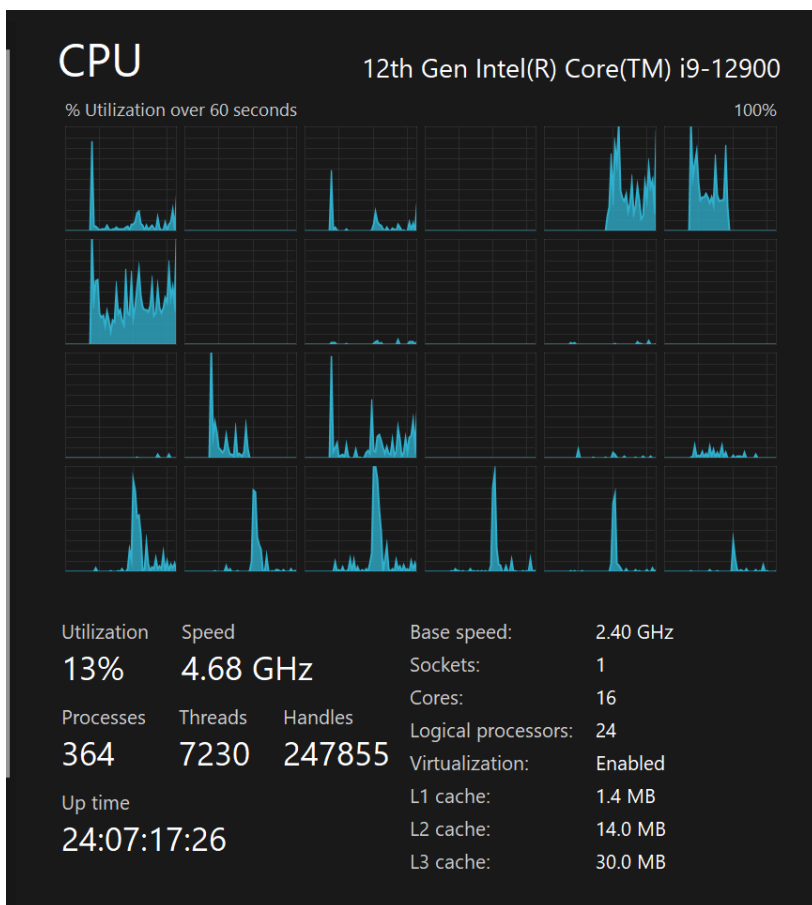


运行时监控

运行时监控

硬件监控

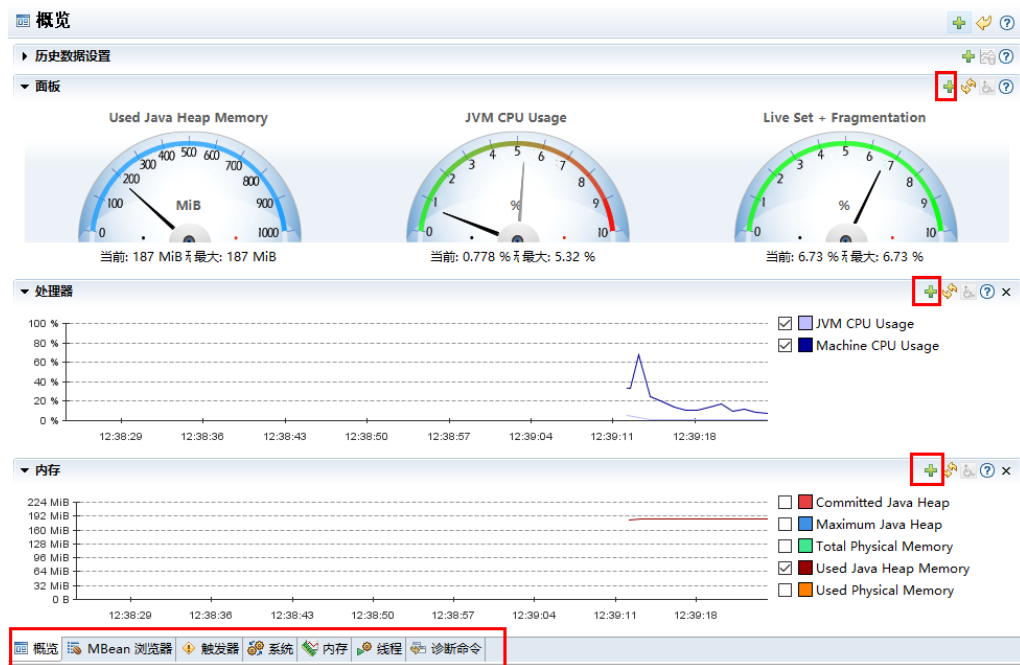


软件监控



运行时监控

- 为什么需要运行时监控?
- 程序Bug很难完全消除
 - 测试中的漏报问题普遍存在
 - 状态空间爆炸目前无解
- 程序行为的合法检查
 - 资源监控、运行调度
 - 安全等要求
- ...



测试阶段、调试阶段、部署运行阶段

运行时监控

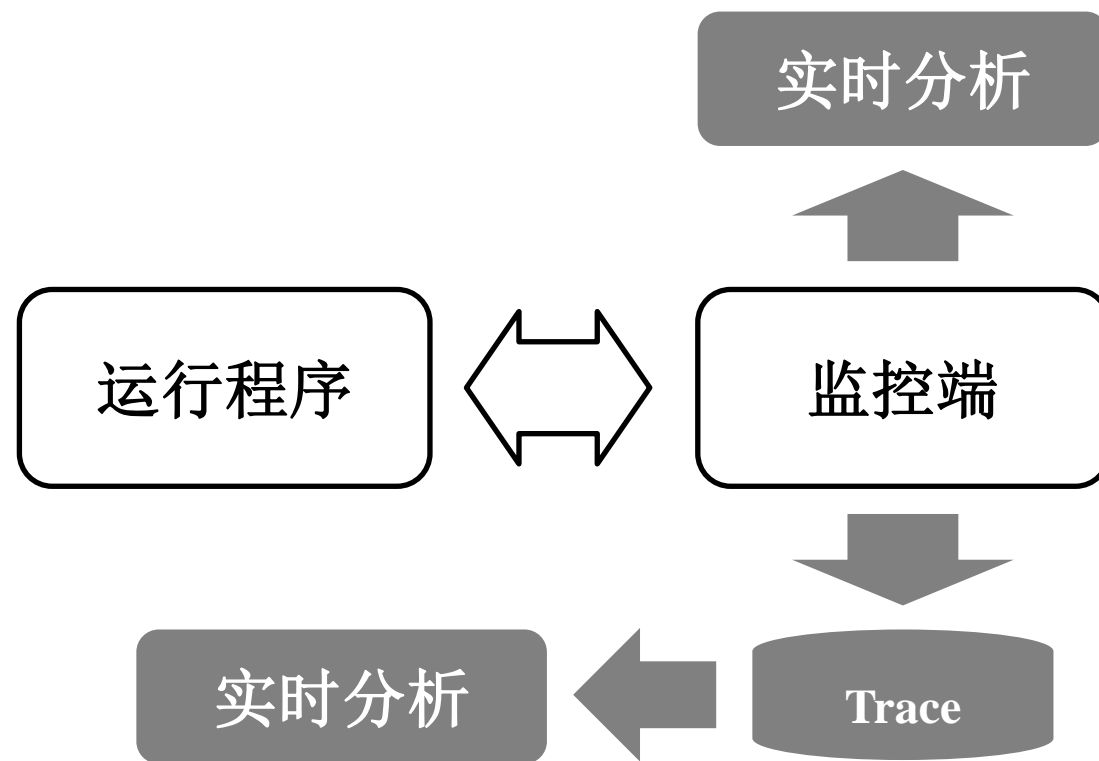
- **运行时监控/验证/检查 (Runtime Monitoring/Verification/Checking)**
- Runtime Monitoring: a lightweight and **dynamic verification** technique that involves observing the internal operations of a software system and/or its interactions with other external entities, with the aim of determining whether the system **satisfies** or **violates** a correctness specification.

Cassar, Ian & Francalanza, Adrian & Aceto, Luca & Ingólfssdóttir, Anna. (2017). A Survey of Runtime Monitoring Instrumentation Techniques. Electronic Proceedings in Theoretical Computer Science. 254. 15-28. 10.4204/EPTCS.254.2.
- Runtime Verification: a computing system **analysis** and **execution based** on extracting information from a running system and using it to detect and possibly react to observed behaviors **satisfying** or **violating** certain properties.

运行时监控

运行时/在线监控 vs 离线监控 (Runtime/Online vs Offline)

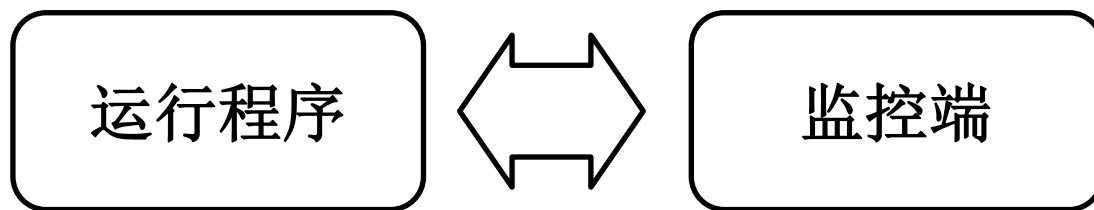
- 均需要记录程序运行 (trace)
- 离线监控
 - 需要完整 (连续) trace分析才可以的场景
 - 运行时仅记录trace, 对程序运行影响较小
- 在线监控
 - 仅需部分 (离散) trace分析的场景
 - 运行时无需显式记录trace,
 - 可能需要主动干扰程序运行



运行时监控

技术手段

- Print
- R/W File
- Debug
- Hook
- 替换
- 软硬件接口
- ...



大量依赖于插桩、系统软硬件支持。

运行时监控

关键点

- 是否轻量级（相对）
- 是否入侵原程序的运行
- 是否有状态

轻量级

- 对原程序运行可能的干扰（被动干扰）

原因

- 过多监控会导致原程序中某些特性无法体现（Heisenbug (海森堡bug)）

举例

- 定时任务、并发行为、破坏原有内存布局

运行时监控

关键点

- 是否轻量级（相对）
- 是否入侵原程序的运行
- 是否有状态

入侵

- 是否需要控制原程序的运行（主动干扰）

原因

- 很多Bug依赖于特定的运行环境（包括程序自身的特定行为），监控中需要主动构造此类环境，需要干扰程序的默认运行

举例

- 循环中特定的i值、流量中特定的字符串、特定函数的调用、并发中特定的线程交替等

运行时监控

关键点

- 是否轻量级（相对）
- 是否入侵原程序的运行
- 是否有状态

状态

- 是否需要依赖于历史运行记录来判定当前的监控行为

原因

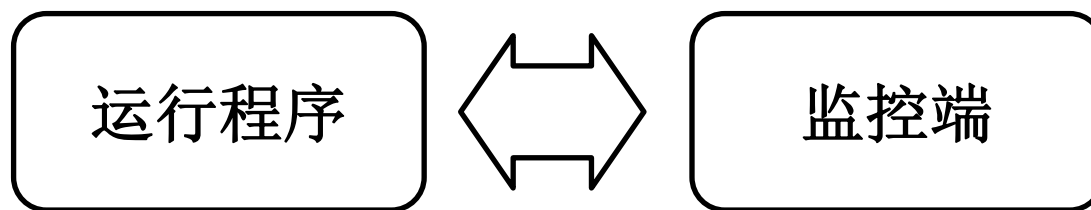
- 很过Bug的发生依赖于历史行为

举例

- 软件漏洞（UAF）等需要多次操作才可触发

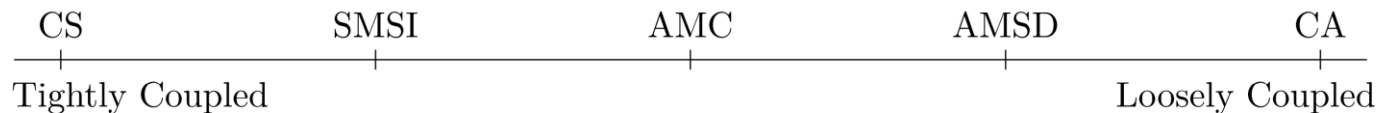
运行时监控

常见监控方法



已有的研究中，根据被监控程序和监控端的耦合关系

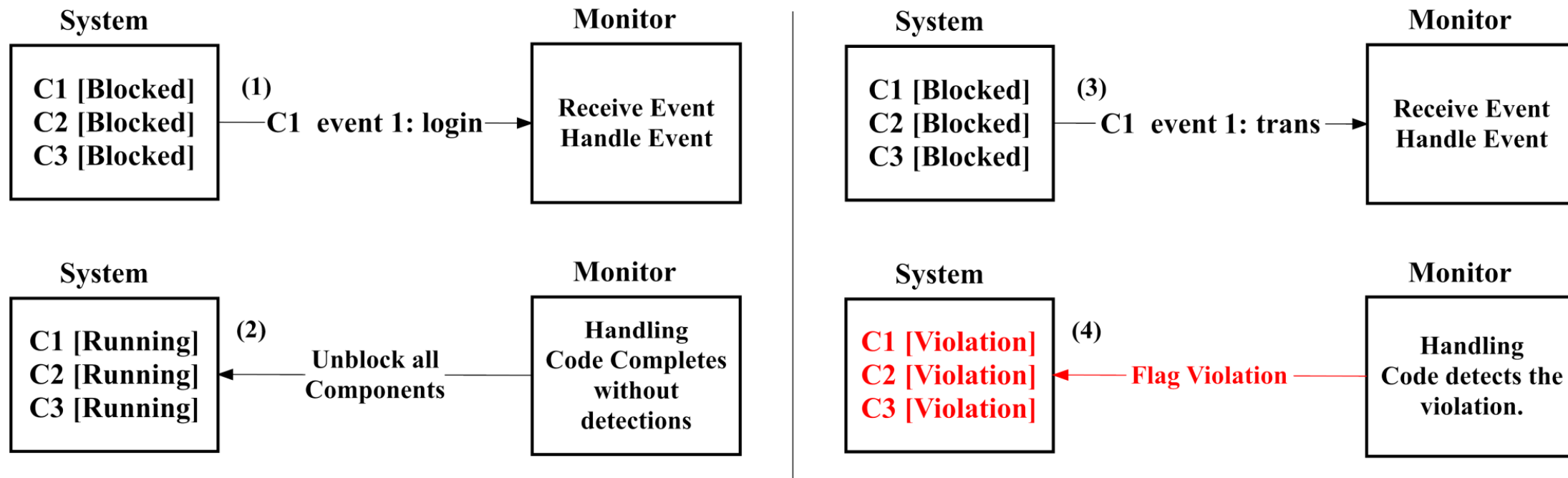
- 全局同步监控（CS: Completely Synchronous Monitoring）
- 同步分析同步监控（SMSI: Synchronous Monitoring with Synchronous Instrumentation）
- 带检查点的异步监控（AMS: Asynchronous Monitoring with Checkpoints）
- 同步检测异步监控（AMSD: Asynchronous Monitoring with Synchronous Detection）
- 全局异步监控（CA: Completely Asynchronous Monitoring）



运行时监控

(1) 完全同步监控 (CS: Completely Synchronous Monitoring)

“A user cannot make a transaction before a login.”

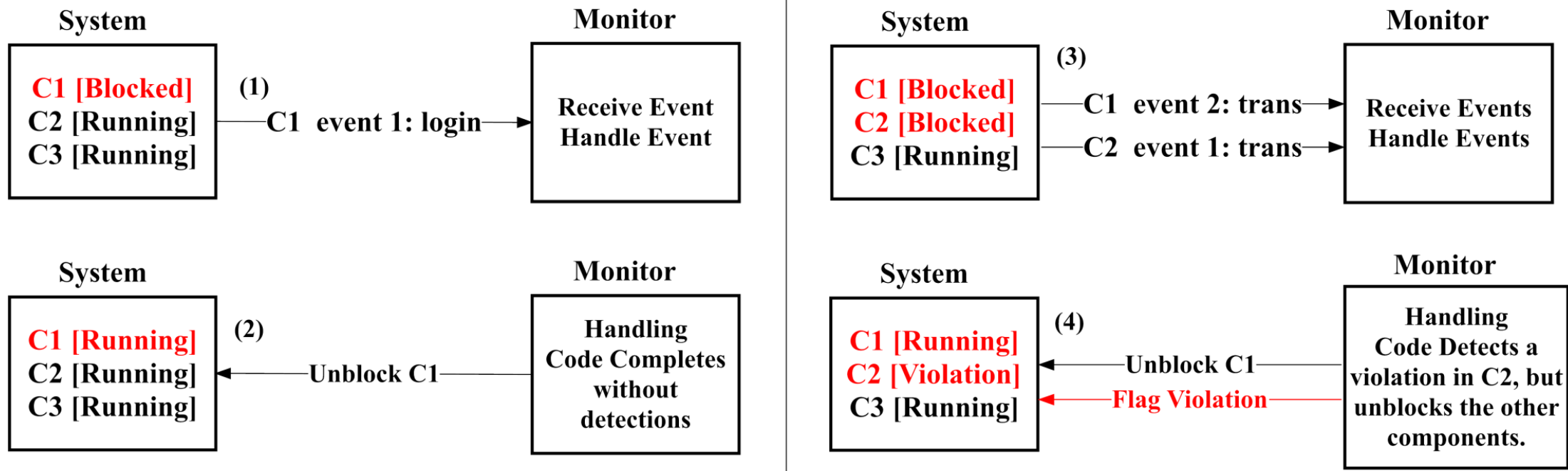


程序（包括线程等组件）和监控端均完全同步

运行时监控

(2) 同步分析同步监控

(SMSI: Synchronous Monitoring with Synchronous Instrumentation)

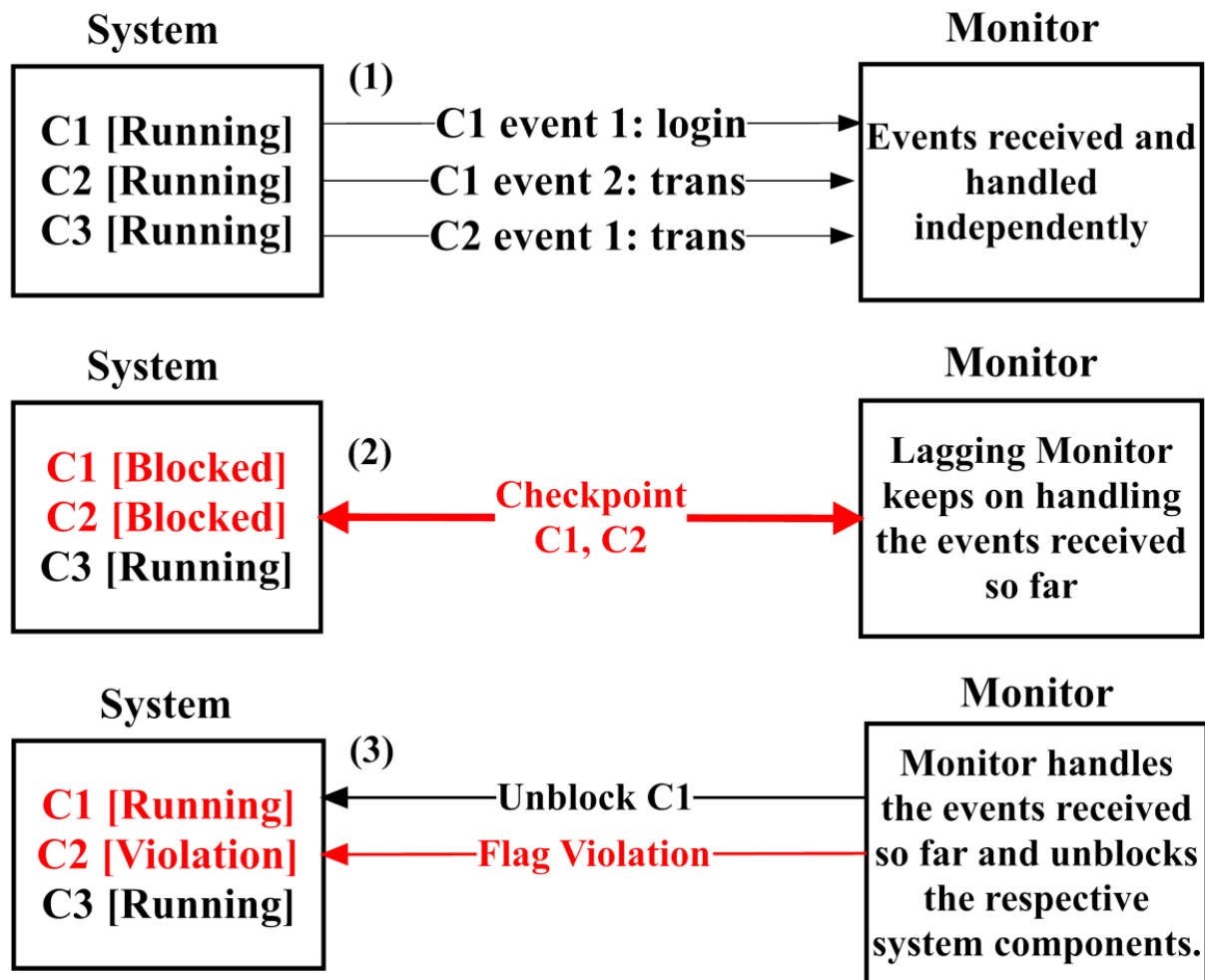


程序中，单个组件和监控端均完全同步

运行时监控

(3) 带检查点的异步监控 (AMS: Asynchronous Monitoring with Checkpoints)

对程序或其组件设定Checkpoint, 仅在Checkpoint处同步监控, 其余异步监控。



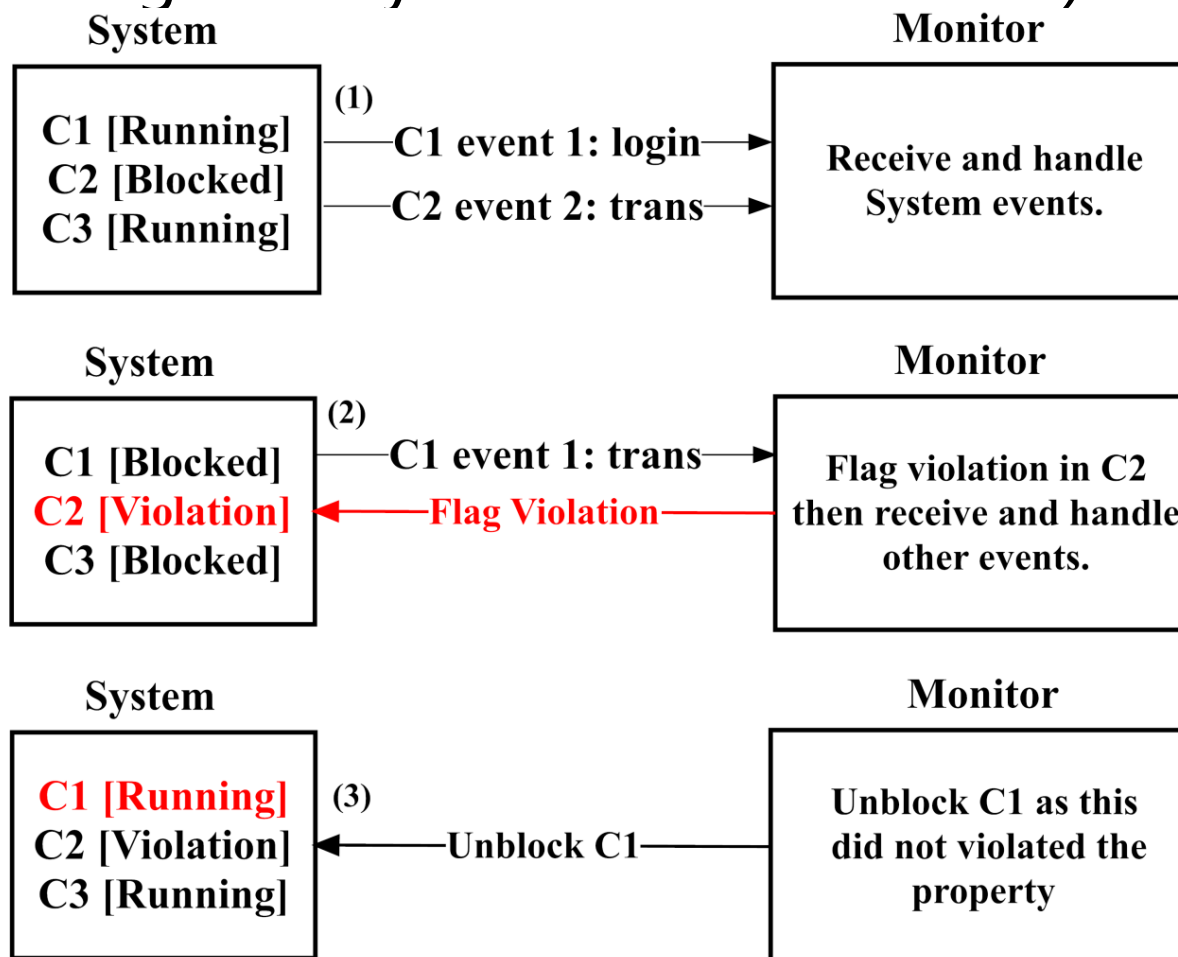
运行时监控

(4) 同步检测异步监控

(AMSD: Asynchronous Monitoring with Synchronous Detection)

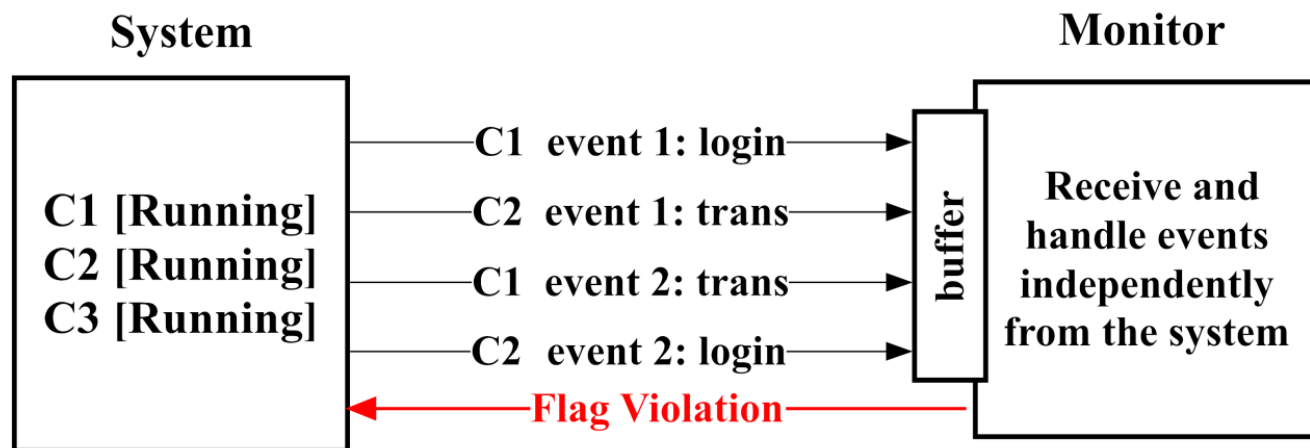
对程序（组件等）拆分：

- 异步监控部分（非关键）组件
- 同步监控部分（关键）组件



运行时监控

(5) 全局异步监控 (CA: Completely Asynchronous Monitoring)



程序（包括线程等组件）和监控端均完全异步

运行时监控

- 相关工具

Tool	Online				Offline	
	CS	SMSI	AMC	AMSD	CA	
DECENTMON	✓					
LOLA	✓					✓
JEAGLE		✓				
DB/Temporal-Rover		✓			✓	✓
Java-MOP		✓			✓	✓
LARVA toolkit		✓			✓	✓
detectEr toolkit		✓	✓	✓	✓	
EXAGO						✓

运行时监控——举例

- 需求
 - 代码覆盖率的计算
- 设计
 - 插桩选择（算法、级别、阶段，运行时数据）
 - 数据收集（离线or在线、重复数据）
 - 数据分析
 - 数据展示

第二次大作业部分内容

代码覆盖率的计算

- 代码覆盖率
 - 运行时所覆盖的代码与程序中代码总量的比例
 - 在测试中达到越高的代码覆盖率就越有机会发现更多漏洞
 - 很多覆盖导向的测试工具（例如各类fuzzer）
 - 因此该如何衡量与计算代码覆盖率？
 - 函数覆盖率（function coverage）
 - 语句覆盖率（statement coverage）
 - 分支覆盖率（branch coverage）
 - 条件覆盖率（condition coverage）
 - 基本块覆盖率（bbl coverage）
 - 路径覆盖率 ...

代码覆盖率的计算

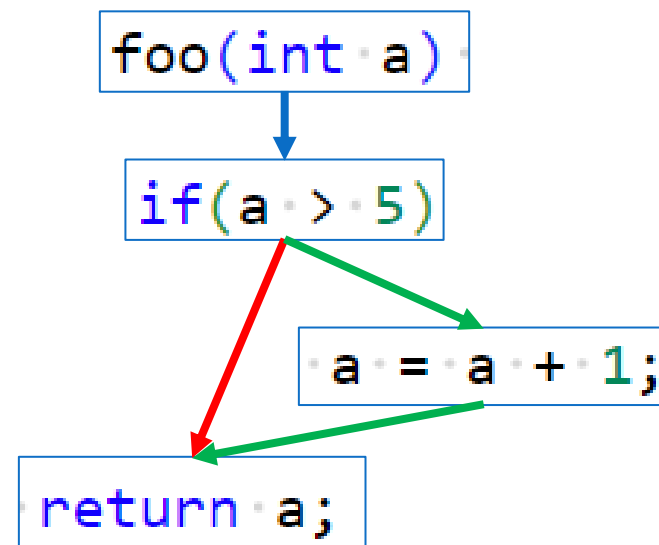
- 语句覆盖率
 - 执行到的语句数量与总的可执行语句数量的比例
 - 不包括头文件声明，代码注释，空行等语句
 - 计算时通常也不考虑else, case以及一些括号行
 - 不考虑判定条件，只考虑执行到语句的比例
 - 一种较弱的覆盖率计算方法

代码覆盖率的计算

- 分支覆盖率

- 执行到的分支数量与总分支数量的比例
- 考虑每个分支判定条件为true或false的情况，即使没有else分支
- 但不会考虑判定条件的子条件为true和false的情况
- 下图中，若 $a = 6$ ，语句覆盖率为100%，而分支覆盖率为50%


```
int foo(int a) {  
    ... if(a > 5) {  
        ... a = a + 1;  
    ... }  
    ... return a;  
}
```



代码覆盖率的计算

- 条件覆盖率
 - 执行到的分支判定中每个子表达式的取值占所有可能取值的比例
 - 注意 或 操作时的短路问题

```
int foo(int a1, int a2) {  
    ... if(a1 || a2) {  
        .....  
    } else {  
        .....  
    }  
}
```



a1	T	F	F
a2	?	T	F

代码覆盖率的计算

- 实例
 - 哪些测试用例能使右图代码覆盖率达到100%?
 - 针对下面的测试用例计算其覆盖率
 - Testcase1
 - a = 6, b = 5
 - Testcase2
 - a = 6, b = 5
 - a = 3, b = 11
 - a = 11, b = 11
 - a = 11, b = 13

```
int foo(int a, int b) {  
    if (a < 10 || b < 10) {  
        return f1(a, b);  
    } else {  
        return f2(b, a);  
    }  
}
```

```
int f1(int a, int b) {  
    if (a * a > 25) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

```
int f2(int a, int b) {  
    if (a * a < 144) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

代码覆盖率的计算

- Testcase1
 - $a = 6, b = 5$
 - 语句覆盖率: 4/9
 - 分支覆盖率: 1/3
 - 条件覆盖率: 1/4

```
int foo(int a, int b) {  
    if (a < 10 || b < 10) {  
        return f1(a, b);  
    } else {  
        return f2(b, a);  
    }  
}
```

```
int f1(int a, int b) {  
    if (a * a > 25) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

```
int f2(int a, int b) {  
    if (a * a < 144) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

代码覆盖率的计算

- Testcase2
 - a = 6, b = 5
 - a = 3, b = 11
 - a = 11, b = 11
 - a = 11, b = 13
 - 语句覆盖率: 100%
 - 分支覆盖率: 100%
 - 条件覆盖率: 7/8
 - 如何修改测试用例使得条件覆盖率为100%?

```
int foo(int a, int b) {  
    if (a < 10 || b < 10) {  
        return f1(a, b);  
    } else {  
        return f2(b, a);  
    }  
}
```

```
int f1(int a, int b) {  
    if (a * a > 25) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

```
int f2(int a, int b) {  
    if (a * a < 144) {  
        return a;  
    } else {  
        return b;  
    }  
}
```


基于插桩的自动化覆盖率计算

- 如何自动化计算代码覆盖率?
 - 插桩收集函数/语句/分支/判定条件的执行信息
 - 使用LLVM Pass对IR插桩
 - IR由基本块构成，一般来说基本块内的指令执行次数相同
 - 基本块通常结尾为跳转指令，跳转的目的地是另一个基本块
 - 可以对IR中的函数，基本块，指令等进行遍历，插入全局变量以及计数代码来统计执行次数等信息
 - getDebugLoc函数可以获取每条指令在module中的行号等信息
 - 根据收集到的执行信息计算代码覆盖率

```
int foo(int a, int b) {  
    ... if (a < 10 || b < 10) {  
        ... return f1(a, b);  
    } else {  
        ... return f2(b, a);  
    }  
}
```

```
define i32 @_Z3fooi(i32, i32) #0 { 函数  
    ...  
    ...  
    %7 = icmp slt i32 %6, 10 判断  
    br i1 %7, label %11, label %8  
; <label>:8:  
    ...  
    %10 = icmp slt i32 %9, 10 判断  
    br i1 %10, label %11, label %15  
; <label>:11:  
    ...  
    %14 = call i32 @_Z2f1ii(i32 %12, i32 %13)  
    ...  
    br label %19  
; <label>:15: 基本块  
    ...  
    %18 = call i32 @_Z2f2ii(i32 %16, i32 %17)  
    ...  
    br label %19  
; <label>:19: 基本块  
    %20 = load i32, i32* %3, align 4  
    ret i32 %20  
}
```

基于插桩的覆盖率计算

- LLVM Pass
 - 对LLVM IR处理的框架
 - 所有的LLVM Pass都是Pass类的子类
 - 根据需求的不同可以选择继承不同的Pass类来进行开发
 - 基于这些类可以方便地对IR中的函数、基本块等进行遍历和处理
 - ModulePass
 - CallGraphSCCPass
 - FunctionPass
 - LoopPass
 - RegionPass
 - BasicBlockPass

基于插桩的覆盖率计算

- 基于FunctionPass的插桩
 - 创建一个新类继承自FunctionPass
 - 重载FunctionPass中的runOnFunction函数
 - runOnFunction函数中的代码对每个函数都会执行一次
 - 在runOnFunction函数中编写桩代码
 - Pass中提供了各种构建和插入各类指令的方法如IRBuilder等
 - 注册Pass

Q & A

问答

