AiDD
2024 AI+研发数字峰会
AI+ Development Digital summit
AI驱动研发变革 促进企业降本增效
北京站 08/16-17

大语言模型时代的变异分析

王博 北京交通大学

演讲嘉宾

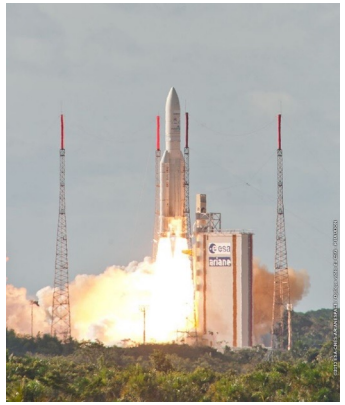
王博

北京交通大学计算机与信息技术学院讲师、硕士生导师，CCF专业会员、CCF系统软件专委执行委员、CCF开源发展委员会执行委员。分别于北京大学、中国科学技术大学和中南大学获得博士、硕士和学士学位。研究兴趣为软件测试与调试，已在ASE、ISSTA、TOSEM、软件学报等发表多篇论文。担任ASE、FSE、ICST、Internetware 等软件工程重要会议PC，担任TSE、TOSEM、TDSC、EMSE、JSS、ASEJ、IET Software、JSME、软件学报等多个期刊审稿人。获得ISSTA 2017 杰出论文奖，全国大学生系统能力大赛优秀指导教师和北京市高校优质教案。

目录
CONTENTS

PART 01
背景

软件正确性至关重要



欧空局 Ariane 5

波音 737 Max 坠机

海湾战争中爱国者飞弹

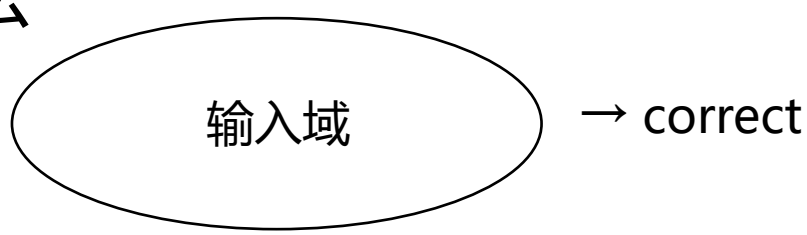
7.23事故

- 软件缺陷已经导致很多灾难性后果
- 保障软件的正确性十分重要
- 当我们说软件是正确的：程序的行为符合正确性规约 (specification)

保障正确性的方法

```
int foo(int a, int b) {
   return a + b;
}
```

1. 形式化方法

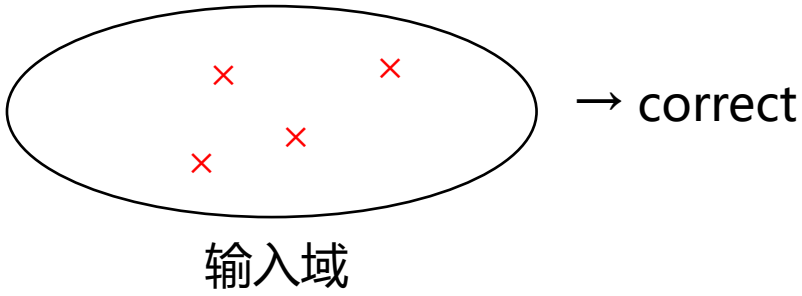


形式化方法成本很高

Formal Method:

```
(a >= 0 && b <=0) ||
(a <= 0 && b >=0) ||
(a >= 0 && b >= 0 && a + b >= 0) ||
(a <= 0 && b <= 0 && a + b <= 0)
```

2. 软件测试



测试是不完备的!

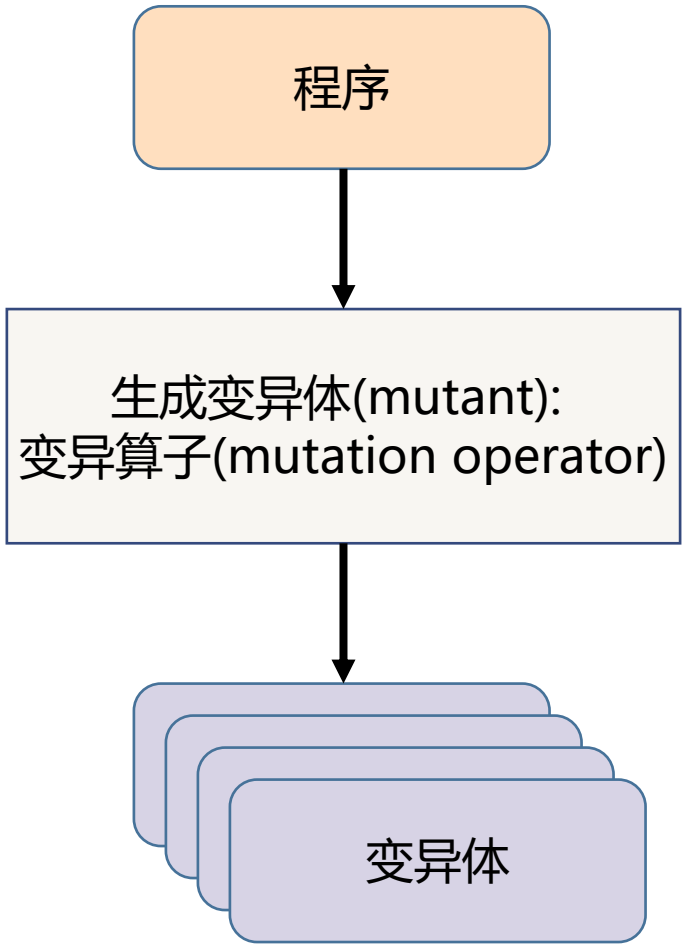
Testing:

```
foo(0, 1) = 1;
foo(INT_MAX, 1) = ERROR;
foo(INT_MAX - 1, 1) = INT_MAX ;
foo(INT_MAX, INT_MIN) = -1;
```

测试质量直接影响到软件质量

- 核心问题是：我们如何度量测试的好坏？
  - 测试质量达标的系统才有一定的可信度
  - 测试集约减
  - 测试排序
- 我们朴素的愿望：希望测试能发现真实缺陷
- 但是在发现之前，真实的缺陷对于我们是未知的
  - “测试可以非常有效地显示bug存在，但却无法证明bug的不存在”
- 我们可以使用一些指标，间接地度量测试质量
  - 测试覆盖
  - 变异测试：用人造缺陷发现率估计真实缺陷发现率

变异测试概览



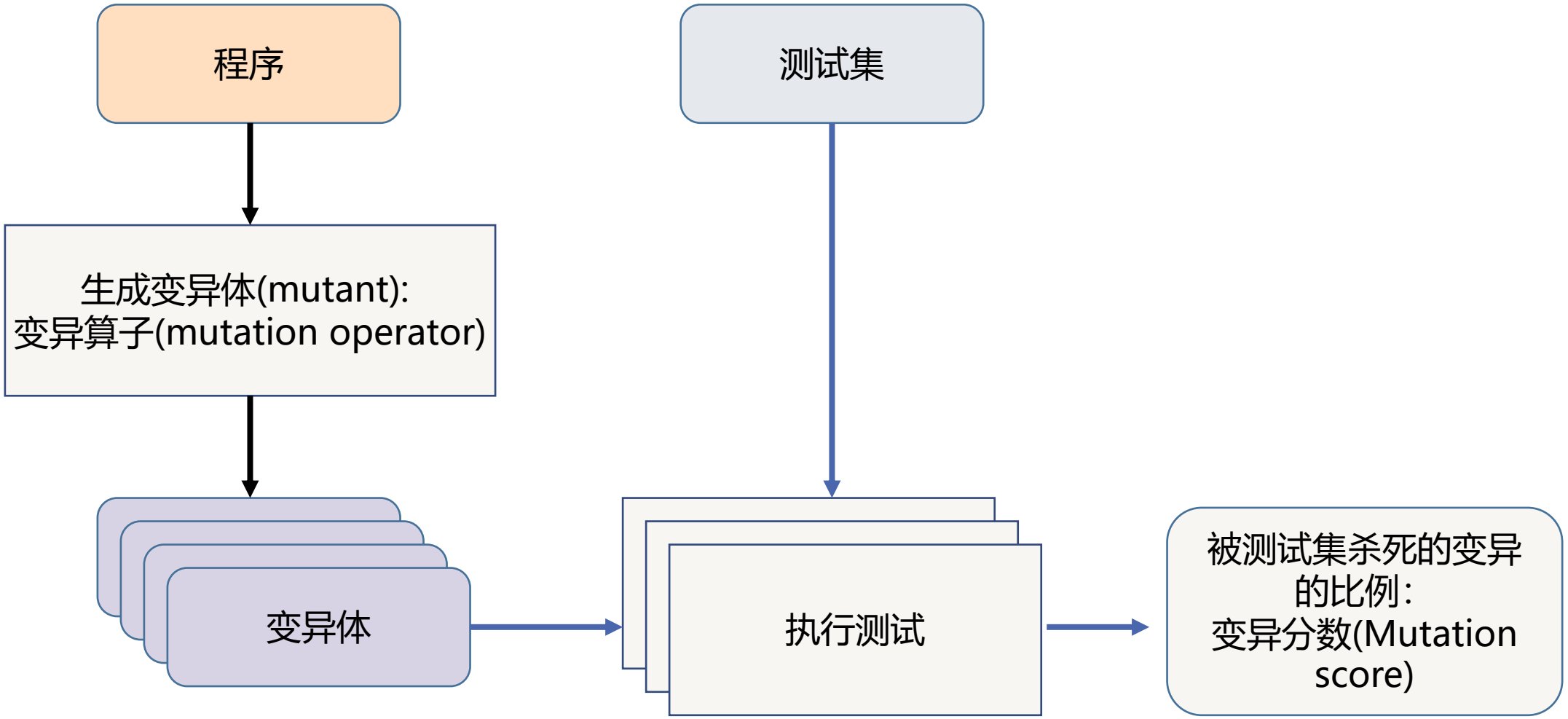

```
public int max(int a, int b){
   return (a > b) ? a : b;
}
```
Original

```
public int max(int a, int b){
   return (a >= b) ? a : b;
}
```
Mutant 1

```
public int max(int a, int b){
   return (a != b) ? a : b;
}
```
Mutant 2

每个变异体是原始程序的小型文法改动

变异测试概览


程序
测试集
生成变异体(mutant):
变异算子(mutation operator)
变异体
执行测试
被测试集杀死的变异的比例:
变异分数(Mutation score)

变异测试在软件测试中的发展

- 变异测试自 1971 年被 DeMillo 和 Hamlet 提出以来，是软件测试中的重要方法
- 修改位置：从一阶变异(first order)到高阶(higher-order)，支持修改多处
- 在单元测试中：
  - 面向高级语言源码：C、Java、Python、JS…
  - 面向中间表示：Java bytecode, LLVM-IR
  - 从桌面应用到 Android、MPI、智能合约程序等
- 从单元测试扩展到其他测试阶段：
  - 集成测试
  - 设计阶段（例如在基于模型的软件开发过程中针对设计FSM的变异）

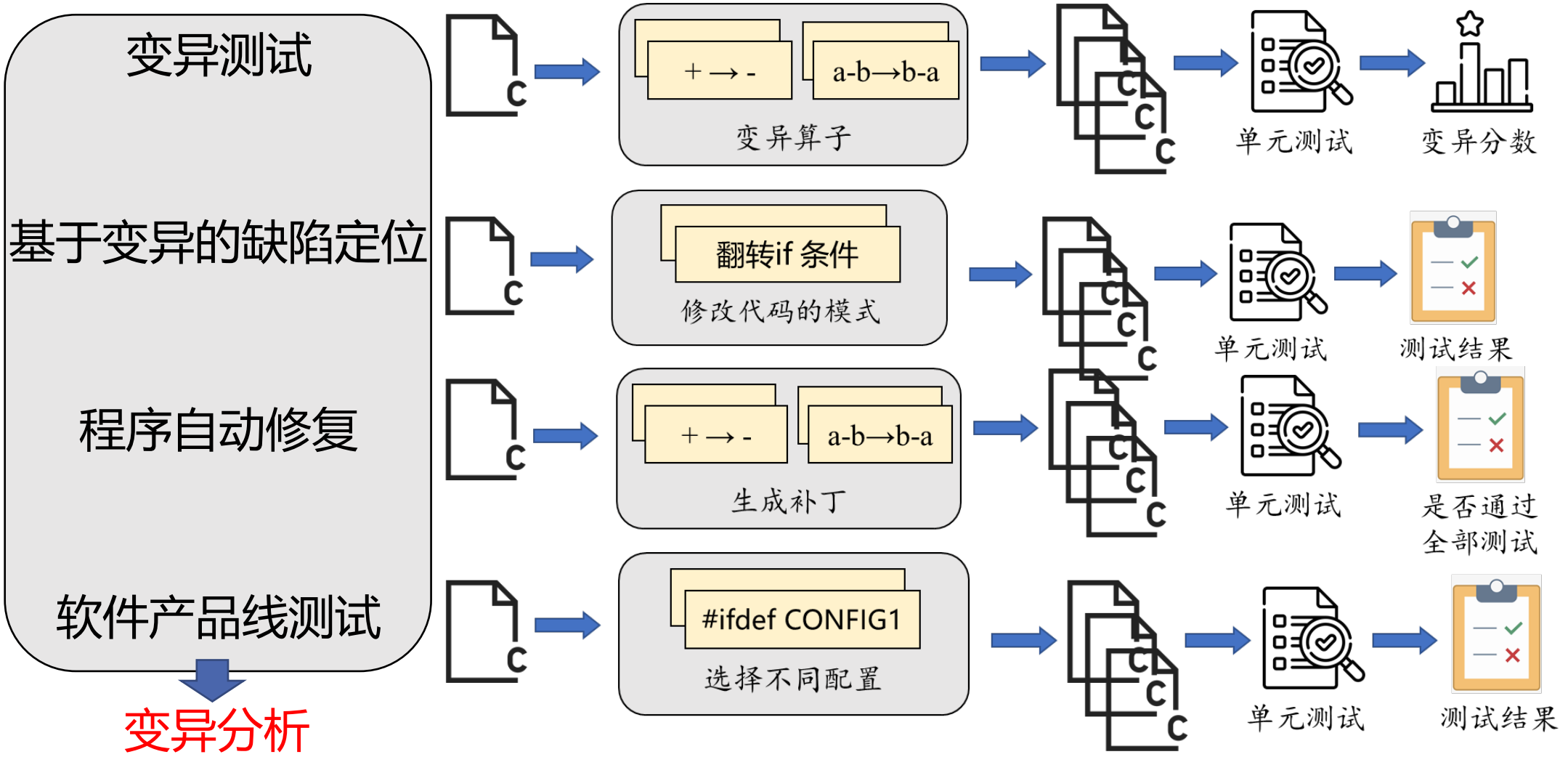
从变异测试到变异分析

- 基于变异的缺陷自动定位（mutation-based fault localization）是变异测试的衍生技术
- 缺陷自动定位：给定测试集（至少有一个未通过测试）和程序，返回程序中的语句出错可疑度分数。
- 传统定位方法：基于测试覆盖信息对语句排序（spectrum-based fault localization）
- 基于变异的方法：通过观察变异对测试结果的改变计算可疑度
  - 若一个变异使失败测试通过了，那更可能是变异了出错语句

$$S_{mutant}(m_i) = \frac{failed(m_i)}{\sqrt{totalfailed \cdot (failed(m_i) + passed(m_i))}}$$

```
double compute(double[] nums){
    int n = nums.length;
    double sum = 0;
    for(int i = 0; i < n; i++){
        sum += nums[i];
        ......
    }
    return sum;
}
```

从变异测试到变异分析


变异测试
基于变异的缺陷定位
程序自动修复
软件产品线测试
变异分析
+ → -
a-b→b-a
变异算子
翻转if 条件
修改代码的模式
生成补丁
#ifdef CONFIG1
选择不同配置
单元测试
变异分数
测试结果
是否通过
全部测试

工具和标准程序集

- 变异测试
  - C 语言:
    - Proteum (108 mutation operators)
    - Milu (Higher-order)
    - WinMut (IR-based)
  - Java 语言:
    - MuJava
    - Major [By Rene Just]
    - JavaLanch (IR-based) [By Zeller]
    - PITest (Commercial tool)
  - 标准测试集
    - Software-artifact Infrastructure Repository (somehow outdated)
    - Defects4J (Java)
    - ManyBugs (C)

PART 02
痛点

挑战1：生成高质量的变异

Name	Description	Example
AOR	Replace arithmetic operator	$a + b \rightarrow a - b$
LOR	Replace logic operator	$a \,\&\, b \rightarrow a \mid b$
ROR	Replace relational operator	$a == b \rightarrow a >= b$
LVR	Replace literal value	$T \rightarrow T + 1$
COR	Replace logical connector	$a \,\&\&\, b \rightarrow a \mid\mid b$
SOR	Replace shift operator	$a >> b \rightarrow a << b$
STDC	Delete a call	$f() \rightarrow nop$
STDS	Delete a store	$a = 5 \rightarrow nop$
UOI	Insert a unary operation	$b = a \rightarrow a++;\ b = a$
ROV	Replace the operation value	$f(a, b) \rightarrow f(b, a)$
ABV	Take absolute value	$f(a, b) \rightarrow f(abs(a), b)$

- 现有方法生成变异主要有两类
- 基于规则的传统方法
  - PIT、Major、JavaLanche、AccMut
  - 生成变异过于简单，不考虑代码上下文，文本上与真实bug差别很大
- 基于学习的方法
  - 训练模型，学习真实 Bug 的代码变换模式
  - DeepMutation [ICSME-19]、LEAM [ASE-22]
  - 训练数据集有限，生成大量错误变异

挑战2：可扩展性较低

- 阻碍变异分析走向工业实践的是可扩展性（Scalability）较低
- 假如有 M 个变异，N 个测试
- 变异分析的计算复杂度：O(M) + O(M×N)
  - $T_{total} = \sum_{m \in M} t_{seed,m} + \sum_{m \in M} t_{compile,m} + \sum_{m \in M} \sum_{n \in N} t_{test,m,n}$
- 在实际规模的程序中，M 会很大

挑战2：可扩展性较低

- 以变异测试为例，如下一行源码产生二十多个变异

```
                    a > b ?  a :  b

a >= b ? a : b          0 > b ? a : b          -a > b ?  a : b
a <  b ? a : b          a > 0 ? a : b           a >-b ?  a : b
a <= b ? a : b          a > b ? 0 : b           a > b ? -a : b
a != b ? a : b          a > b ? a : 0           a > b ?  a :-b
a == b ? a : b          0                       a > b ?  a :-b
(a > b) ? a : b         b > a ? a : b          -(a > b ?  a : b)
true  ? a : b           a > b ? b : a           a
false ? a : b                                   b
```

挑战3：等价变异体

- 等价变异体：变异后的程序与原始程序在功能上完全相同，即它们对所有可能的输入产生相同的输出。
- 等价变异的危害：
  - 增加测试成本
  - 影响变异分数计算的精确度

```
for (int i = 0; i < 10; i++)
```

```
for (int i = 0; i != 10; i++)
```

- 判断等价变异体是不可判定问题：不存在一个自动算法完美解决

PART 03
解决思路

面向挑战1：大模型时代的变异生成

- 大模型在代码理解和代码变换上出色的能力，为提升变异分析提供了新的方向
- 大模型在程序修复上取得了显著的效果
  - buggy -> correct
- 问题：大模型在变异分析上能否取得良好效果?

面向挑战2：基于共享计算的加速

- 变异分析执行过程中需要反复执行测试
- 其中存在大量冗余计算
- 我们可以尝试共享冗余计算进行加速

PART 04
具体实现

基于大模型的变异生成

- 大模型能否生成更接近真实bug的变异?
- 有效性:
  - 语法正确
  - 变异有“揭错”能力
- 自然性:
  - 符合一般的编程规范
  - 编码模式和习惯与真实代码一致
- 变异足够多样，且有足够数量的变异被杀死
  - 避免生成等价变异
  - 生成的有足够比例的变异能被测试检测
  - 语义尽可能丰富

基于大模型的变异生成：模型设置

- 选用了 4 个模型
- 包括开源模型和商用闭源模型（实验还在扩展中）

Model Type	Model	Training Data Time	Release Time
Closed	GPT-3.5-turbo	2021/09	2023/03
	GPT-4-turbo	2023/04	2023/06
Open	CodeLlama-13b-Instruct	—	2023/08
	StarChat-β-16b	—	2023/06

- GPT-3.5-Turbo 和 GPT-4-Turbo 通过购买 API Token
- 开源模型通过租用 2 台 双卡 3090 服务器

基于大模型的变异生成：数据集

- 面向 Java 语言（Java 是拥有最多 Baseline 的语言）
- Defects4J v1.20 上 395 个真实的缺陷
- ConDefects 上的 45 个没有数据泄露风险的缺陷

Dataset	Project	# of Bugs	Time Span
Defects4J	Math	106	2006/06/05 - 2013/08/31
	Lang	65	2006/07/16 - 2013/07/07
	Chart	26	2007/07/06 - 2010/02/09
	Time	27	2010/10/27 - 2013/12/02
	Closure	133	2009/11/12 - 2013/10/23
	Mockito	38	2009/06/20 - 2015/05/20
ConDefects	—	45	2023/09/01 - 2023/09/30
Total	—	440	2006/07/16 - 2023/09/30

基于大模型的变异生成：对比方法

- 我们的对比方法涵盖了所有的 Java 最新变异生成方法
- 基于规则
  - PIT [ISSTA-16]
  - Major [ISSTA-11]
- 基于学习
  - LEAM [ASE-22]
- 基于小规模的预训练模型
  - muBert [ICST-22]

基于大模型的变异生成：Prompt设计

为了避免数据泄露，选用了QuixBug中的6个真实缺陷作为 Few-shot Example

'{WHOLE_JAVA_METHOD}'

Above is the original code. your task is to generate '{MUT_NUMBER}' mutants, (notice: mutant refers to the mutant in software engineering, i.e., making subtle changes to the original code) in: '{CODE_ELEMENT}', as follows are some examples of mutants which you can refer to:

…

{

"precode": "while (Math.abs(x-approx*approx) > epsilon) { "

"aftercode": " while (Math.abs(x-approx) > epsilon) {",

}

…

#Requirement:

1. Provide generated mutants directly
2. A mutation can only occur on one line
3. Your output must be like:

[{ "id":, "line":, "precode":"", "filepath":"kk", "aftercode":"" }],

where "id" stands for the mutant serial number, "Line" represents the line number of the mutated, "precode" represents the line of code before mutation and it can not be empty, "aftercode" represents the line of code after mutation

4. Prohibit generating the exact same mutants
5. All write in a JSON file

Context (Whole Method)

Instructions and ***Input Data***

Context (Few-Shot Examples)

Output Indicator

基于大模型的变异生成：评估生成代价

- 代价指标：
  - 时间代价（秒）
  - 平均生成1k个变异的代价

Metric Category	Metric	GPT 3.5		CodeLlama-13b		LEAM		μBERT		PIT		Major	
		D4J	CD	D4J	CD	D4J	CD	D4J	CD	D4J	CD	D4J	CD
–	Mut. Score	0.731		0.734		0.716		0.698		0.529		0.682	
	Mut. Count	351,557		303,166		671,737		141,031		1,032,392		314,694	
Cost	Avg. Gen. Time	1.79		9.06		3.06		2.23		0.017		0.083	
	$ Cost per 1K Mut.	0.288		0.577		0.195		–		–		–	

基于大模型的变异生成：评估可用性

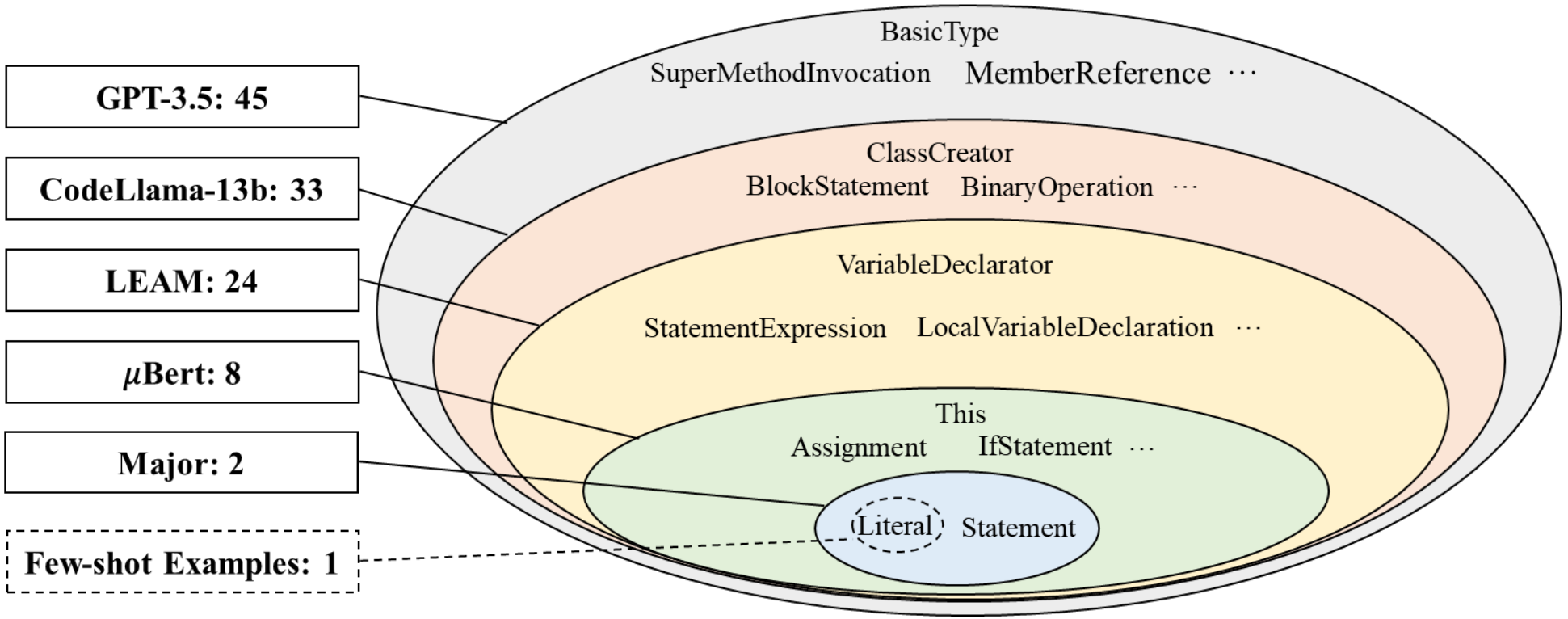
- 机器学习方法不可避免地生成无用变异
- 可用性指标：
  - 编译通过率
  - 无效变异率
  - 等价变异率（按95%置信度和5%误差幅度采样）

Metric Category	Metric	GPT 3.5		CodeLlama-13b		LEAM		μBERT		PIT		Major	
		D4J	CD	D4J	CD	D4J	CD	D4J	CD	D4J	CD	D4J	CD
Usability	Compilability Rate	61.7%	79.6%	75.8%	73.3%	33.8%	50.9%	21.9%	26.6%	—	—	98.3%	92.8%
	Useless Mut. Ratio	11.6%	9.2%	41.0%	9.3%	0.5%	9.6%	1.8%	4.6%	—	—	0.0%	0.0%
	Eq. Mut. Rate	2.3%	2.1%	1.3%	0.7%	1.6%	1.0%	2.1%	3.1%	—	—	0.5%	0.8%

Dataset	Type	GPT-3.5	CL-13b	LEAM	μBert	Major
Defects4J	# All	351,332	302,941	671,342	140,753	314,527
	# Sampled	384	384	384	384	384
	# Eq. Mut.	9	5	6	8	2
ConDefects	# All	225	225	395	278	167
	# Sampled	143	143	196	162	123
	# Eq. Mut.	3	1	2	5	1

基于大模型的变异生成：评估变换多样性

- 变换多样性指标：
  - 变异是否引入新的 AST 节点类型 (例如，a + b -> a – b 没有引入新类型，但 a + b -> foo(a, b) 引入了方法调用)



基于大模型的变异生成：评估变换多样性

- 变换多样性指标：
  - 删除变异所占比例
  - 引入新的 AST 节点类型分布

Top K	GPT-3.5	CodeLlama-13b	LEAM	μBERT	Major
Del.	5.1%	0.3%	49.7%	0.2%	17.6%
1st	LT (47.6%)	LT (49.5%)	LT (23.6%)	LT (53.8%)	ST (59.9%)
2nd	BO (24.7%)	BO (35.5%)	MI (17.6%)	MR (22.7%)	LT (40.1%)
3rd	MI (8.2%)	MI (4.1%)	MR (16.3%)	MI (12.5%)	–

ST: Statement, BO: BinaryOperation, LT: Literal, MI: MethodInvocation, MR: MemberReference

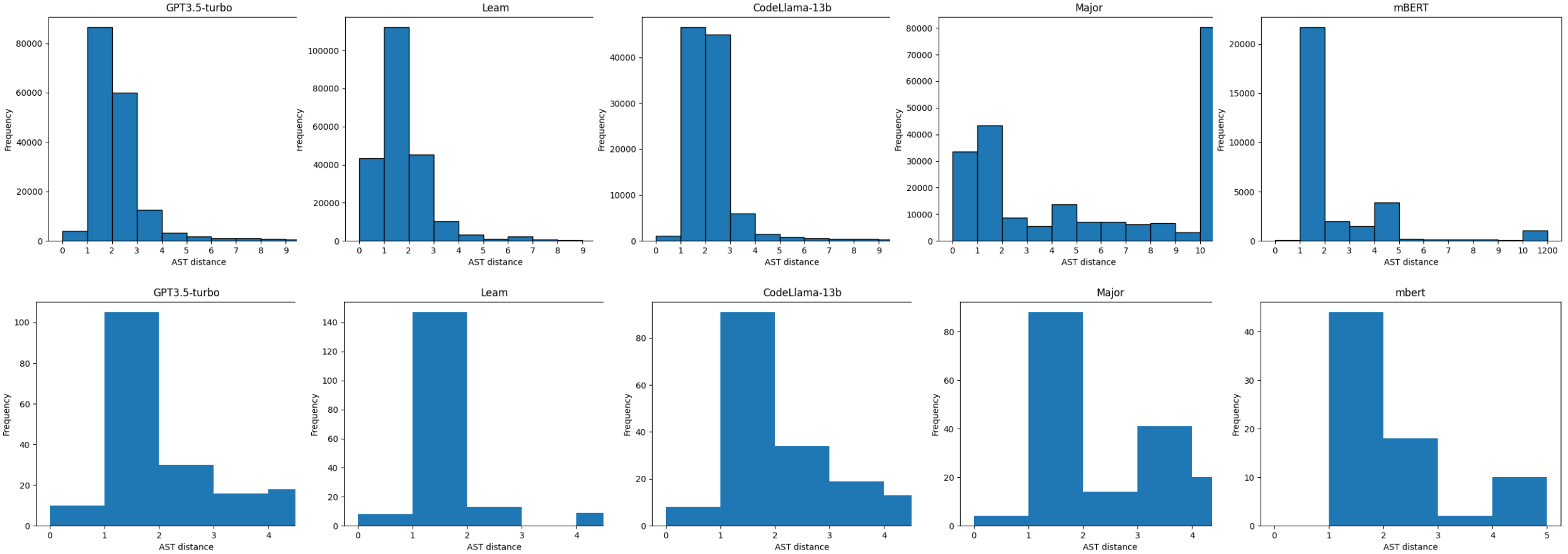
基于大模型的变异生成：与真实缺陷的语法相似度

- 指标：
  - 变异与真实缺陷的 BLEU 分数
  - 变异与真实缺陷的 AST 编辑距离

Metric Category	Metric	GPT 3.5		CodeLlama-13b		LEAM		μBERT		PIT		Major	
		D4J	CD	D4J	CD	D4J	CD	D4J	CD	D4J	CD	D4J	CD
Syntactic	BLEU Score	0.649	0.672	0.703	0.658	0.248	0.307	0.709	0.676	–	–	0.226	0.316
	AST Distance	2.369	1.62	2.413	1.65	3.07	1.19	2.631	1.74	–	–	28.284	1.98

基于大模型的变异生成：与真实缺陷的语法相似度

- 与真实缺陷的AST编辑距离



基于大模型的变异生成：与真实缺陷的行为相似度

- 行为相似度指标：
  - 真实缺陷检测率
  - 与真实缺陷耦合率
  - Ochiai 系数（语义相似度指标）

Metric Category	Metric	GPT 3.5		CodeLlama-13b		LEAM		μBERT		PIT		Major	
		D4J	CD	D4J	CD	D4J	CD	D4J	CD	D4J	CD	D4J	CD
Behavior	Real Bug Detectability	0.967	0.867	0.906	0.667	0.767	0.489	0.785	0.378	0.511	0.489	0.916	0.689
	Coupling Rate	0.416	0.625	0.398	0.612	0.267	0.318	0.409	0.581	0.133	0.593	0.342	0.479
	Ochiai Coefficient	0.638	0.689	0.390	0.378	0.286	0.400	0.230	0.578	0.271	0.489	0.519	0.600

基于大模型的变异生成：不同Prompt的影响

- P1：默认prompt
- P2：P1 移除few-shot example
- P3：P2 移除方法 Context
- P4：P1 + 添加方法对应的测试

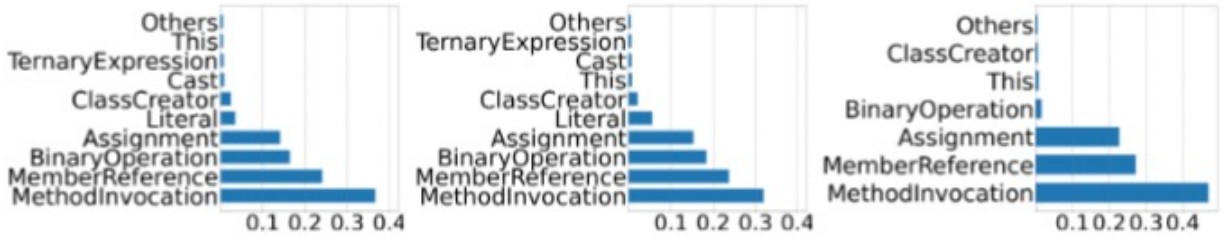
Metric Category	Metric	GPT-3.5			
		P1	P2	P3	P4
–	Mutation Count	47,666	47,905	41,548	39,638
	Mutation Score	0.705	0.703	0.731	0.782
Cost	$ Cost per 1K Mutations	0.288	0.288	0.287	0.289
	Average Generation Time	1.79	1.78	1.81	1.88
Usability	Compilability Rate	65.7%	64.1%	61.9%	60.8%
	Useless Mutation Ratio	13.7%	11.3%	22.3%	15.0%
	Equivalent Mutation Rate	1.5%	1.2%	1.2%	1.5%
Behavior	Real Bug Detectability	0.895	0.857	0.714	0.629
	Coupling Rate	0.140	0.136	0.096	0.096
	Ochiai Coefficient	0.590	0.571	0.438	0.362

基于大模型的变异生成：不同模型的表现

Metric Category	Metric	Closed Source LLM		Open Source LLM	
		GPT-3.5	GPT-4	StarChat-16b	CodeLlama-13b
—	Mutation Count	47,666	23,526	25,496	41,808
	Mutation Score	0.705	0.702	0.538	0.734
Cost	$ Cost per 1K Mutations	0.288	18.701	0.876	0.577
	Average Generation Time	1.79	42.25	13.76	9.06
Usability	Compilability Rate	65.7%	79.9%	58.2%	74.1%
	Useless Mutation Ratio	13.7%	7.2%	38.3%	39.2%
	Equivalent Mutation Rate	1.5%	0.9%	2.1%	1.0%
Behavior	Real Bug Detectability	0.895	0.905	0.324	0.724
	Coupling Rate	0.140	0.143	0.101	0.110
	Ochiai Coefficient	0.590	0.619	0.152	0.286

基于大模型的变异生成：导致不可编译变异的原因

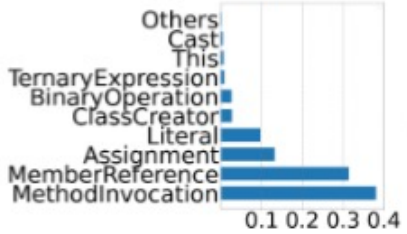
ID	Error Type	%	ID	Error Type	%
1	Usage of Unknown Methods	27.3%	6	Type Mismatch	7.6%
2	Code Structural Destruction	22.9%	7	Incorrect Initialization	3.1%
3	Incorrect Method Parameters	12.8%	8	Incorrect Location	3.1%
4	Usage of Unknown Variables	11.2%	9	Incorrect Exceptions	2.3%
5	Usage of Unknown Types	9.6%	—	—	—



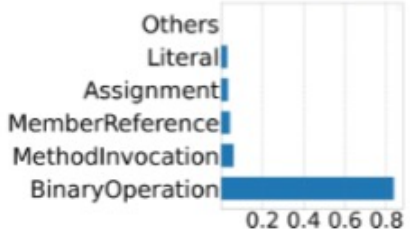
(a) GPT-3.5

(b) CodeLlama-13b

(c) LEAM



(d) μBert



(e) Major

基于大模型的变异生成：不同上下文长度

Metric Category	Metric	1-Line Context	2-Line Context	3-Line Context
Usability	Compilability Rate	0.610	0.617	0.657
	Useless Mutation Ratio	0.119	0.128	0.137
	Equivalent Mutation Rate	1.6%	1.1%	1.3%
Behavior	Real Bug Detectability	0.933	0.917	0.917
	Coupling Rate	0.141	0.121	0.136
	Ochiai Coefficient	0.517	0.417	0.467

Settings	Spea. Coef.	*p*-value	Pear. Coef.	*p*-value
1-Line v.s. 2-Line	0.9762	3.31E-05	0.9980	2.12E-08
2-Line v.s. 3-Line	0.9762	3.31E-05	0.9997	1.04E-10
1-Line v.s. 3-Line	0.9524	2.60E-04	0.9978	2.69E-08

基于大模型的变异生成：不同的Few-shot Example

Metric	QB-6a (Default)	QB-3	QB-6b	QB-9	D4J-6
Comp. Rate	65.7%	62.7%	63.9%	61.8%	62.1%
Useless Mut. Ratio	13.7%	12.8%	12.8%	13.8%	12.5%
Equivalent Mutation Rate	1.3%	1.6%	1.3%	1.1%	1.6%
Real Bug Det.	0.917	0.933	0.917	0.917	0.900
Coupling Rate	0.136	0.128	0.132	0.121	0.139
Ochiai Coefficient	0.467	0.450	0.434	0.434	0.367

QB-N: Select N Examples from QuixBug. QB-6a is the set of default examples, while QB-6b involves 6 other examples.

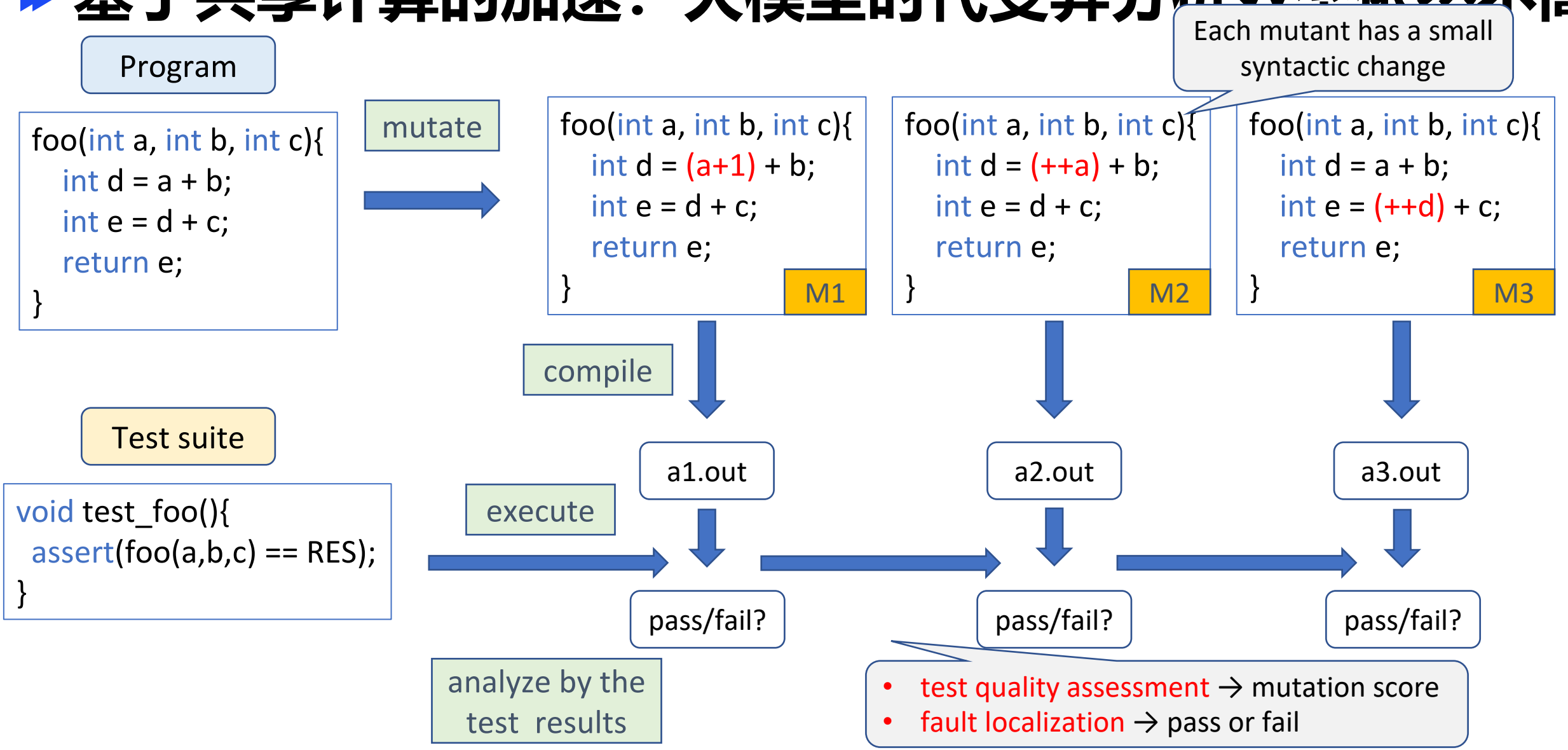
Settings	Spea. Coef.	*p*-value	Pear. Coef	*p*-value
QB-6a (Default) v.s. QB-3	0.9701	6.55E-05	0.9996	1.70E-10
QB-6a (Default) v.s. QB-3b	0.9762	3.31E-05	0.9984	1.00E-08
QB-6a (Default) v.s QB-9	0.9762	3.31E-05	0.9985	7.98E-09
QB-6a (Default) v.s D4J-6	0.9524	2.60E-04	0.9980	2.13E-08

基于大模型的变异生成：生成固定数量的变异

- 如果要求生成同样数量的变异
- 我们将所有工具生成的变异数限制为最少的一个（muBert）

Metric	GPT-3.5	CL-13b	LEAM	μBERT	Major	PIT
Comp. Rate	62.9%	74.2%	38.9%	18.5%	96.5%	–
Useless Mut. Ratio	11.0%	39.0%	1.4%	1.4%	0.0%	–
Real Bug Det.	0.892	0.735	0.675	0.550	0.833	0.437
Coupling Rate	0.133	0.105	0.132	0.132	0.128	0.024
Ochiai Coefficient	0.358	0.178	0.167	0.150	0.341	0.192

基于共享计算的加速：大模型时代变异分析效率依然不高



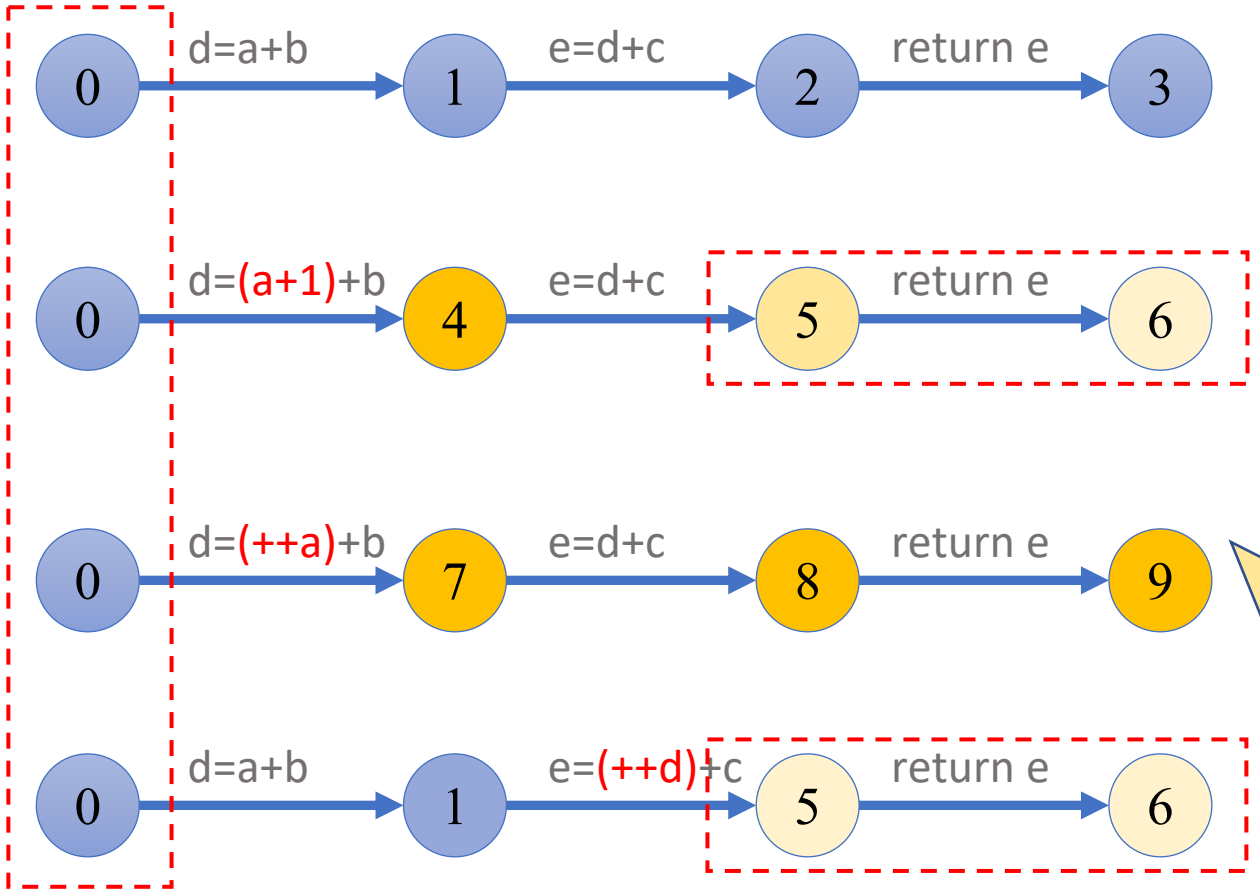
基于共享计算的加速：标准变异分析中的冗余计算

```
int d = a + b;
int e = d + c;
return e;
```

```
int d = (a+1) + b;
int e = d + c;
return e;
```
M1

```
int d = (++a) + b;
int e = d + c;
return e;
```
M2

```
int d = a + b;
int e = (++d) + c;
return e;
```
M3



Cycles represent states, the numbers inside are id

Arrows represent state transitions by executing the label statement

There are a lot of redundant computation. For example, the execution from the beginning to the state 0

基于共享计算的加速：分支流计算

- Split-stream execution: fork at the first mutated statement [King, Offutt, 1991][Tokumoto et al., 2016][Gopin… 2016]

```
int d = a + b;
int e = d + c;
return e;
```

```
int d = (a+1) + b;
int e = d + c;
return e;
```
M1

```
int d = (++a) + b;
int e = d + c;
return e;
```
M2

```
int d = a + b;
int e = (++d) + c;
return e;
```
M3

Replace mutated statements with SSE interpreter in compile time.

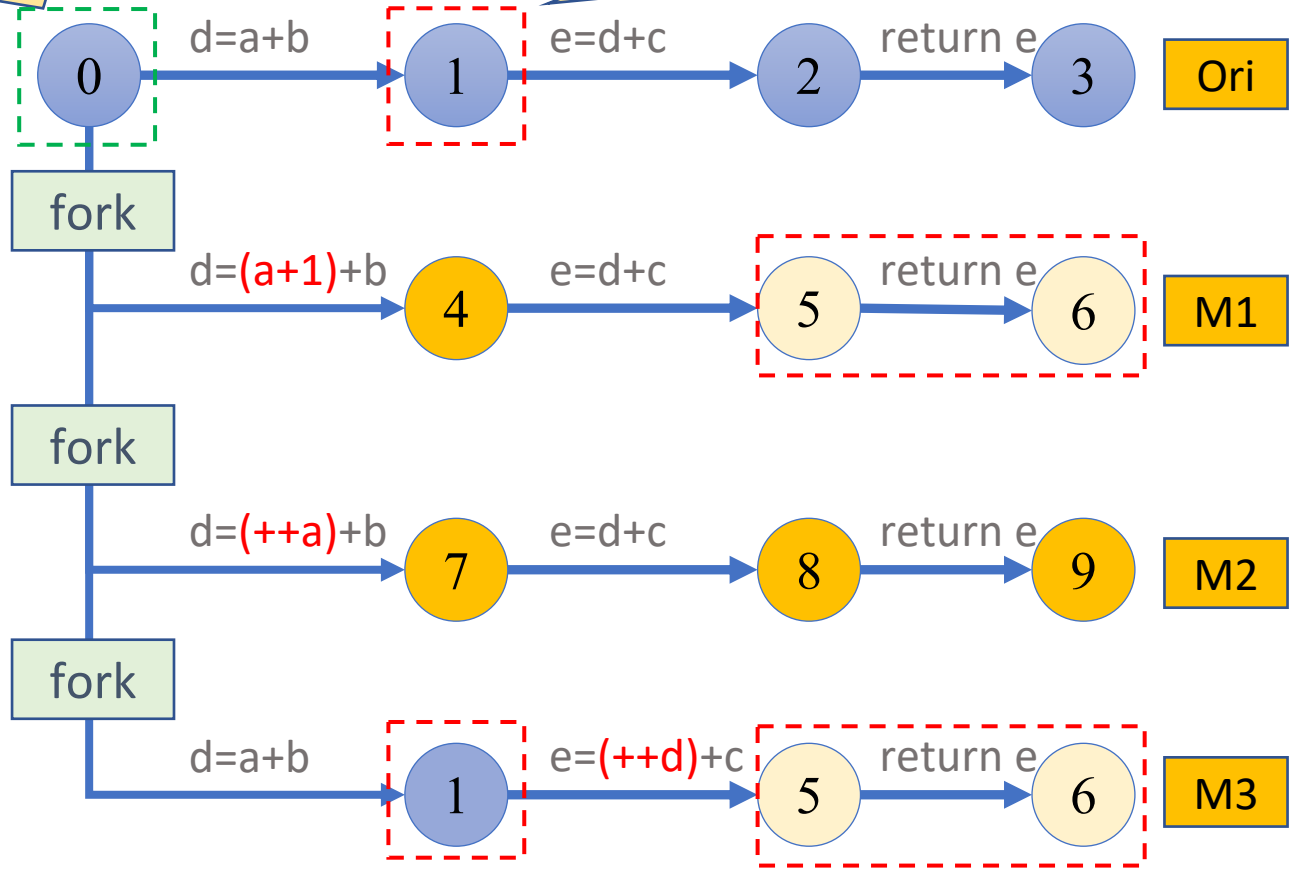
```
int d = interpret(a,b,ADD);
int e = interpret(d,c,ADD);
return e;
```

Start with a process carrying all mutants

The interpreter forks child-processes when encounters a mutated statement.

The computation before the state 0 is shared

The redundancy after the first line is still alive



基于共享计算的加速：AccMut [ISSTA-17]

- Equivalence Modulo States: Two mutants are *equivalent modulo the current state* if executing them leads to the same state from the current state [W

Instrument as SSE

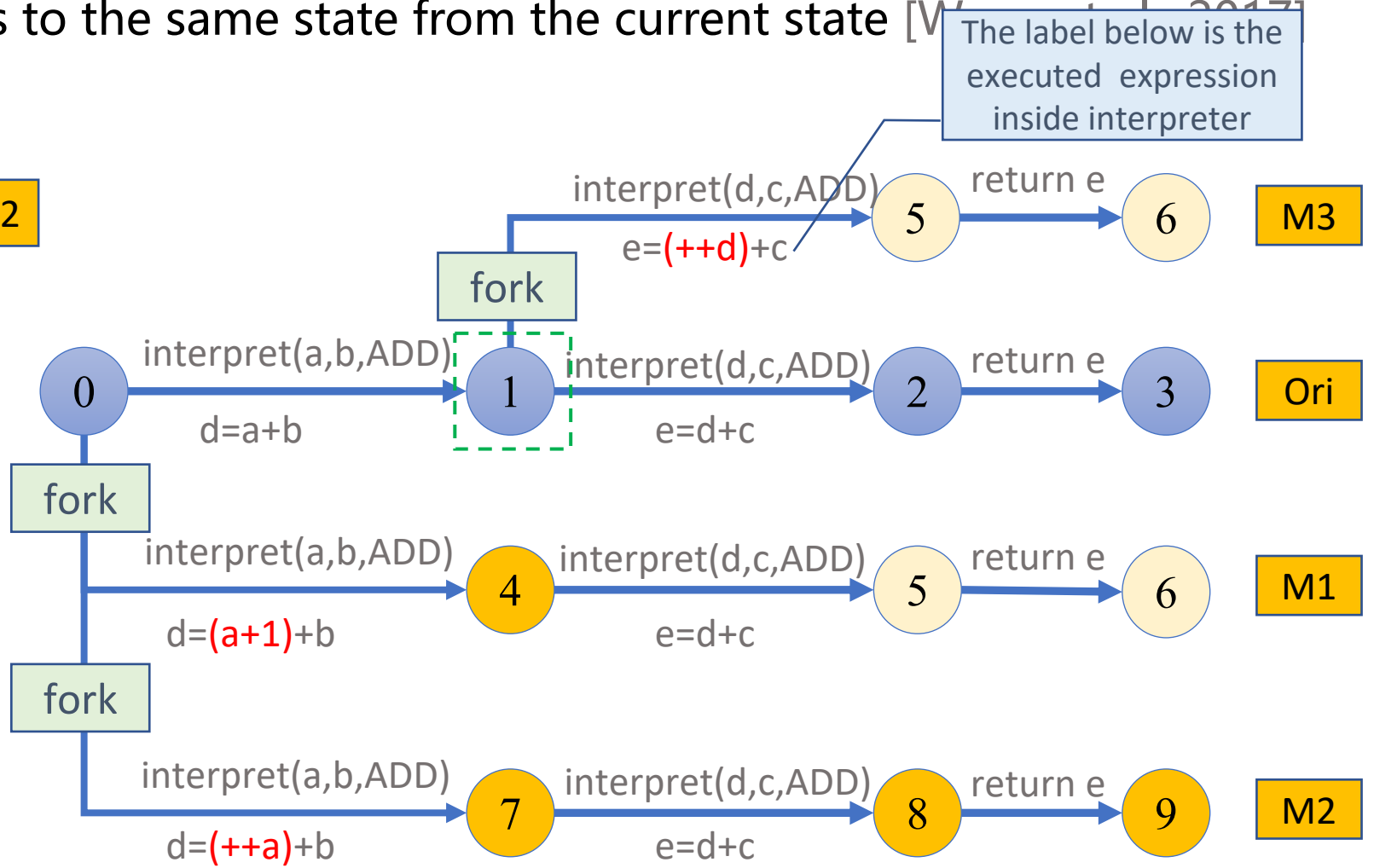
M1, M2

```
int d = interpret(a,b,ADD);
int e = interpret(d,c,ADD);
return e;
```

M3

The interpreter also carries all mutants

It clusters the mutants of one statement by EMS, and forks new process for each group



基于共享计算的加速：AccMut 依然存在大量冗余计算

1. AccMut is unable to merge more mutants

2. AccMut involves a lot of overhead

```
int d = a + b;
int e = d + c;
return e;
```

```
int d = (a+1) + b;
int e = d + c;
return e;
```
M1

```
int d = (++a) + b;
int e = d + c;
return e;
```
M2

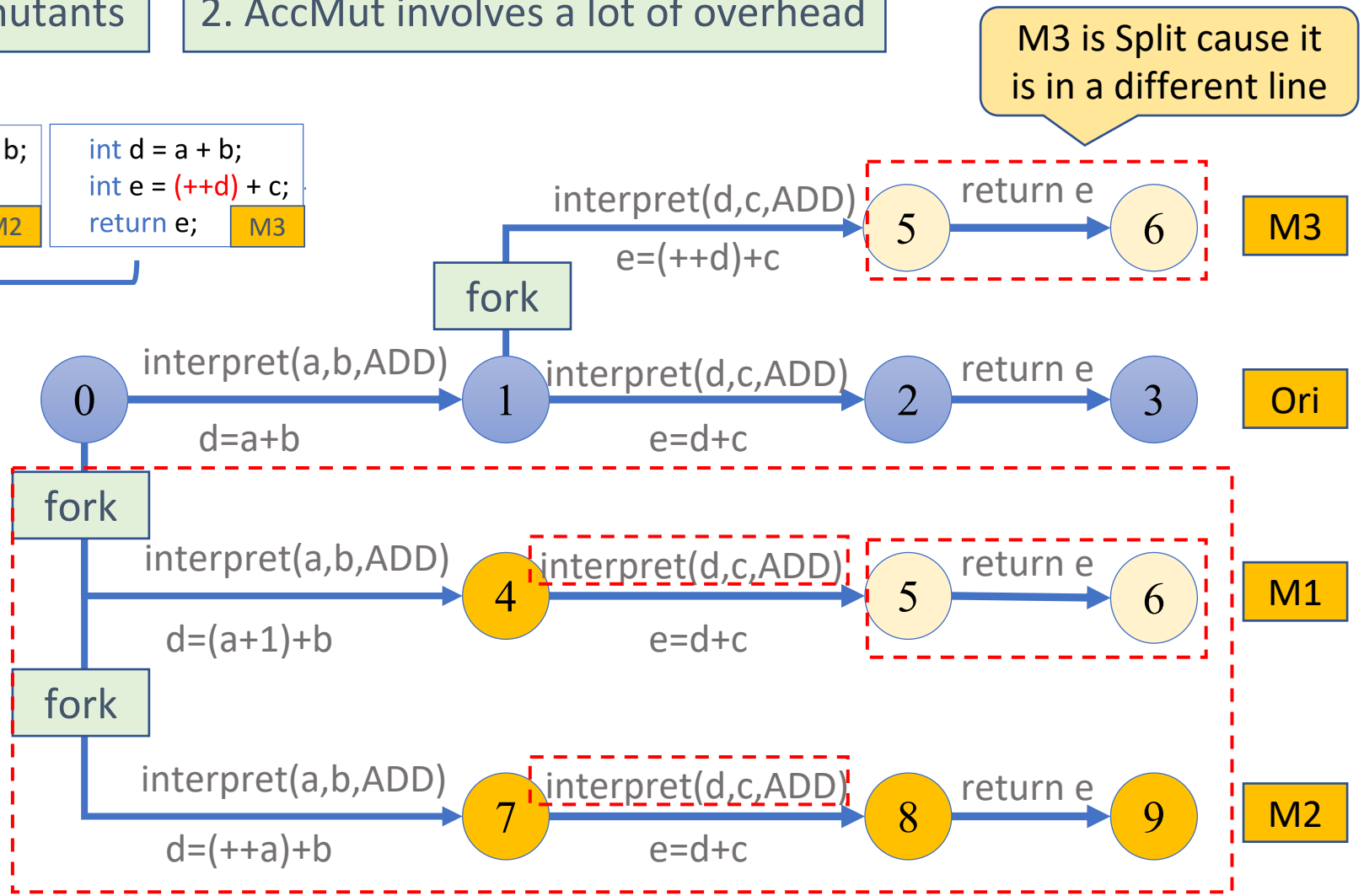
```
int d = a + b;
int e = (++d) + c;
return e;
```
M3

M1, M2

```
int d = interpret(a,b,ADD);
int e = interpret(d,c,ADD);
return e;
```
M3

The interpreter invocations in a child process is useless

M1 and M2 are split by the value of the variable *a*, however, which has no effect on the final result



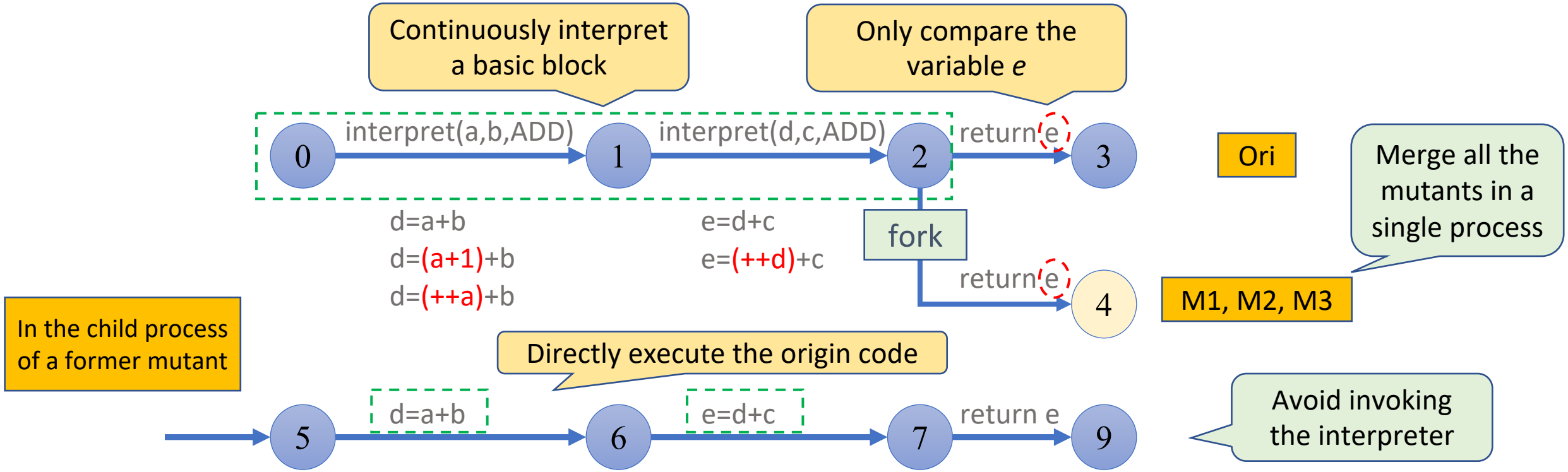
基于共享计算的加速：WinMut [ASE-21]

1. Enlarge analysis scope from a statement to a basic block
2. Only compare the variables that *may* affect the final result

→ Fewer processes

3. Avoid unnecessarily interpreting mutated statements

→ Smaller overheads



基于共享计算的加速：WinMut的实现

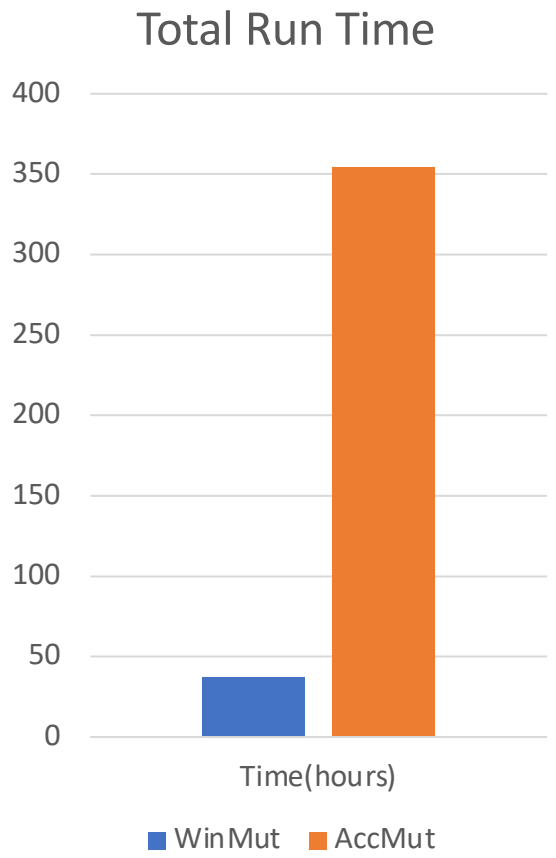
- 在 LLVM-IR 基础上实现，即它是在中间表示级别实施变异分析
- 变异算子涵盖当前所有中间表示级别分析的工具
- 程序员透明的、内存上的 IO 库

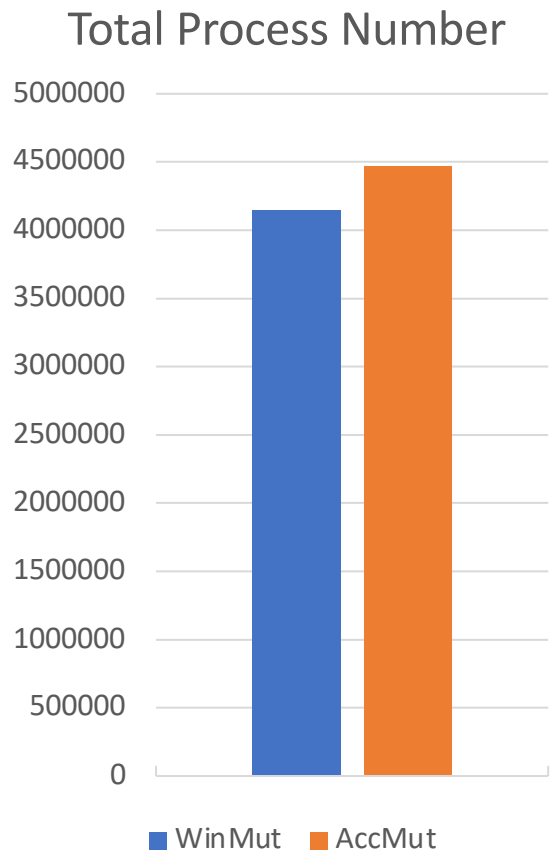
Name	Description	Example
AOR	Replace arithmetic operator	$a + b \rightarrow a - b$
LOR	Replace logic operator	$a \,\&\, b \rightarrow a \mid b$
ROR	Replace relational operator	$a == b \rightarrow a >= b$
LVR	Replace literal value	$T \rightarrow T + 1$
COR	Replace logical connector	$a \,\&\&\, b \rightarrow a \mid\mid b$
SOR	Replace shift operator	$a >> b \rightarrow a << b$
STDC	Delete a call	$f() \rightarrow nop$
STDS	Delete a store	$a = 5 \rightarrow nop$
UOI	Insert a unary operation	$b = a \rightarrow a++;\ b = a$
ROV	Replace the operation value	$f(a,b) \rightarrow f(b,a)$
ABV	Take absolute value	$f(a,b) \rightarrow f(abs(a),b)$

基于共享计算的加速：WinMut的实验对象程序

Name	Loc	# Exec Tests	# Mut	# BB	# Split	# Mut per Inst	# Mut per Split
Binutils-gas	299K	290	166,488	6,477	11,261	13.5	14.8
Coreutils	144K	287	400,150	11,532	19,628	20.4	7.2
Gmp	115K	30	613,595	10,774	23,225	22.3	26.4
Libsodium	45K	43	426,025	5,657	13,813	18.4	30.8
Lz4	13K	185	472,591	11,286	22,656	16.9	22.7
Pcre2	80K	33	266,399	6,900	11,722	16.7	22.7
Libpng	56K	9	282,831	8,527	15,394	15.0	18.4
Lua	16K	19	172,493	6,981	11,840	13.6	14.6
Grep	83K	207	217,399	8,406	16,144	12.9	13.5
Ffmpeg	1,032K	46	17,185,545	359,409	819,284	16.2	21.0
Total	1,584K	1,149	20,203,516	435,949	964,967	16.3	20.9

基于共享计算的加速：WinMut的实验效果





WinMut is faster than AccMut with an geometric average speed up of 5.57X.

WinMut uses 7.3% fewer processes than AccMut.

基于共享计算的加速：下一步的WinMut-Turbo

- 当前方法都是在尝试优化 Fork 之前的计算
- 我们通过程序分析，可以优化 Fork 之后的计算
- 如果我们确定 Fork 之后的子进程一定会影响测试结果，就及时终止
- WinMut-Turbo：在WinMut基础上，进一步带来30%的加速

PART 05
总结和展望

基于大模型的变异生成

- 大模型在生成变异上有独特的优势
- 最大的优势在于生成行为上接近真实bug的变异
- 目前仍有很大提升空间
  - 使用prompt-engineering
  - 使用LLM微调
  - 探索多智能体在变异生成的应用

```
    // Mutation-1
-     Size2D size = this.rightBlock.arrange(g2, c4);
+     Size2D size = this.leftBlock.arrange(g2, c4);
    // Mutation-2
-     else {
-       g2.setStroke(getItemOutlineStroke(row, column, selected));
-     }
+     else if (seriesCount == 1) {
+       g2.setStroke(getItemOutlineStroke(row, column, selected));
+     }
```

面向大模型的复杂变异加速

- 虽然相比于当前最新方法（SSE），已经累计加速 50x
- 当前的加速方法有一定局限性
- 仅能优化限定代码区间的简单变异
- 面对大模型生成的复杂变异，如何进一步优化是未来工作重点

THANKS