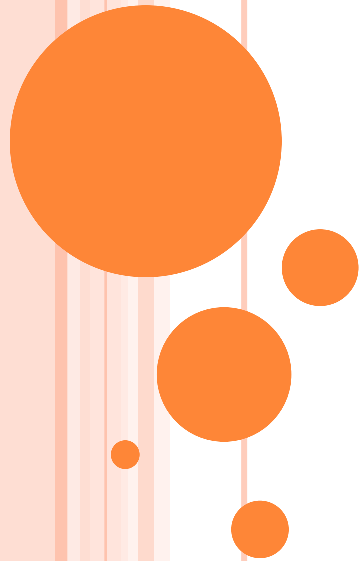
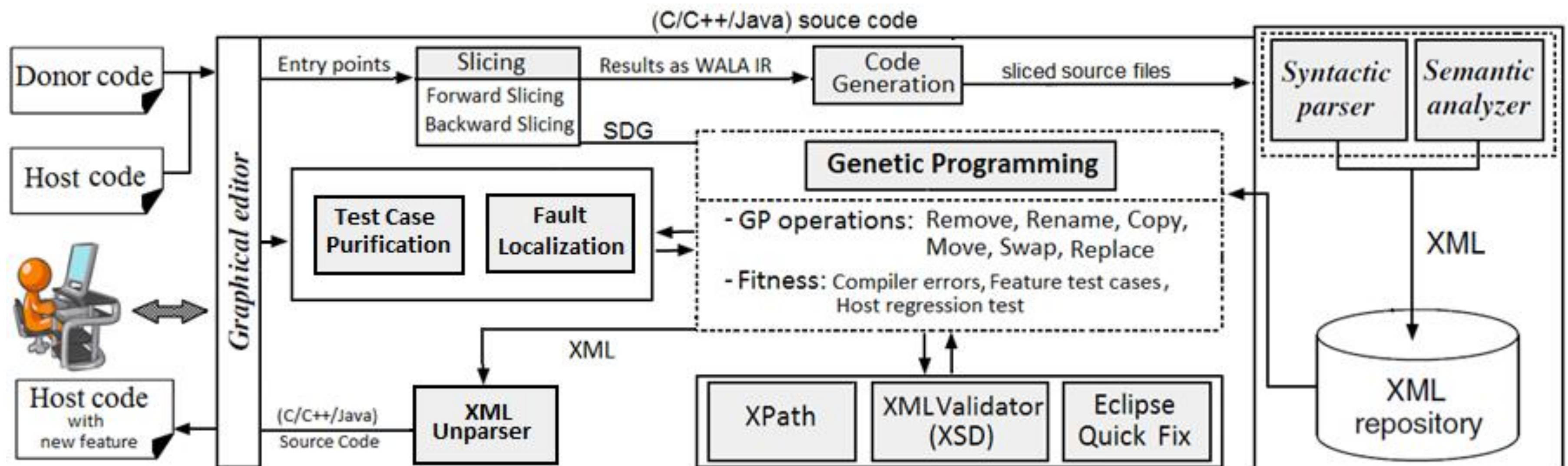
Genetic Programming for Software Transplants

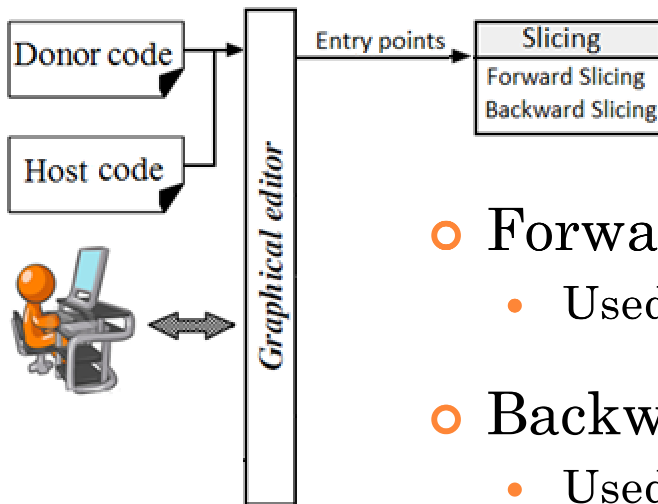
Iman Hemati Moghadam



Implemented Approach: Overview



SLICING:



- Forward Slicing:
  - Used to extract the implementation of the desired feature.
- Backward Slicing:
  - Used to extract how a desired feature is called.

The slicing is implemented using Wala.

CONSOLE OUTPUT FOR SLICING

1. NORMAL_RET_CALLER:Node: < Application, Lc2/apps/klax/comp/ChuteArtist, handle(Lc2/fw/Notification;)V > Context: Everywhere[1]5 = invokevirtual< Application, Lc2/fw/Notification, name()Ljava/lang/String; > 2 @1 exception:4

2. NORMAL handle:8 = invokevirtual< Application, Ljava/lang/String, equals(Ljava/lang/Object;)Z > 5,6 @8 exception:7 Node: < Application, Lc2/apps/klax/comp/ChuteArtist, handle(Lc2/fw/Notification;)V > Context: Everywhere

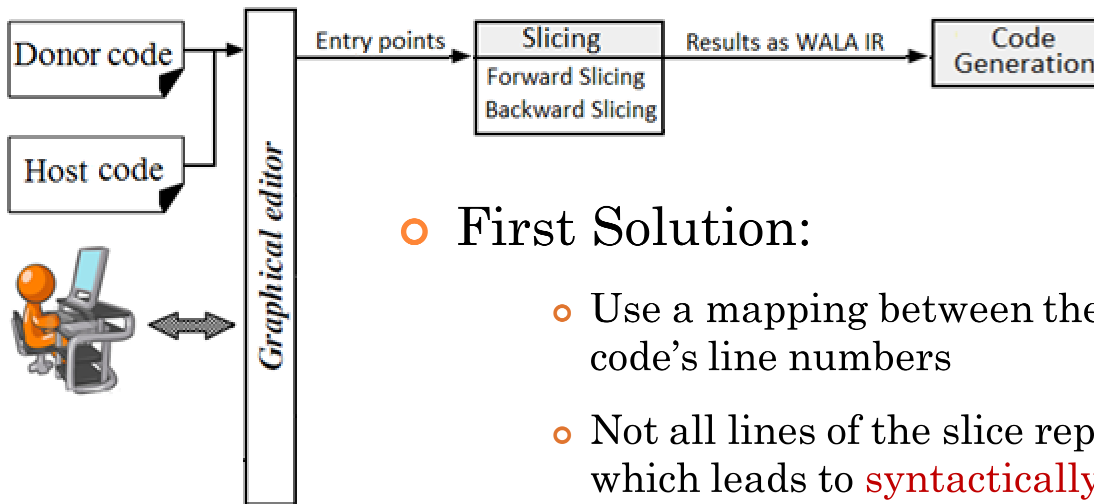
3. PARAM_CALLER:Node: < Application, Lc2/apps/klax/comp/ChuteArtist, handle(Lc2/fw/Notification;)V > Context: Everywhere[5]8 = invokevirtual< Application, Ljava/lang/String, equals(Ljava/lang/Object;)Z > 5,6 @8 exception:7 v5

4. NORMAL handle:12 = invokevirtual< Application, Ljava/lang/String, equals(Ljava/lang/Object;)Z > 5,10 @56 exception:11 Node: < Application, Lc2/apps/klax/comp/ChuteArtist, handle(Lc2/fw/Notification;)V > Context: Everywhere

5. PARAM_CALLER:Node: < Application, Lc2/apps/klax/comp/ChuteArtist, handle(Lc2/fw/Notification;)V > Context: Everywhere[29]12 = invokevirtual< Application, Ljava/lang/String, equals(Ljava/lang/Object;)Z > 5,10 @56 exception:11 v5

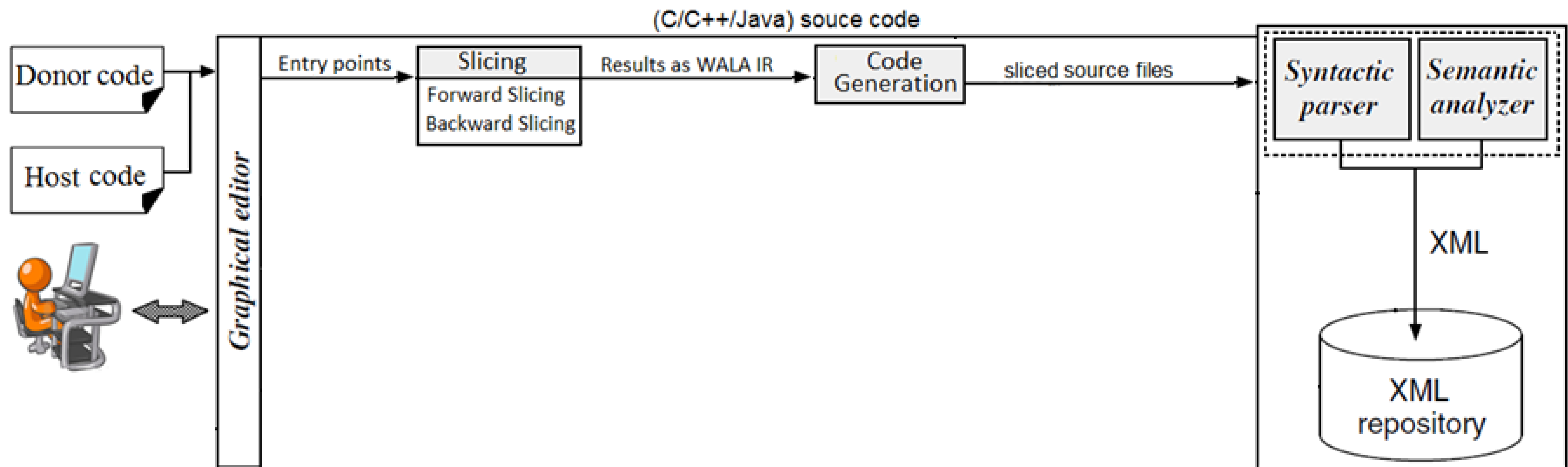
Difficult to translate the generated slices (which is in the form of WALA's IR) back to source code.

Code Generation:



- First Solution:
  - Use a mapping between the slice's statements and the source code's line numbers
  - Not all lines of the slice represent complete Java statements, which leads to syntactically incorrect code
- Second Solution:
  - Transform the source code into an abstract syntax tree rather than using the original source file.

XML Extractor:

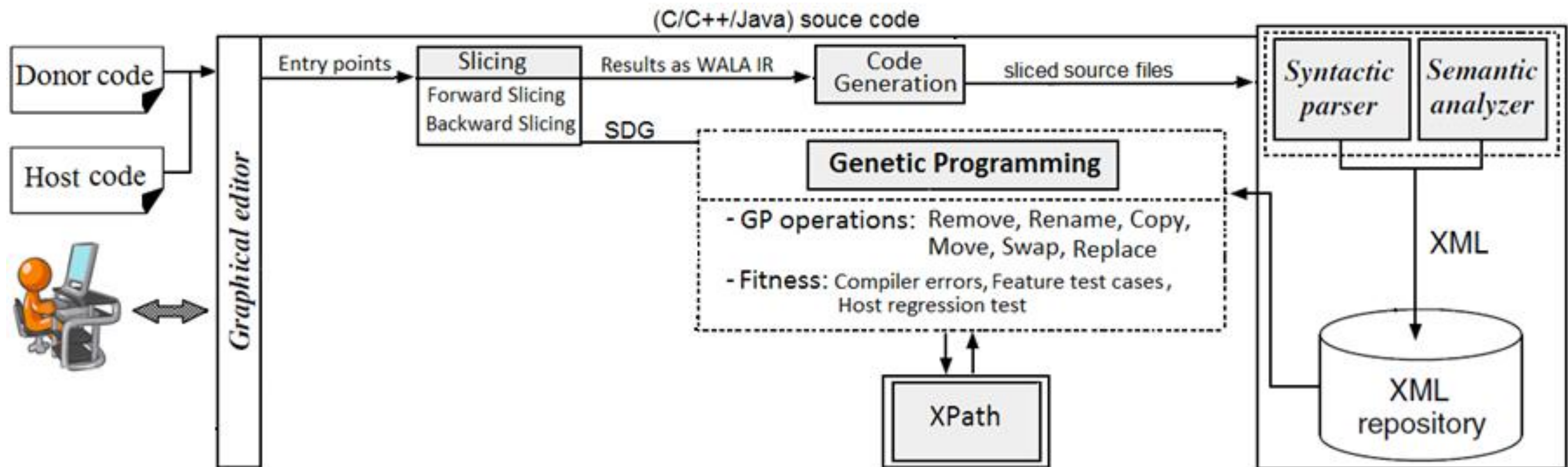


- Opportunistic use of XML technologies
  - Addressing and querying with **xPath**
  - Validating with schema languages such as **XSD**

XML Representation

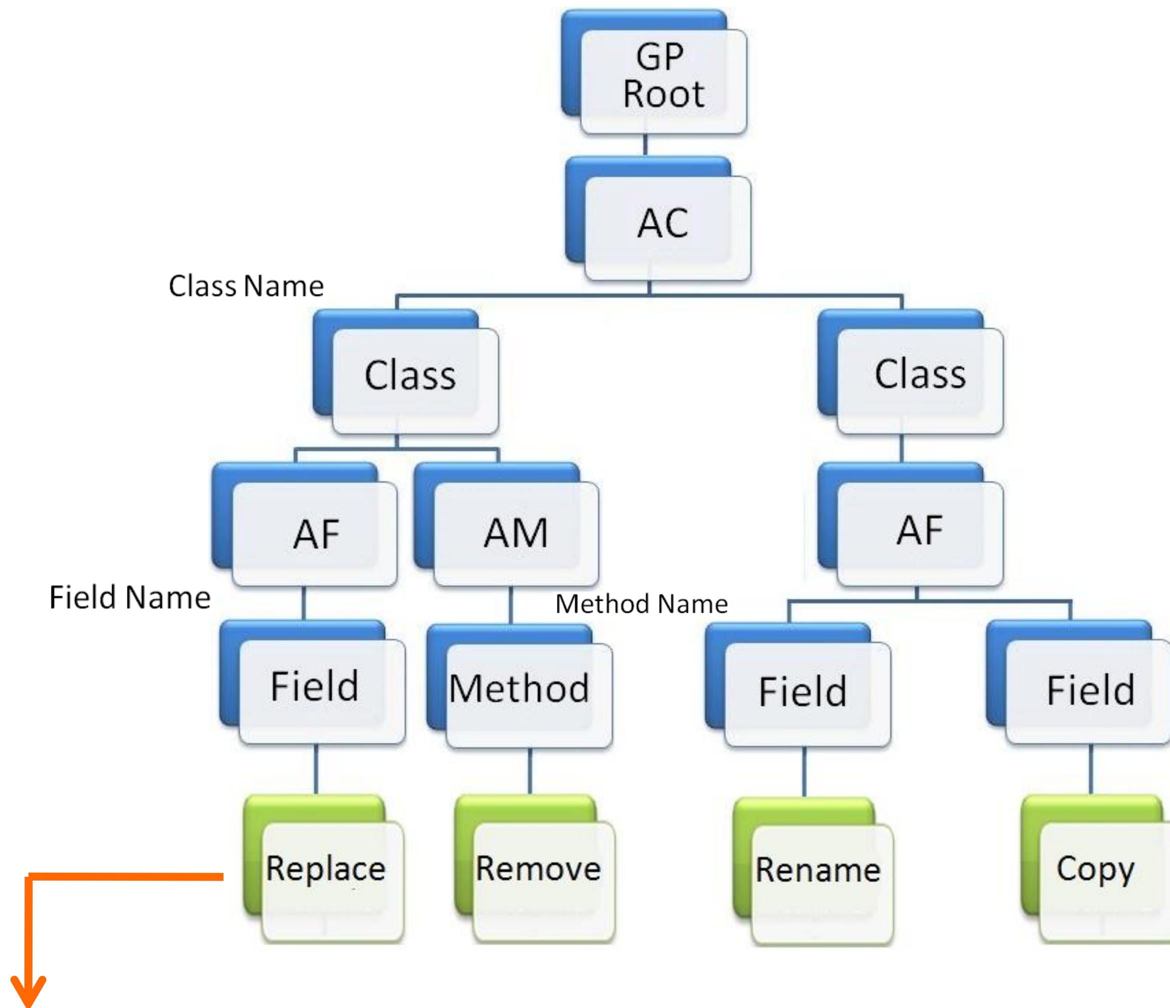
- srcML:
  - A translator from code (C/C++/Java/C#) to srcML , and vice versa
  - A combination of source code (text) and AST information (tags)
- srcML features:
  - Presevation of all source code text (robust to code irregularities)
  - Easy to use and extend (compare it with AST)
  - Scalable translation
    - Translation speed over 25 KLOC/sec

XPATH EXPRESSIONS:



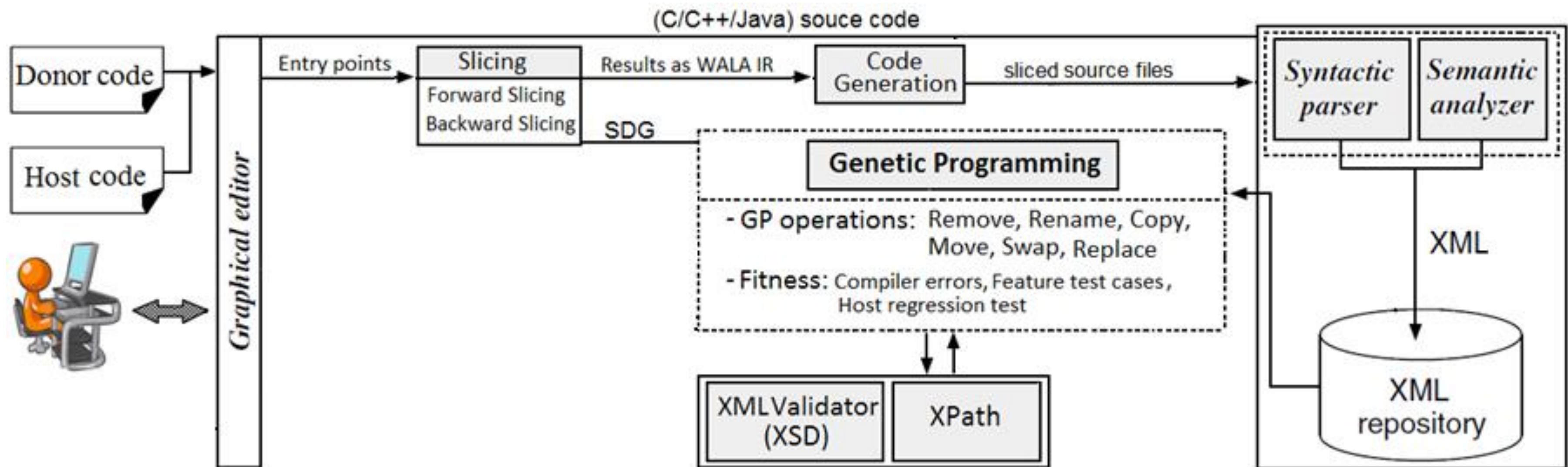
The GP algorithm is implemented using ECJ.

TREE BASED GP



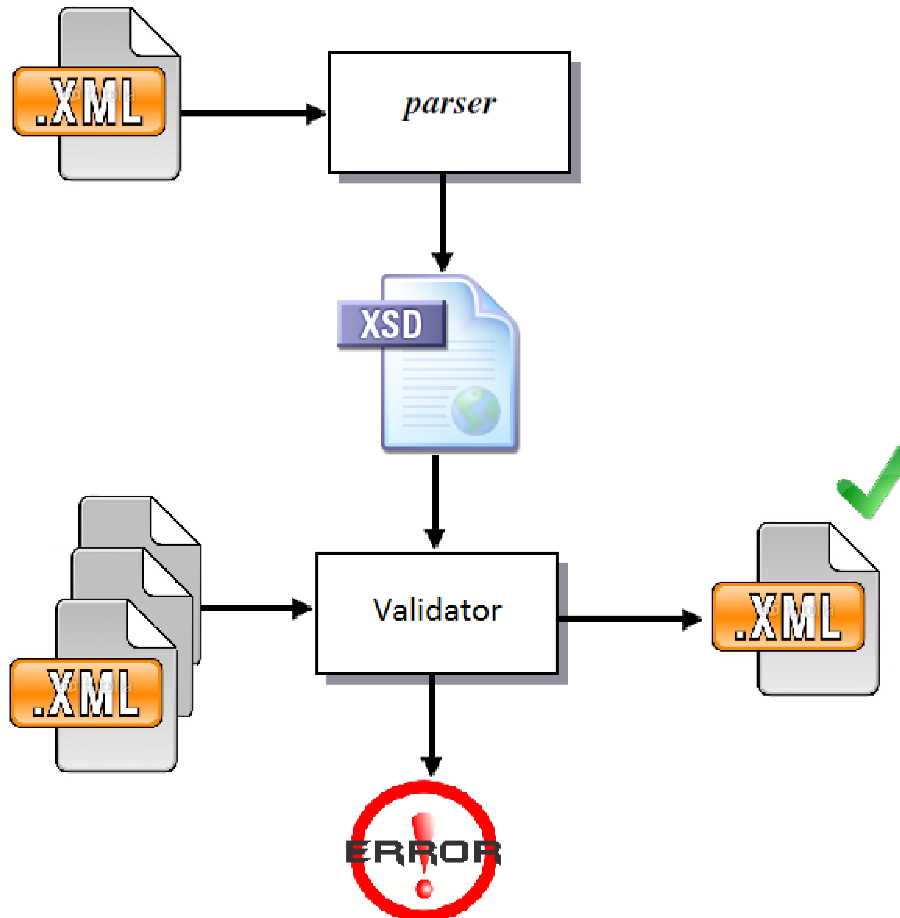

```
Query1 = //unit[1]/class[1]/block[1]/field[3]/type[1]/specifier[1]

Query2 = //unit[2]/class[1]/block[1]/field[1]/type[1]/specifier[1]
```

XML VALIDATOR:

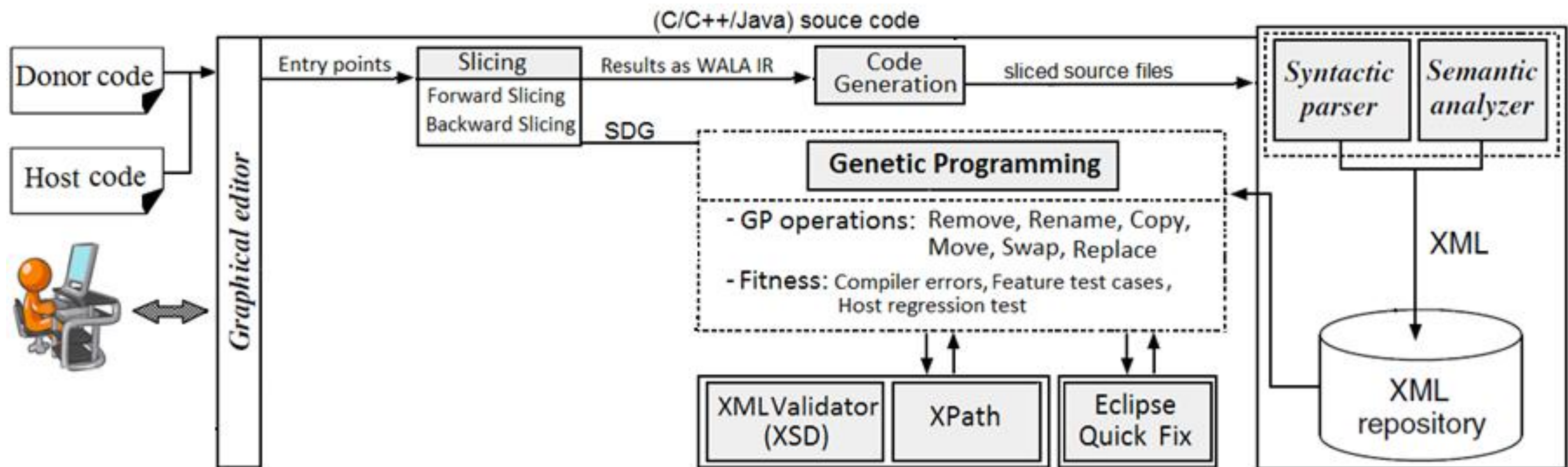


Validating with schema language

- XML Schema Definition (XSD)
  - Defining the restriction on XML data structure, and used for validating XML files.

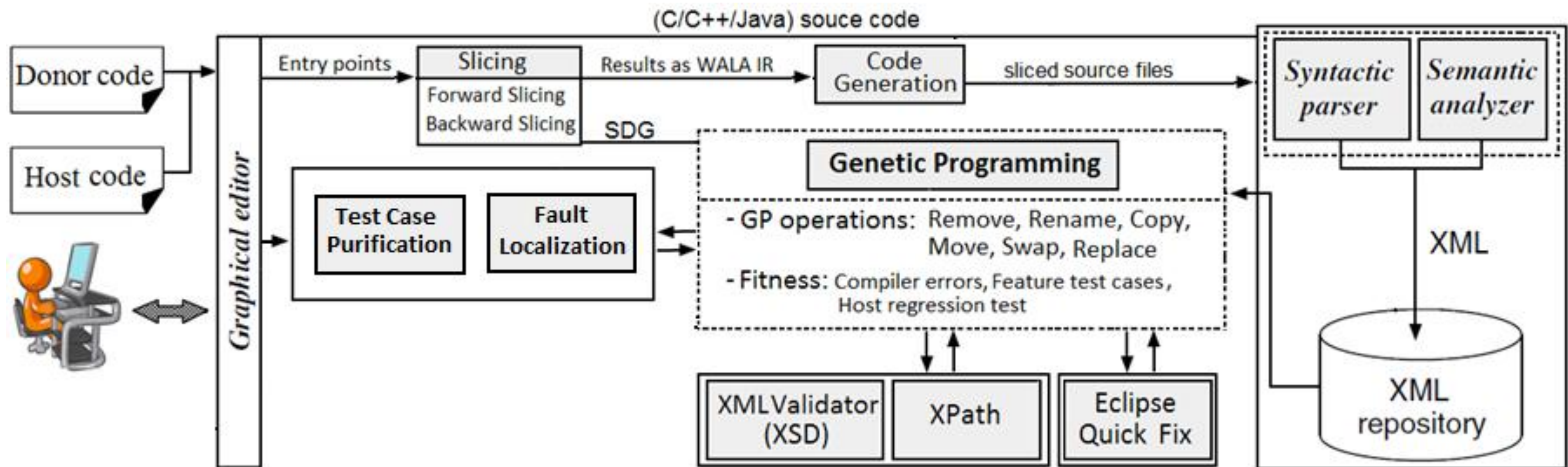


ECLIPSE QUICK FIX:



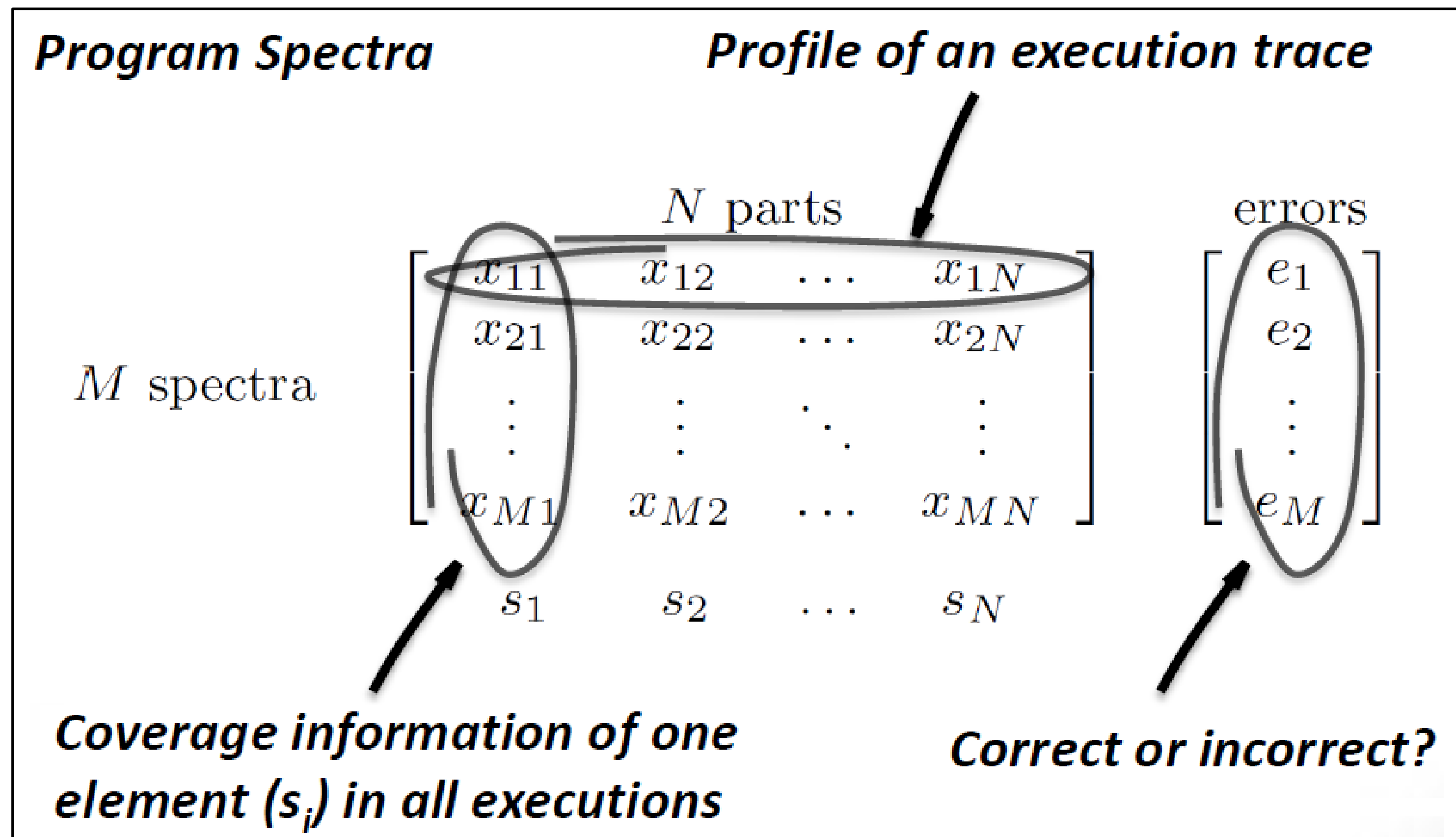
- The current version supports **224** different kind of compiler errors.
- Use also **SDG** in a case that quick fix has no suggestion.

Fault Localization & Test Case Purification:



SPECTRUM-BASED FAULT LOCALIZATION

- Automatically recommend a list of suspicious program elements for inspection based on testing results.



Spectrum-based Fault Localization

- Different SBFL techniques are implemented:
  - **Tarantula, Ochiai, Jaccard, and ...**
  - No strong study of the effectiveness of various SBFL techniques in automated program repair.

- Missing code problem
  - When the logic error caused by missing some code, then no code available to be "suspected".
  - Might be no problem in software transplant, but can be a problem in automated program repair?

Test Case Purification for Improving SBFL

- Generate additional failing test cases to execute all assertions in a given failing test case [1].

test case t1

```
1 Public class targetTest{
2  @Test
3  void t1(){
4    target t = new target();
5    int a=1;
6    assertEquals(2, t.inc(a));
7    int b=1;
8    assertEquals(0, t.dec(b));
9    int c=3;
10   assertEquals(1, t.dec_twice(c));
11 };
12}
```

Target Code	Test case		
	t1		
1 Public class target{			
2 int inc(int n){			
3 return ++n;	●		
4 };			
5 int dec(int n){			
6 return ++n;	●		
7 };			
8 int dec_twice(int n){			
9 n = dec(n);			
10 return dec(n);			
11 };			
12}			

● means the statement is executed by the test case

[1] Xuan, J., & Monperrus, M. "Test Case Purification for Improving Fault Localization.", *FSE, 2014.*

TEST CASE PURIFICATION FOR IMPROVING SBFL

- Generate additional failing test cases to execute all assertions in a given failing test case [1].

Ignore the exception

test case p1

```
1 Public class targetTest{
2  @Test
3  void p1(){
4    target t = new target();
5    int a=1;
6    assertEquals(2, t.inc(a));
7    int b=1;
8    assertEquals(0, t.dec(b));
9    int c=3;
10   assertEquals(1, t.dec_twice(c));
11 };
12}
```

The assertion will be executed

Target Code	Test case		
	t1	p1	
1 Public class target{			
2 int inc(int n){			
3 return ++n;	●	●	
4 };			
5 int dec(int n){			
6 return ++n;	●	●	
7 };			
8 int dec_twice(int n){			
9 n = dec(n);		●	
10 return dec(n);		●	
11 };			
12}			

● means the statement is executed by the test case

[1] Xuan, J., & Monperrus, M. "Test Case Purification for Improving Fault Localization.", *FSE, 2014.*

Test Case Purification for Improving SBFL

- Generate additional failing test cases to execute all assertions in a given failing test case [1].

Slicing

test case p2

```
1 Public class targetTest{
2  @Test
3  void p2(){
4    target t = new target();
5    int a=1;
6    assertEquals(2, t.inc(a));
7    int b=1;
8    assertEquals(0, t.dec(b));
9    int c=3;
10   assertEquals(1, t.dec_twice(c));
11 };
12}
```

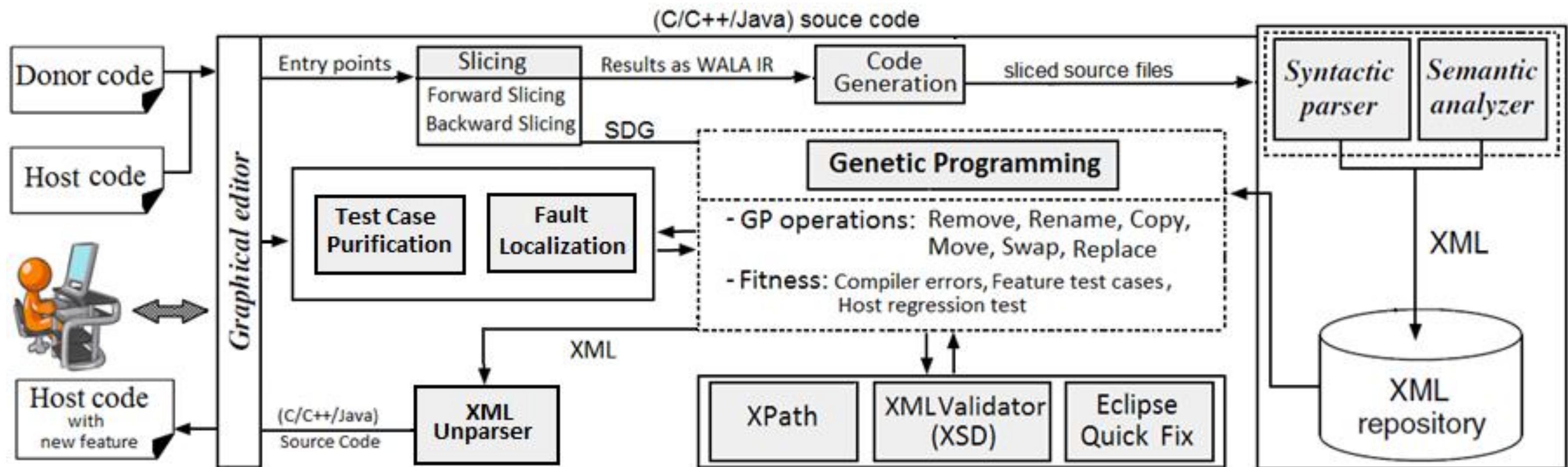
Target Code	Test case		
	t1	p1	p2
1 Public class target{			
2 int inc(int n){			
3 return ++n;	●	●	
4 };			
5 int dec(int n){			
6 return ++n;	●	●	●
7 };			
8 int dec_twice(int n){			
9 n = dec(n);		●	●
10 return dec(n);		●	●
11 };			
12}			

● means the statement is executed by the test case

- Fault localization Improved on **18** to **43%** of faults while performed worse on 1.3 to 2.4% of faults [1].

[1] Xuan, J., & Monperrus, M. "Test Case Purification for Improving Fault Localization.", *FSE, 2014.*

XML Unparser:



EXPERIMENTS

Subject	Type	Functionality
JGAP	Donor	Marshalling Populations to XML
ECJ	Host	
TestCasePurification	Donor	Test case Purification for improving Fault Localization
GZoltar	Host	
Zest	Donor	Layout algorithms, which are currently missing in JGraphT
JGraphT	Host	
JEdit	Donor	Auto indent , and syntax highlighting
Ekit	Host	

Conclusion

- Present a GP Approach: used for both software transplant and program bug repair
- Advantages:
  - Based on XML and xPath
  - Fix compiler errors
  - Use Fault location technique and test case purification

Thank You