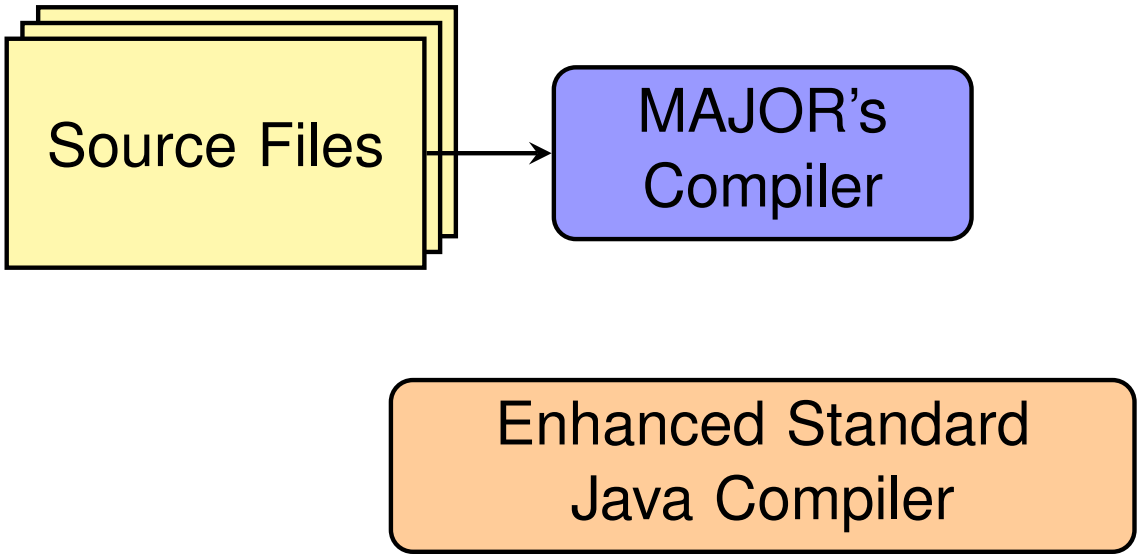
- `if(b < a)`
- `if(b != a)`
- `if(b == a)`

MAJOR's Compiler

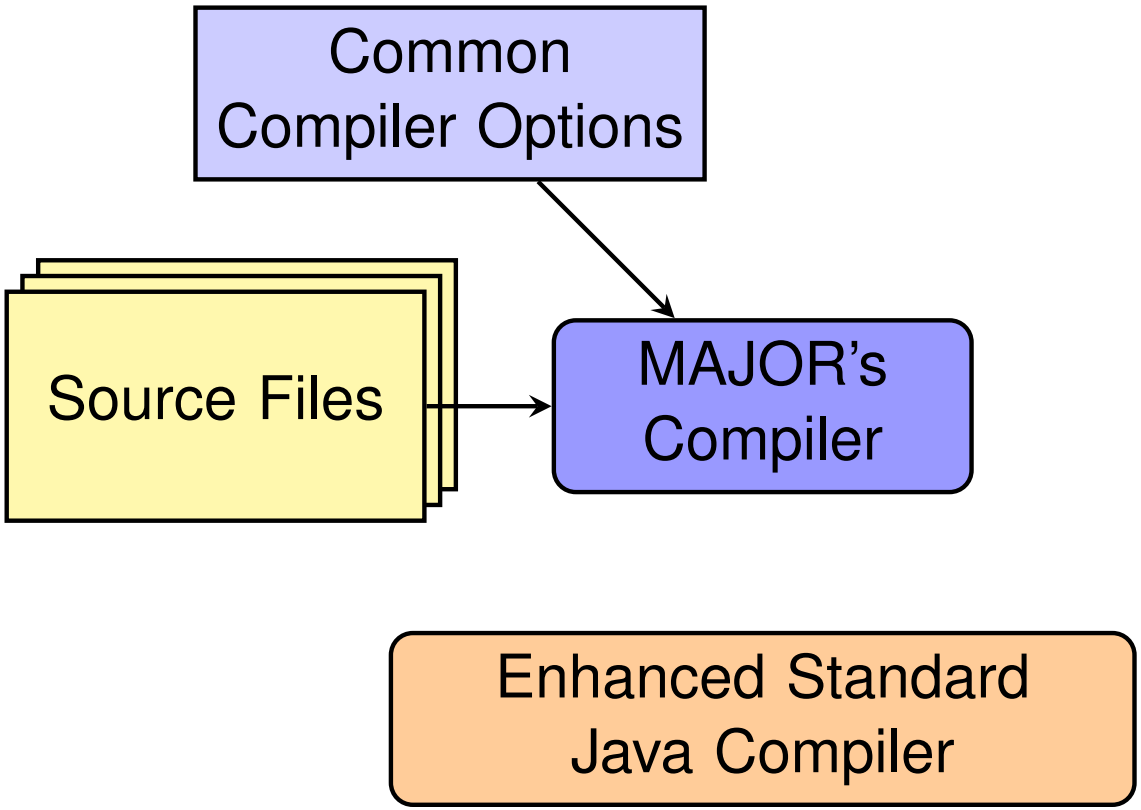
MAJOR's Compiler



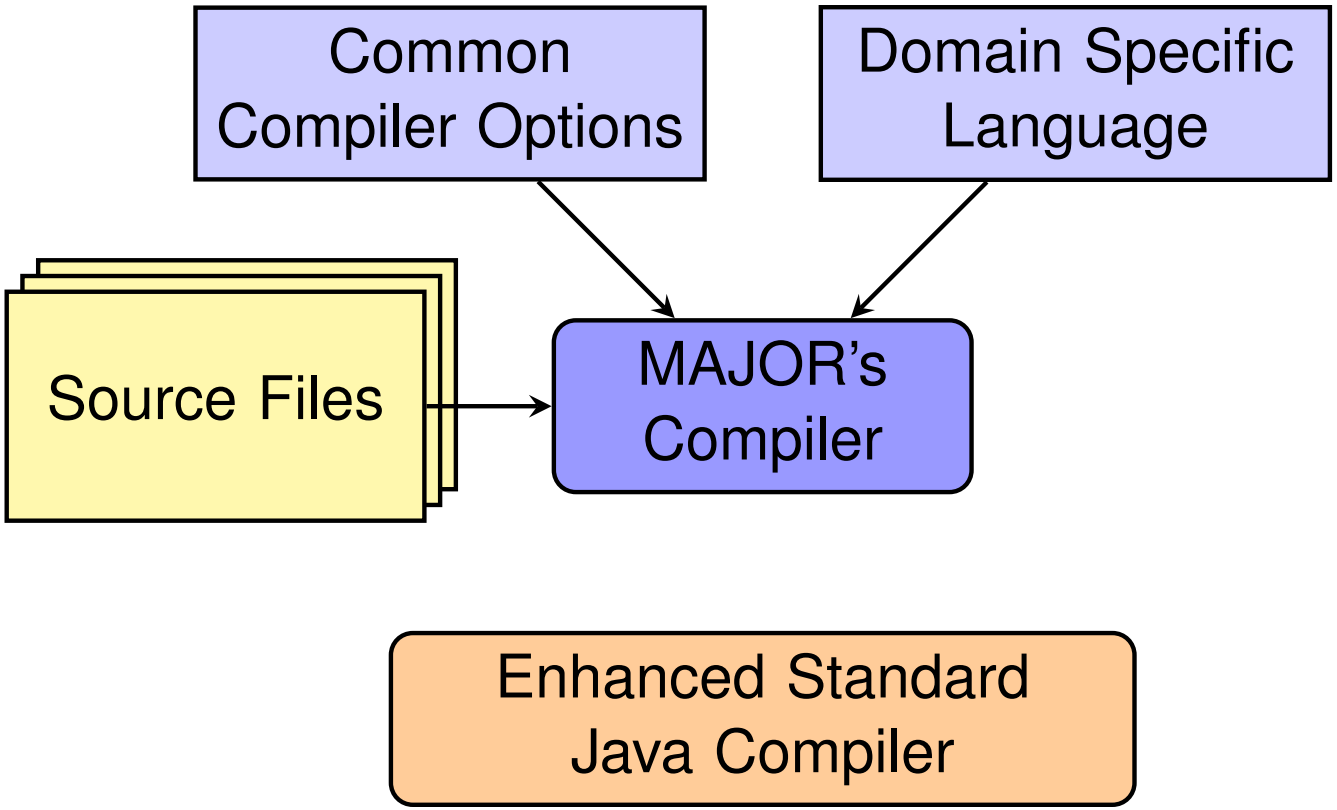
MAJOR's Compiler



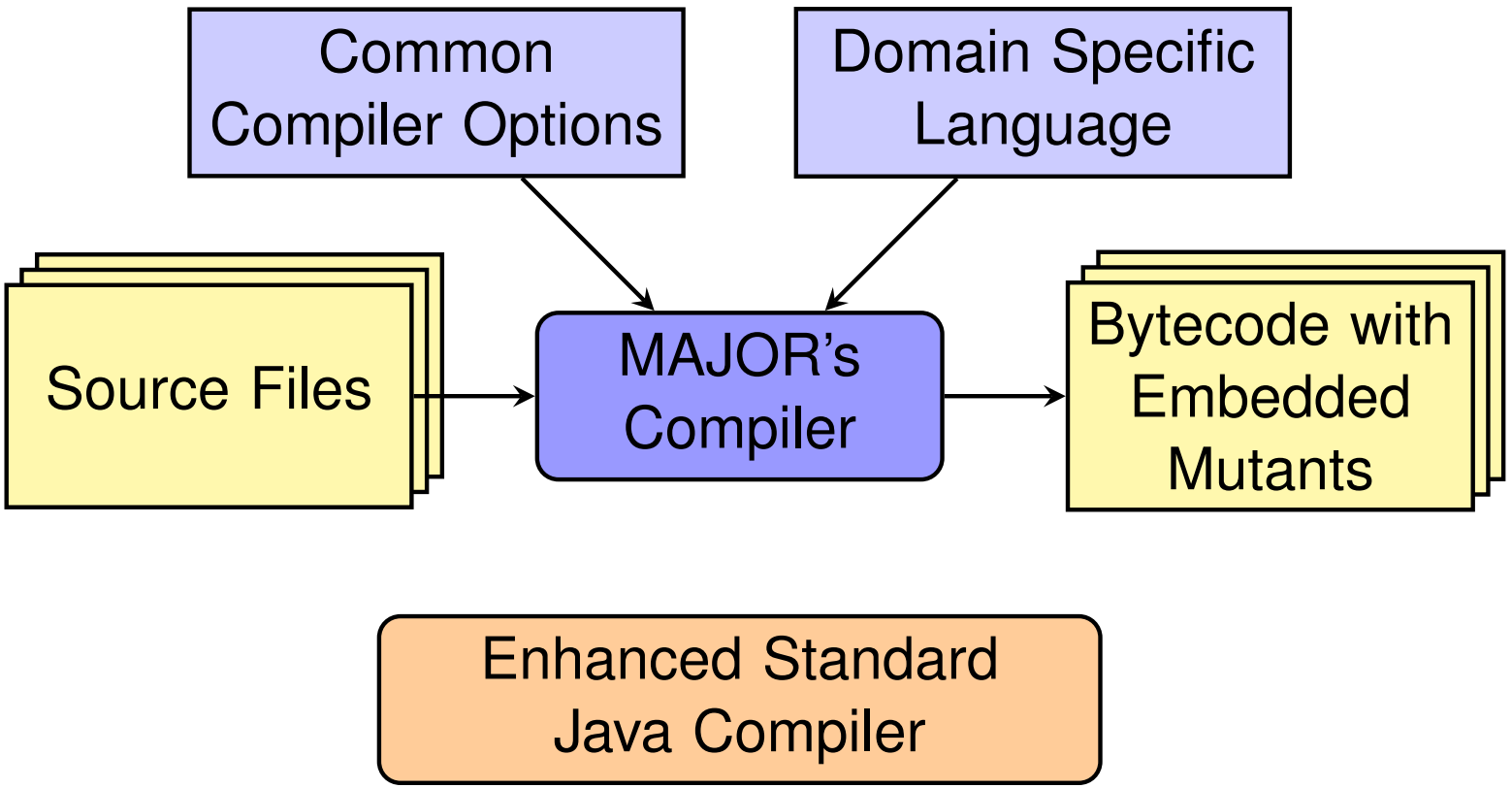
MAJOR's Compiler



MAJOR's Compiler



MAJOR's Compiler



MAJOR's Domain Specific Language

```
// variable declaration
listCOR={&&, ||, ==, !=};
// Define replacement list
BIN(+)<"org"> -> {-,*};
BIN(*)<"org"> -> {/,%};
// Define own operator
myOp{
  BIN(&&) -> listCOR;
  BIN(||) -> listCOR;
  COR;
  LVR;
}
// Enable built-in operator AOR
AOR<"org">;
// Enable operator myOp
myOp<"java.lang.System@println">;
```

MAJOR's Domain Specific Language

```
// variable declaration
listCOR={&&, ||, ==, !=};
// Define replacement list
BIN(+)<"org"> -> {-,*};
BIN(*)<"org"> -> {/,%};
// Define own operator
myOp{
  BIN(&&) -> listCOR;
  BIN(||) -> listCOR;
  COR;
  LVR;
}
// Enable built-in operator AOR
AOR<"org">;
// Enable operator myOp
myOp<"java.lang.System@println">;
```

Specify mutation operators in detail

MAJOR's Domain Specific Language

```
// variable declaration
listCOR={&&, ||, ==, !=};
// Define replacement list
BIN(+)<"org"> -> {-,*};
BIN(*)<"org"> -> {/,%};
// Define own operator
myOp{
  BIN(&&) -> listCOR;
  BIN(||) -> listCOR;
  COR;
  LVR;
}
// Enable built-in operator AOR
AOR<"org">;
// Enable operator myOp
myOp<"java.lang.System@println">;
```

Specify mutation operators in detail

Define own mutation operator groups

MAJOR's Domain Specific Language

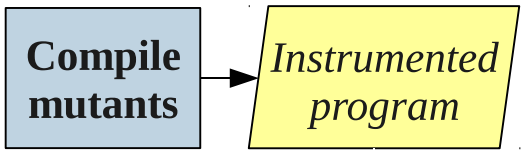
```
// variable declaration
listCOR={&&, ||, ==, !=};
// Define replacement list
BIN(+)<"org"> -> {-,*};
BIN(*)<"org"> -> {/,%};
// Define own operator
myOp{
   BIN(&&) -> listCOR;
   BIN(||) -> listCOR;
   COR;
   LVR;
}
// Enable built-in operator AOR
AOR<"org">;
// Enable operator myOp
myOp<"java.lang.System@println">;
```

Specify mutation operators in detail

Define own mutation operator groups

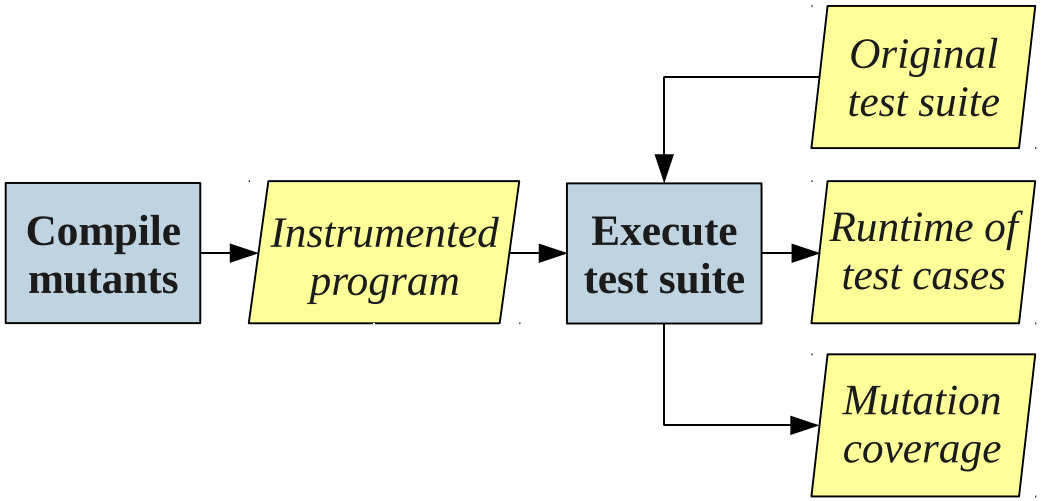
Enable operators for a specific package, class, or method

Optimized Mutation Analysis Process



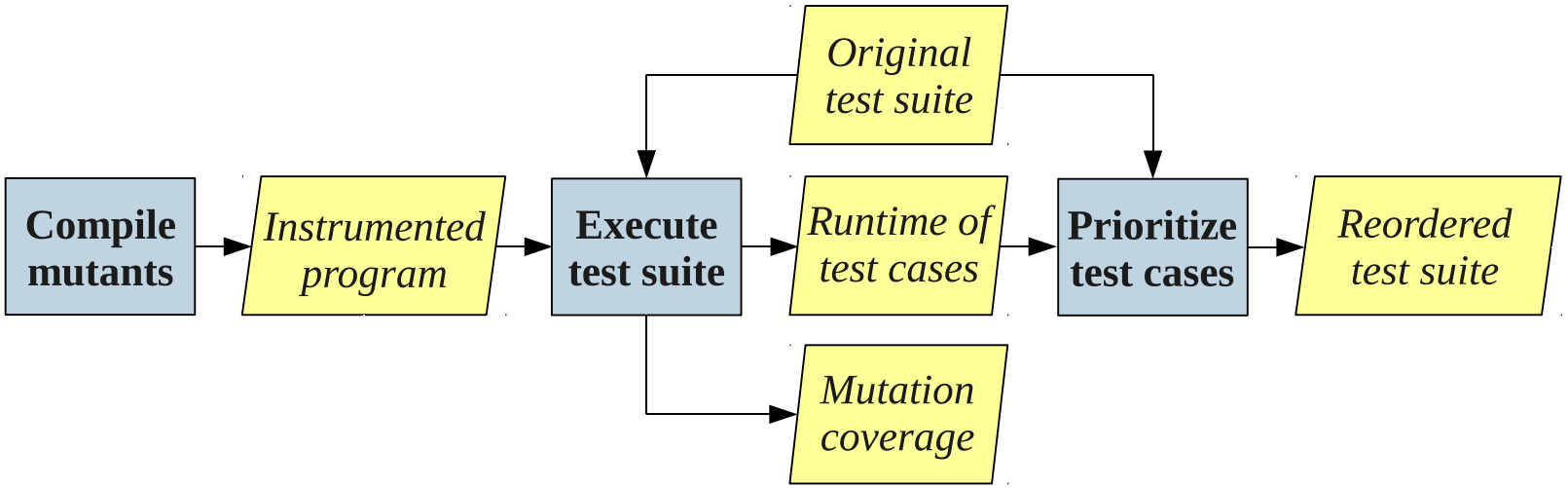
1. Embed and compile all mutants
2. Run test suite on instrumented program
3. Sort tests according to their runtime
4. Perform mutation analysis with reordered test suite

Optimized Mutation Analysis Process



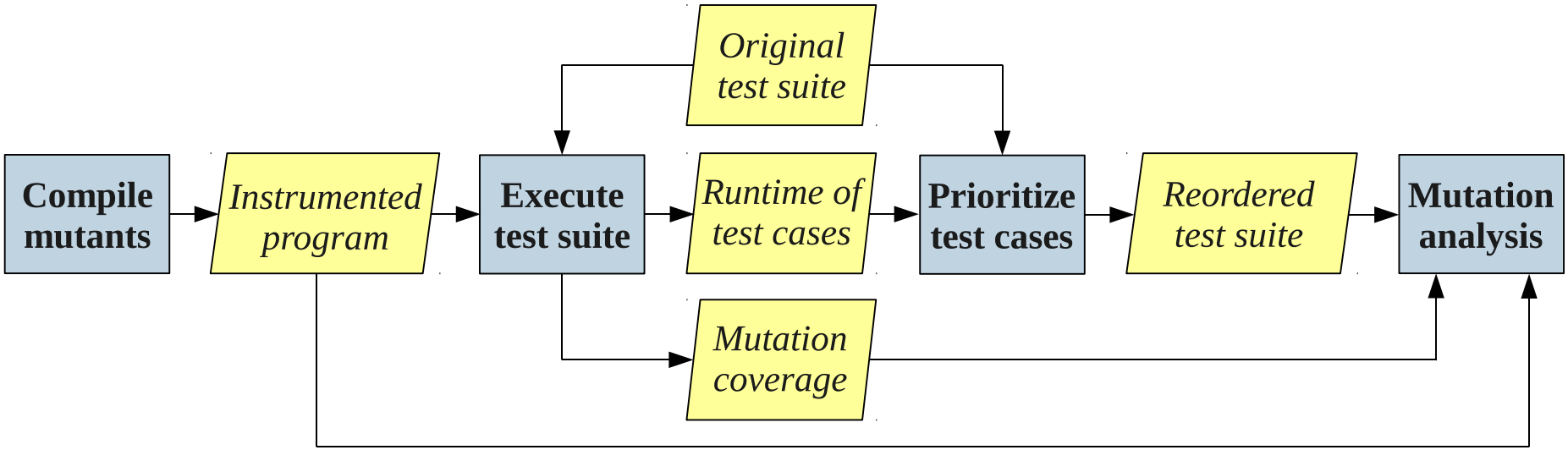
1. Embed and compile all mutants
2. Run test suite on instrumented program
3. Sort tests according to their runtime
4. Perform mutation analysis with reordered test suite

Optimized Mutation Analysis Process



1. Embed and compile all mutants
2. Run test suite on instrumented program
3. Sort tests according to their runtime
4. Perform mutation analysis with reordered test suite

Optimized Mutation Analysis Process



1. Embed and compile all mutants
2. Run test suite on instrumented program
3. Sort tests according to their runtime
4. Perform mutation analysis with reordered test suite

Conclusion

Key Concepts and Features:

- Compiler-integrated solution
- Furnishes its own domain specific language
- Provides mutation coverage information

Conclusion

Key Concepts and Features:

- Compiler-integrated solution
- Furnishes its own domain specific language
- Provides mutation coverage information

Characteristics of MAJOR:

- Fast and scalable technique
- Configurable and extensible mutation tool
- Enables an optimized workflow for mutation analysis

Do you want to learn more details about MAJOR?

MAJOR: An Efficient and Extensible Tool for Mutation Analysis in a Java Compiler

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

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

Important Contributions

- Enhances the Java 6 Standard Edition compiler
- Provides its own domain specific language (DSL)
- 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

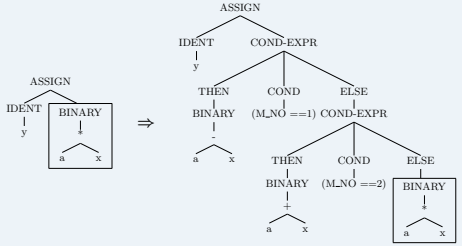


Figure: Multiple mutated binary expression as the right hand side of an assignment statement.

Supported Features

- Simple compiler options enable the mutation analysis
- Configurable mutation operators by means of a DSL
- Determination of mutation coverage by running the original code

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 && COVERED(1,3)) ?
       a * x : a * x; // original

   if(M_NO==4){
      y -= b;
   }else if(M_NO==0 && COVERED(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

Implementation Details

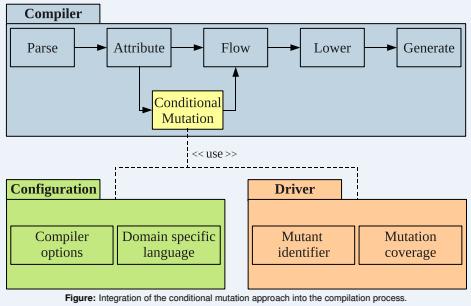


Figure: Integration of the conditional mutation approach into the compilation process.

```
listCOR={&&, ||, ==, !=};
method ="java.lang.System@println";

myOp{
  BIN(&&) -> listCOR;
  BIN(||) -> listCOR;
  COR;
  LVR;
}

// Define replacement list
BIN(+)<"org"> -> (-,*);
BIN(*)<"org"> -> (/,%);
// Enable built-in operator AOR
AOR<"org">;
// Enable operator myOp
myOp<method>;
```

Figure: DSL script to define the mutation process.

```
package major.mutation;

public class Driver{
   public static int MAX_NO = 100000;
   public static int[] COV  =
          new int[MAX_NO];

   public static int M_NO=0;

   public static boolean COVERED
   (int from, int to){
       for(int i=from; i<=to; ++i){
           COV[i]++;
       }
       return false;
   }
}
```

Figure: Simple driver class implementation.

Optimized Mutation Analysis Process

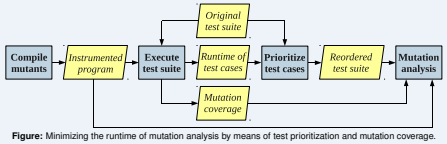


Figure: Minimizing the runtime of mutation analysis by means of test prioritization and mutation coverage.

1. Embed and compile all mutants into the original program
2. Run tests on original program to gather runtime and coverage
3. Sort tests in ascending order according to their runtime
4. Perform mutation analysis while employing the reordered test suite and the mutation coverage information

Runtime of MAJOR's Compiler

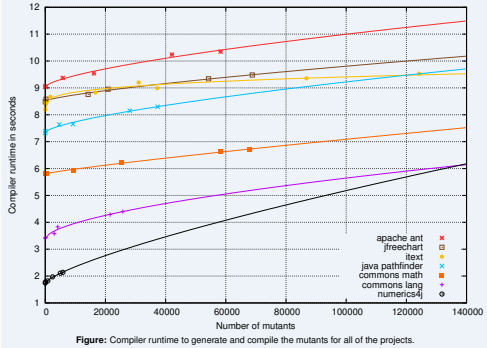


Figure: Compiler runtime to generate and compile the mutants for all of the projects.

- Negligible overhead for generating and compiling the mutants
- Applicable on commodity workstations, even for large projects

Evaluation of Mutation Analysis Processes

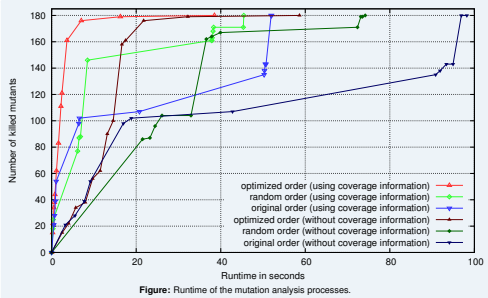


Figure: Runtime of the mutation analysis processes.

Future Work

- Implementing new mutation operators at the semantic level
- Extending the domain specific language to support new operators
- Integrating conditional mutation into the new Java 7 compiler

rene.just@uni-ulm.de

26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)

gkapfham@allegheny.edu

See you tomorrow for a live demonstration!