北京大学
PEKING UNIVERSITY

基于AI的数字系统设计

第9讲：AI辅助数字系统验证

主讲：陶耀宇

2026年秋季

北京大学版权所有

北京大学
PEKING UNIVERSITY

AI驱动的功能验证

测试TB、激励与UVM框架

北京大学版权所有

功能验证：芯片项目的关键路径

- 设计复杂度激增，首次流片成功率跌至二十年来最低

14%

首次流片成功率

IC/ASIC项目，2024年，二十年来最低

75%

项目进度落后于计划

历史水平约为三分之二

~30%

历史上的首次流片成功率

作为对比基线

复杂度从何而来

- **异构SoC / AI加速器**：多核、NoC、多级存储，[illegible]成倍增加
- **配置与协议空间爆炸**：参数组合呈指数增长，人工难以穷举
- **规格频繁变更**：大量重流片源于设计错误与规格变化
- **人工搭环境、写用例**：周期长，corner case容易遗漏

数据来源：Siemens EDA & Wilson Research Group, 2024 IC/ASIC Functional Verification Trend Report；SemiEngineering《First-Time Silicon Success Plummets》

思想自由 兼容并包

北京大学版权所有

验证闭环与AI切入点

- 验证是“计划—环境—激励—检查—覆盖—回归”的闭环，AI可在每一环提速



核心思想：AI不替代验证方法学，而是嵌入每个环节，缩短“发现问题 → 定位 → 修复 → 收敛”的周期

为什么需要验证方法学

- 从手写Verilog Testbench到统一标准UVM：可复用、可扩展、可互操作



Verilog
IEEE 1364：用task手写定向测试，难以复用

SystemVerilog
IEEE 1800：OOP、约束随机、断言、覆盖率

VMM
Synopsys与ARM推出的验证方法学

OVM
Cadence与Mentor联合开源方法学

UVM 1.0
Accellera发布，统一三大EDA厂商

UVM 1.2
广泛使用的Accellera版本

IEEE 1800.2
UVM成为IEEE标准

可复用
VIP / agent跨项目、跨层级复用

可扩展
factory+配置机制，不改原代码即可扩展

可互操作
统一类库，跨仿真器、跨厂商运行

思想包袱：兼容并包

UVM依赖的SystemVerilog验证特性

- UVM是建立在SystemVerilog语言特性之上的标准类库与方法学

类与面向对象

继承、多态、虚方法，让组件可扩展、可替换

```
class my_dr
  extends uvm_driver #(txn);
```

约束随机

rand / randc + constraint，由求解器产生合法激励

```
assert(tr.randomize()
       with {len < 8;});
```

interface与virtual interface

把静态的RTL信号交给动态的类对象访问

```
virtual fifo_if vif;
```

SVA断言

描述时序属性，仿真中实时检查协议

```
ass t property (@(posedge clk)
r  |-> ##[1:3] ack);
```

covergroup功能覆盖

定义功能点与交叉组合，量化验证完备度

```
cove group cg; coverpoint op;
endg oup
```

UVM = 类库 + 方法学

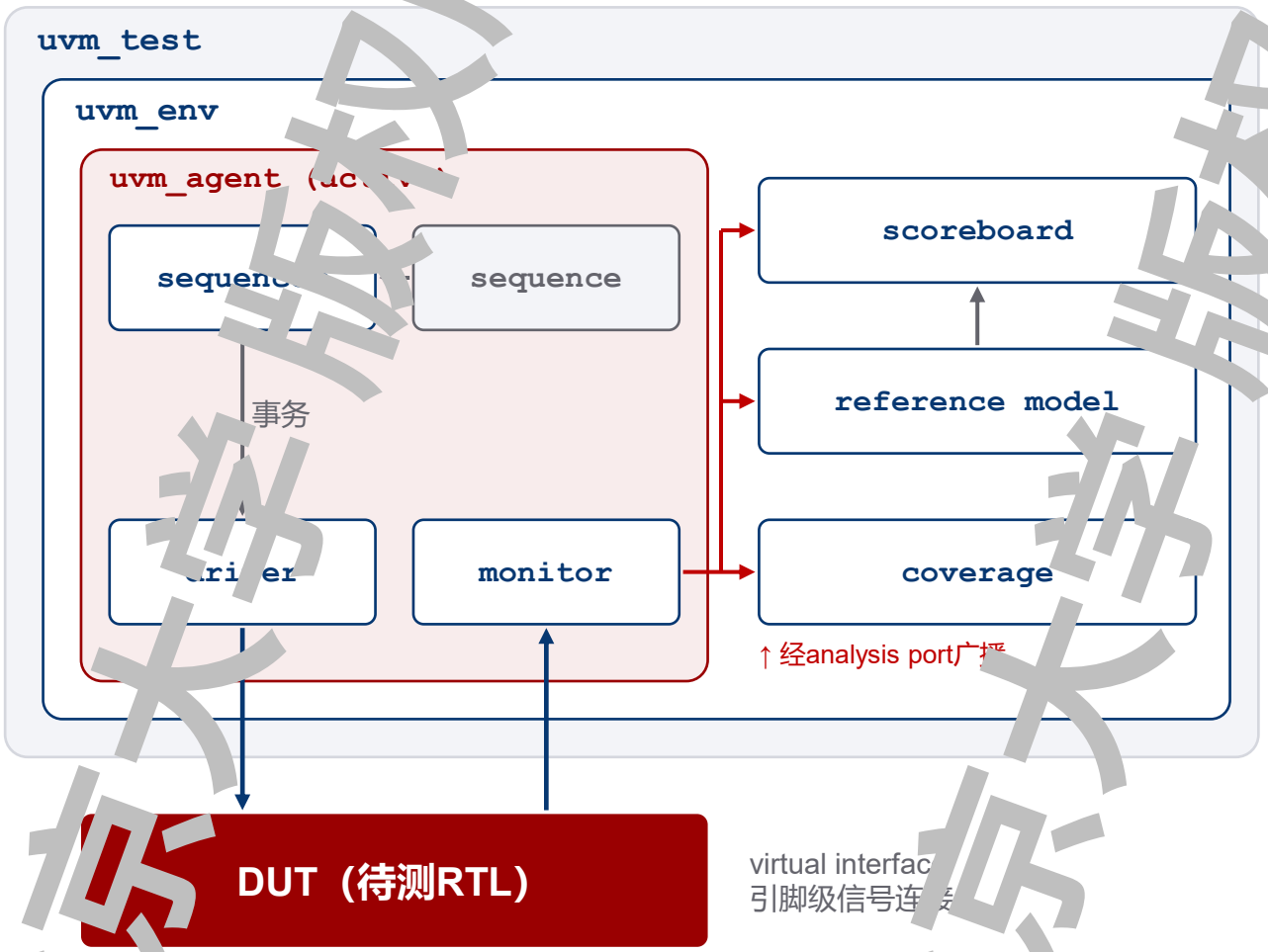
把上述语言特性封装为标准基类、宏与机制：

phase · objection · factory · config_db · TLM · RAL

思想：包容并包

UVM Testbench分层架构

- UVM以分层、可复用的组件组织验证环境，是AI生成代码的“目标骨架”



test层

选择场景：配置环境、启动sequence、设置override

env层

组装agent、scoreboard、coverage，完成连接

agent层

封装一个接口协议：sequencer + driver + monitor

信号层

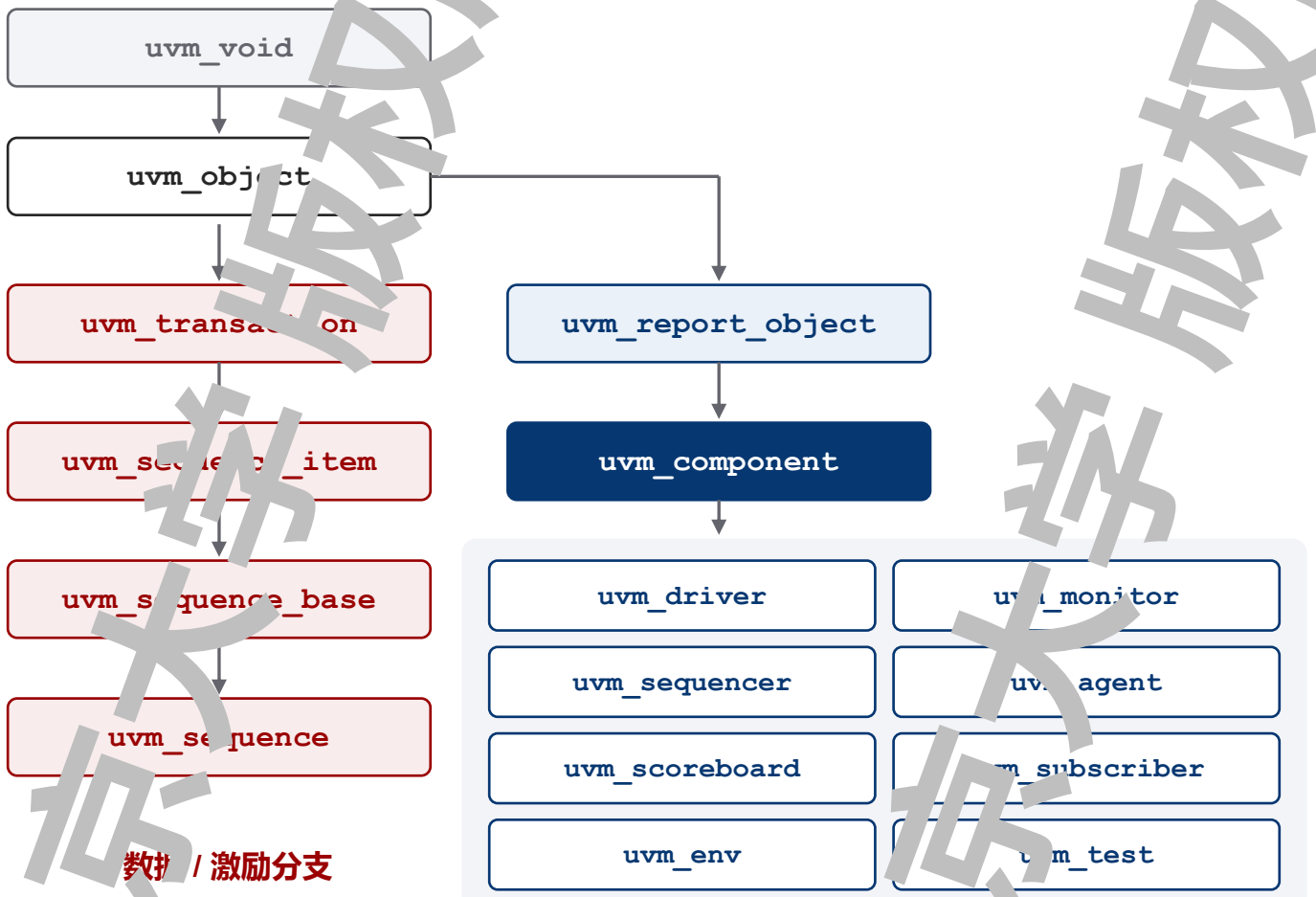
driver / monitor经virtual interface与DUT交互

思想包袱 兼容并包

北京大学版权所有

UVM类库层次：object与component

- 数据与激励派生自uvm_object，验证平台结构派生自uvm_component



维度	uvm_object	uvm_component
构造函数	new(name)	new(name, parent)
注册宏	`uvm_object_utils	`uvm_component_utils
层次结构	无父节点，可自由传递	有parent，构成组件树
生命周期	随时创建、随时丢弃	build时创建，贯穿仿真
phase	不参与	参与全部phase
典型	事务、sequence、配置对象	driver、env、test

思想包袱 兼容并包

UVM核心组件职责一览

- 每类组件职责单一、接口固定：这是可复用，也是可自动生成的基础

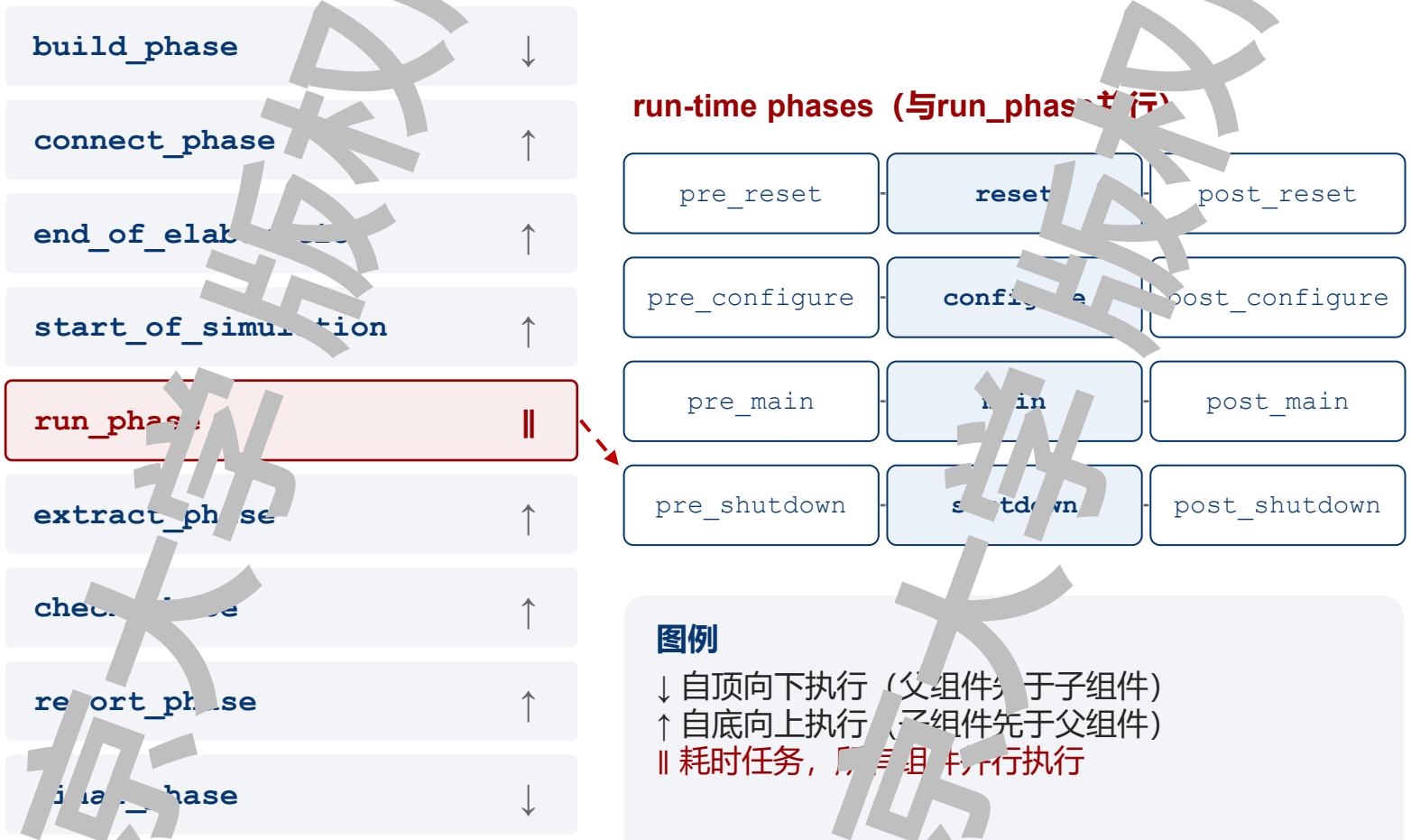
组件	职责	关键方法 / 端口
uvm_sequence_it	事务对象：一次传输的数据与控制字段	do_copy / do_compare / convertz ng
uvm_sequence	产生事务序列，描述一个测试场景	body()、start_item / finish_i e
uvm_sequencer	仲裁多个sequence，把事务转交driver	seq_item_export
uvm_driver	事务 → 引脚级时序，驱动DUT	seq_item_port.get_next_item / item_done
uvm_mon or	采样引脚 → 还原为事务并广播	uvm_analysis_port::writ
uvm_agen	封装一个接口：active含driver，passiv 只监测	is_active (UVM_ACTIVE / UVM_PASSIVE)
uvm_ co eb d	与参考模型比对，判定正确性	uvm_analysis_imp + w it ()
uvm_s osc iber	订阅事务，采样功能覆盖率	write()中调用covergrou .sa le()
uv _e v	组装agent / scoreboard / c v rage	build_phase / con c_ h e
uvm_ est	选择配置与sequence，启动 景	run_test()、+UVM_TE NAM

思想包袱 兼容并包

北京大学版权所有

Phase机制详解

- 统一的执行时序：构建 → 连接 → 运行 → 收尾，所有组件步调一致



图例

↓ 自顶向下执行（父组件先于子组件）
↑ 自底向上执行（子组件先于父组件）
‖ 耗时任务，[illegible]并行执行

执行顺序

build_phase自顶向下：父组件build时创建子组件；connect等其余函数ph[illegible]自底向上

run_phase并行

所有组件的run_phase[illegible]的启动；每个run-time phase须全部组[illegible]进入下一个

使用建议

多数项目只用run_phas[illegible] + objection；子phase适合复[illegible]、重配置等多阶段场景

Objection机制与仿真结束

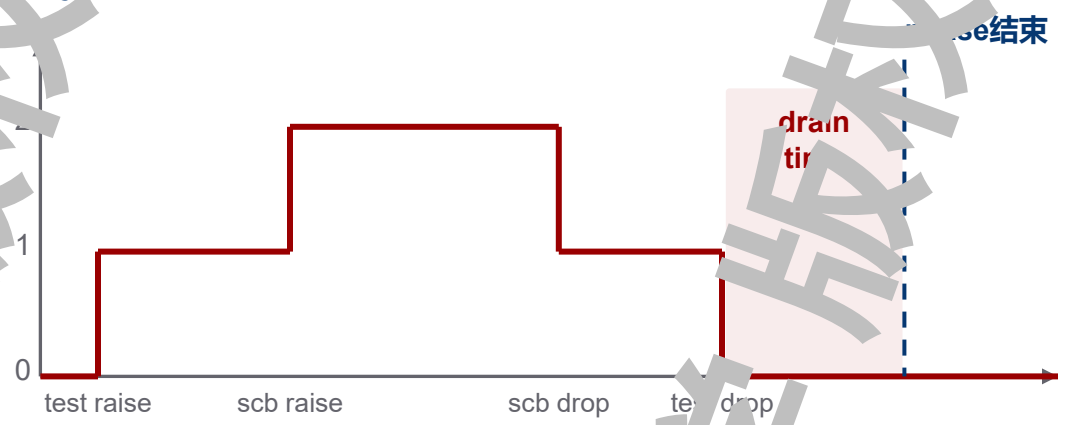
- run_phase何时结束？由各组件“举手反对”的计数决定

raise / drop / drain time

```
class base_test extends uvm_test;
  `uvm_component_utils(base_test)
  fifo_env env;
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction

  task run_phase(uvm_phase phase);
    fifo_seq seq = fifo_seq::type_id::create("seq");
    phase.raise_objection(this, "main seq start");
    seq.start(env.agt.sqr);
    // 计数归零后再等 200ns，让流水线排空
    phase.get_objection().set_drain_time(this, 200ns);
    phase.drop_objection(this, "main seq done");
  endtask
endclass
```

objection计数随时间变化（示意）



drain time	计数归零后延时结束，等待最后的响应被比对
忘记raise	run_phase立即结束，仿真在0时刻退出
忘记drop	仿真挂起直到全局超时（set_timeout）

思想自由 兼容并包

事务与约束随机

- rand / randc、约束、分布与优先级：定义激励的合法空间与概率

SystemVerilog / UVM

```
class bus_txn extends uvm_sequence_item;
  rand  bit [31:0]  addr;
  rand  bit [31:0]  data;
  rand  bit         wr;
  rand  int unsigned len;
  randc bit [3:0]   id;       // 循环随机：16 个值遍历完才重复
  constraint c_addr { addr[1:0] == 2'b00;
                      addr inside {[32'h0:32'hFFFF]}; }
  constraint c_len  { len dist {1 := 50, [2:8] :/ 40,
                                [9:16] :/ 10}; }
  constraint c_ord  { solve wr before len;
                      wr -> len <= 8; }
  constraint c_dflt { soft data == 32'h0; }
  `uvm_object_utils(bus_txn)
  function new(string name = "bus_txn");
    super.new(name);
  endfunction
endclass
```

randc循环随机

取值不重复，直到遍历完全部取值后ID、通道号

:= 与 :/ 的区别

:= 给区间内每个值同样权重；:/ 把权重平分给整个区间

solve … before

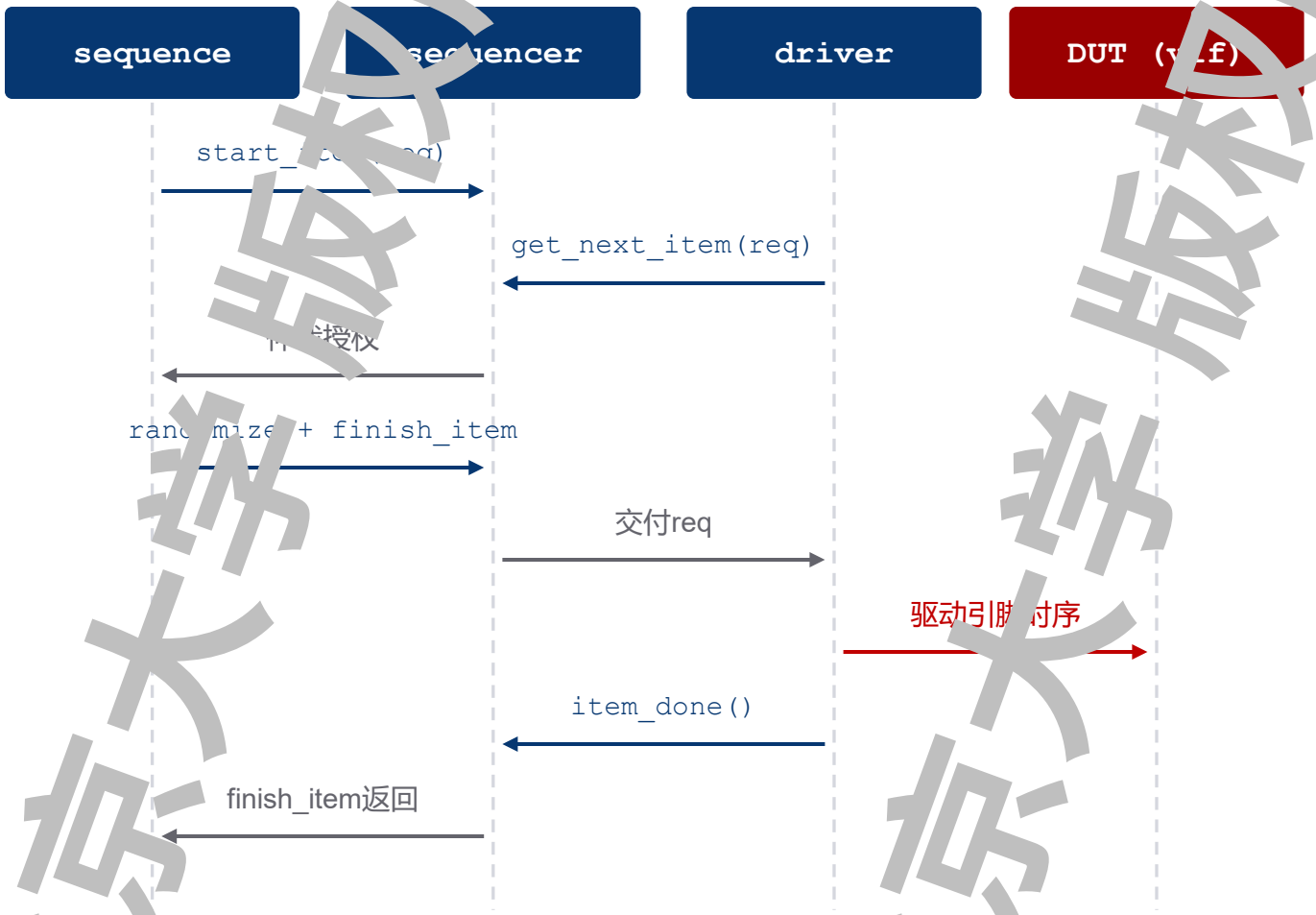
指定求解顺序，改变分布但不改变合法解空间

soft软约束

提供默认值，可被randomize() with {...} 覆盖

Sequence–Sequencer–Driver握手

- start_item / finish_item与get_next_item / item_done成对出现



成对调用

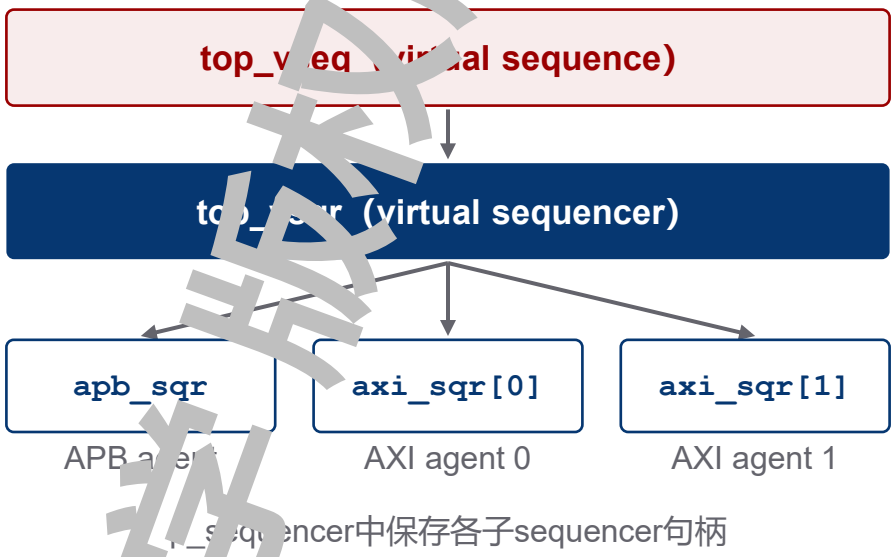
```
// driver: 拉取模式
task run_phase(uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(req);
    drive_item(req);
    seq_item_port.item_done();
  end
endtask

// sequence: 产生事务
task body();
  repeat (10) begin
    req = bus_txn::type_id::create("req");
    start_item(req);
    assert(req.randomize());
    finish_item(req);
  end
endtask
```

兼容并包

Sequence进阶：嵌套、库与Virtual Sequence

- virtual sequence不产生事务，只负责调度多个接口的sequencer



p_sequencer中保存各子sequencer句柄

virtual sequence示例

```
class top_vseq extends uvm_sequence;
  `uvm_object_utils(top_vseq)
  `uvm_declare_p_sequencer(top_vsqr)
  function new(string name = "top_vseq"); super.new(name); endfunction
  task body();
    cfg_seq_c  = cfg_seq::type_id::create("c");
    traffic_seq t0 = traffic_seq::type_id::create("t0");
    traffic_seq t1 = traffic_seq::type_id::create("t1");
    c.start(p_sequencer.apb_sqr);          // 1. 先配寄存器
    fork                                   // 2. 两路并发流量
      t0.start(p_sequencer.axi_sqr[0]);
      t1.start(p_sequencer.axi_sqr[1]);
    join
  endtask
endclass
```

嵌套sequence

在body()中start子sequence，像搭积木一样组合复杂场景

sequence library

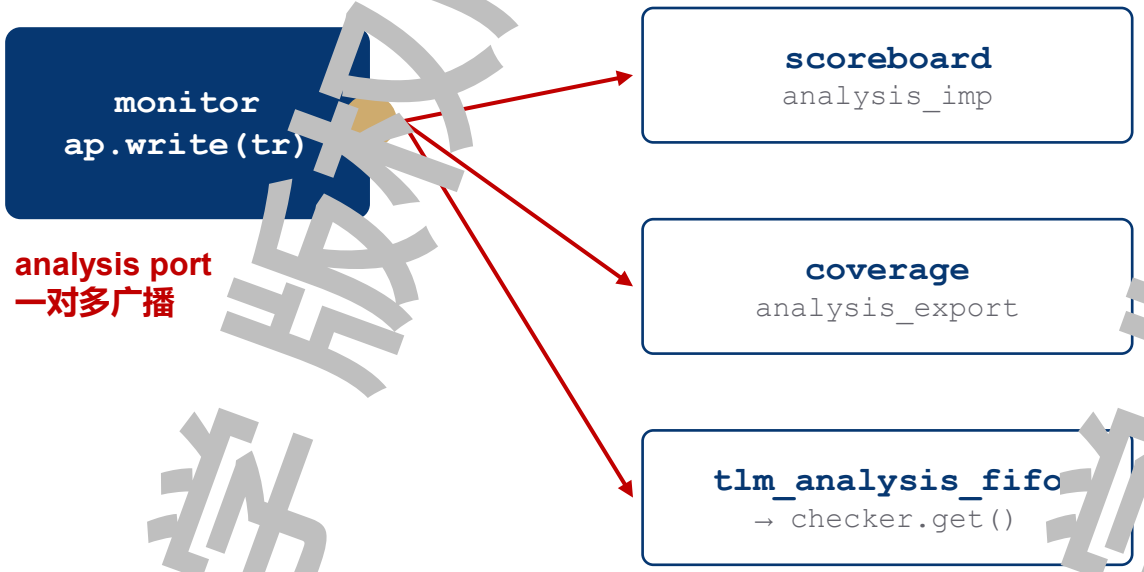
uvm_sequence_library按随机 / 循环模式挑选已注册的sequence

virtual sequence

跨接口协调时序：先配置、再并发、最后检查

TLM通信：port、export、imp与analysis port

- 组件之间只通过事务级端口通信，彼此解耦、可独立替换



analysis port
一对多广播

port	发起方：调用put / get / write
export	中转：把请求转发到内部组件
imp	实现方：真正实现write()等方法

analysis连接示例

```
// monitor
  analysis_port #(bus_txn) ap;
ap = new("ap", this);              // build_phase
ap.write(tr);                      // 采到一笔事务即广播

// scoreboard
  analysis_imp #(bus_txn, fifo_scb) imp;
function void write(bus_txn t);
  // 比对逻辑
endfunction

// env.connect_phase: port → imp/export
agt.mon.ap.connect(scb.imp);
agt.mon.ap.connect(cov.analysis_export);
agt.mon.ap.connect(fifo.analysis_export);
```

思想包袱 兼容并包

Factory机制：注册 创建与覆盖

- 用type_id::create代替new，测试层即可替换任何组件或事务类型

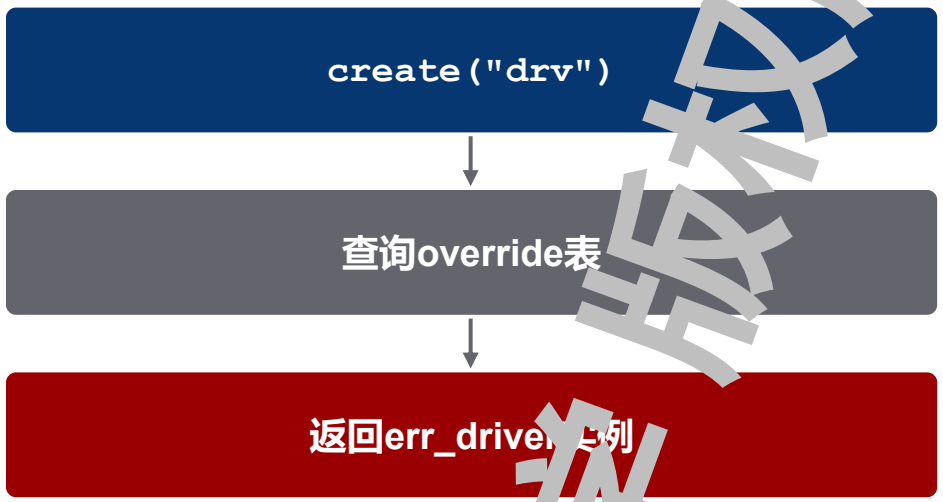
SystemVerilog / UVM

```
class my_driver extends uvm_driver #(bus_txn);
  `uvm_component_utils(my_driver)          // 注册到 factory
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
endclass

class err_driver extends my_driver;        // 注错版本
  `uvm_component_utils(err_driver)
  // new() 同上; 重写 drive_item() 注入错误
endclass

// agent.build_phase: 通过 factory 创建
drv = my_driver::type_id::create("drv", this);

// env.build_phase: 不改 env 代码即可替换
set_type_override_by_type(my_driver::get_type(),
                          err_driver::get_type());
set_inst_override_by_type("env.agt1.drv",
  my_driver::get_type(), err_driver::get_type());
```



type override

全局替换该类型的所有实例

inst override

替换指定层次路径的实例，优先级更高

思想：包容 兼容并包

uvm_config_db：层次化配置机制

- 按“路径 + 名字 + 类型”存取配置，最典型的用途是传递virtual interface

set / get

```
// tb_top (module)：把接口句柄放入配置库
initial begin
  uvm_config_db#(virtual fifo_if)::set(
      null, "uvm_test_top.env.agt*", "vif", fifo_if0);
  run_test();
end

// fifo_driver.build_phase：取出接口
if (!uvm_config_db#(virtual fifo_if)::get(
      this, "", "vif", vif))
  `uvm_fatal("NOVIF", "virtual interface not set")

// base_test.build_phase：把 agt1 配成 passive
uvm_config_db#(uvm_active_passive_enum)::set(
    this, "env.agt1", "is_active", UVM_PASSIVE);
```

作用域

inst路径相对context组件，支持 * 通配；get时按自身路径匹配

优先级

build期间层次更高的set优先；同一层次后set覆盖先set

类型须一致

set与get的参数化类型必须完全相同，否则get返回0

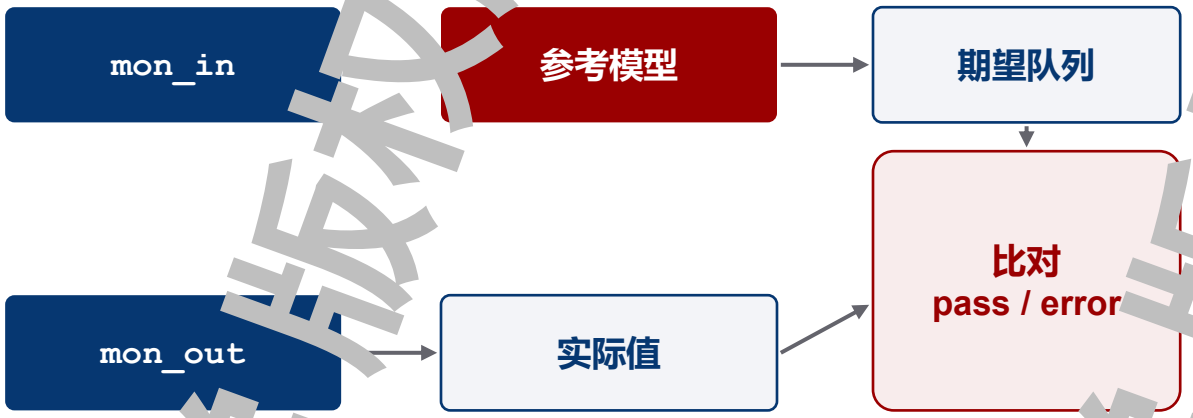
经验：get失败一律 `uvm_fatal，避免空句柄在仿真中途才报错

思想包袱 兼容并包

北京大学版权所有

Scoreboard与参考模型

- 预测值与实际值逐笔比对：参考模型可用SV、C/C++（DPI-C）或Python实现



in-order比对

期望值进入FIFO队列，实际值到达时逐笔弹出比对；适合保序接口

out-of-order比对

以[illegible]地址为键存入关联数组，乱序返回时按键匹配；适合A[illegible]乱序协[illegible]

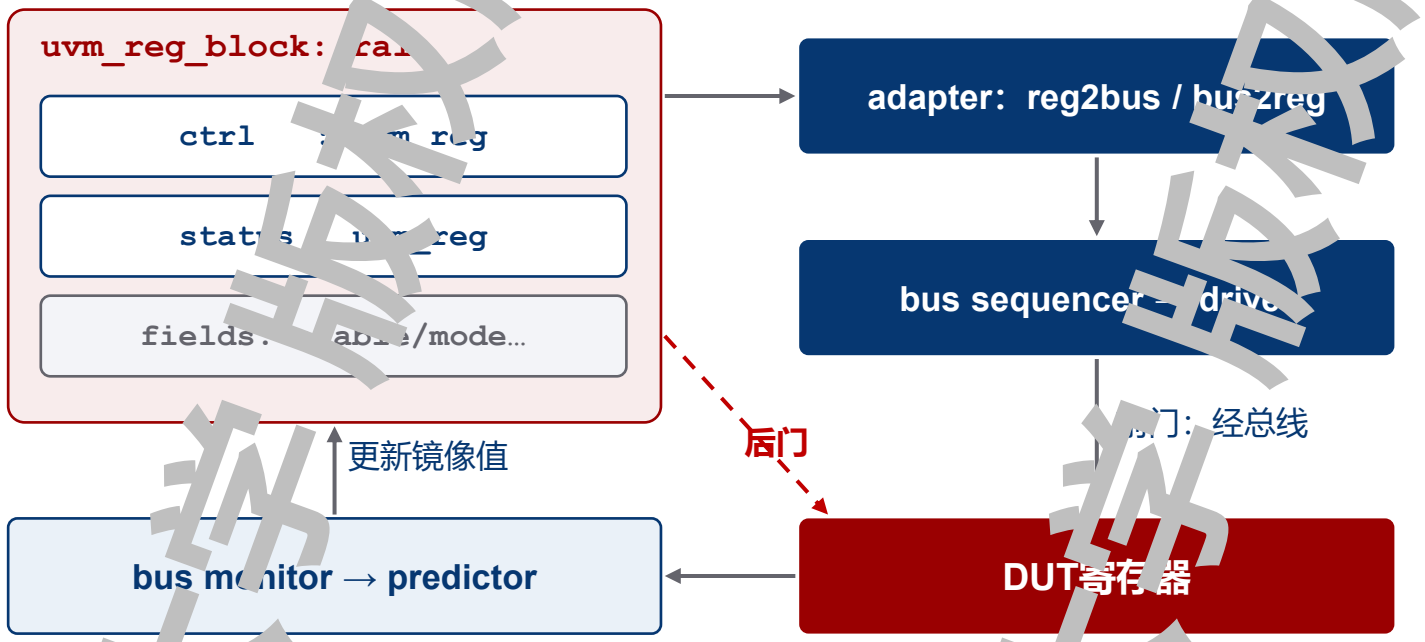
in-order scoreboard + DPI-C参考模型

```
[illegible]analysis_imp_decl(_in)
[illegible]analysis_imp_decl(_out)
[illegible]t "DPI-C" function int unsigned
 c_ref_alu(int unsigned a, int unsigned b, i[illegible]t [illegible]);

class alu_scb extends uvm_scoreboard;
  uvm_analysis_imp_in  #(alu_txn, alu_scb) i[illegible]mp;
  uvm_analysis_imp_out #(alu_txn, alu_scb) ou[illegible]mp;
  int unsigned exp_q[$];
  // ... `uvm_component_utils、new、build 略
  function void write_in(alu_txn t);
    exp_q.push_back(c_ref_alu(t.a, t.b, t.o[illegible]));
  endfunction
  function void write_out(alu_txn t);
    int unsigned expv = exp_q.pop_front();
    if (t.res !== expv)
      `uvm_error("SCB", $sformatf("e[illegible]=0h [illegible]=%0h",
                                  exp[illegible]t.res))
  endfunction
endclass
```

寄存器模型RAL

- 用抽象的寄存器读写代替总线细节：同一份测试可跨总线协议复用



寄存器sequence片段

```
uvm_status_e   status;
uvm_reg_data_t val;

// 前门：经总线写 / 读
ral.ctrl.enable.set(1);
ral.ctrl.update(status);
ral.status.read(status, val);

// 后门：直接修改 RTL 寄存器
ral.ctrl.poke(status, 32'h1);

// 读回并与镜像值比对
ral.status.mirror(status, UVM_CHECK);
```

前门访问frontdoor

write / read经总线完成，验证真实访问路径

后门访问backdoor

peek / poke经HDL路径直接读写，零时间、快速初始化

Testbench目录结构与运行

- 一次编译、多次测试：用命令行参数选择测试、打印级别与随机种子

典型目录

```
tb/
├── tb_top.sv          // DUT + interface + run_test()
├── fifo_if.sv
├── fifo_pkg.sv        // `include 全部 class
├── agent/
│   ├── fifo_xtn.sv        fifo_seq_lib.sv
│   ├── fifo_driver.sv     fifo_monitor.sv
│   └── fifo_agent.sv
├── env/
│   ├── fifo_scb.sv        fifo_cov.sv
│   └── fifo_env.sv
└── test/
    ├── base_test.sv       smoke_test.sv
    └── stress_test.sv
```

编译与运行

```
# Synopsys VCS
vcs -sverilog -ntb_opts uvm-1.2 \
    -timescale=1ns/1ps -f filelist.f -l comp.log
./simv +UVM_TESTNAME=smoke_test \
       +UVM_VERBOSITY=UVM_MEDIUM \
       +ntb_random_seed=1234 -l sim.log

# Cadence Xcelium
xrun -uvm -f filelist.f \
     +UVM_TESTNAME=stress_test -svseed random
```

+UVM_TESTNAME 同一份编译结果运行不同测试

+UVM_VERBOSITY 控制打印级别：LOW / MEDIUM / HIGH / DEBUG

随机种子seed 记录种子可精确复现失败用例

思想自由 兼容并包

小结：UVM的规范性让AI生成成为可能

- 固定的结构、接口与报告格式，正是LLM擅长“照模板填空”的场景

UVM机制	对AI生成的意义
固定的类结构与宏	可模板化生成，生成后易于静态检查
标准phase与TLM端口	组件可独立生成，按约定自动连接
factory与config_db	生成代码以override方式注入，不改动已有环境
统一的日志与报告格式	编译 / 仿真错误可被解析，反馈给LLM修复

下一部分

AI辅助Testbench生成

模板负责结构，LLM负责行为，EDA工具负责把关

北京大学
PEKING UNIVERSITY

AI驱动的功能验证

AI辅助Testbench生成

北京大学版权所有

为什么用LLM生成Testbench

- Testbench代码量大、模式化强——正是LLM擅长的“结构化生成”

人工搭建的痛点

样板代码多

每个agent都要interface、driver、monitor等多个文件

协议时序多

握手、背压、时序对齐，稍有疏忽即测错或漏测

搭建周期长

新IP往往要先花大量时间搭环境，才能开始真正的验证

依赖个人经验

风格不统一、新人上手慢，知识难以沉淀复用

LLM擅长

- 按模板生成样板代码
- 把规格描述翻译成代码
- 解释编译错误与仿真日志
- 起草断言与covergroup
- 补全注释与文档

LLM不擅长

- 精确的周期级时序推理
- 跨多个文件保持一致
- 规格中未写明的隐含约定
- 保证功能正确——会“自信地出错”
- 超长上下文中的细节

合理分工

LLM写初稿 → EDA工具验证 → 工程师审查签核

北京大学版权所有

LLM在硬件验证[illegible]的能力谱

- 从“补全一行代码”到“自主完成验证闭环”，自主性与集成度逐级提升



L1
代码补全
行级、片段级补全，
IDE内即时提示

已普遍可用

L2
组件生成
按规格生成UVM组件

可用，需审查

L3
断言与覆盖
生成SVA与covergroup
草稿

可用，需审查

L4
调试辅助
解析日志、定位错误、
给出修复

研究与试点

L5
验证智能体
自主规划、生成、仿真
与迭代

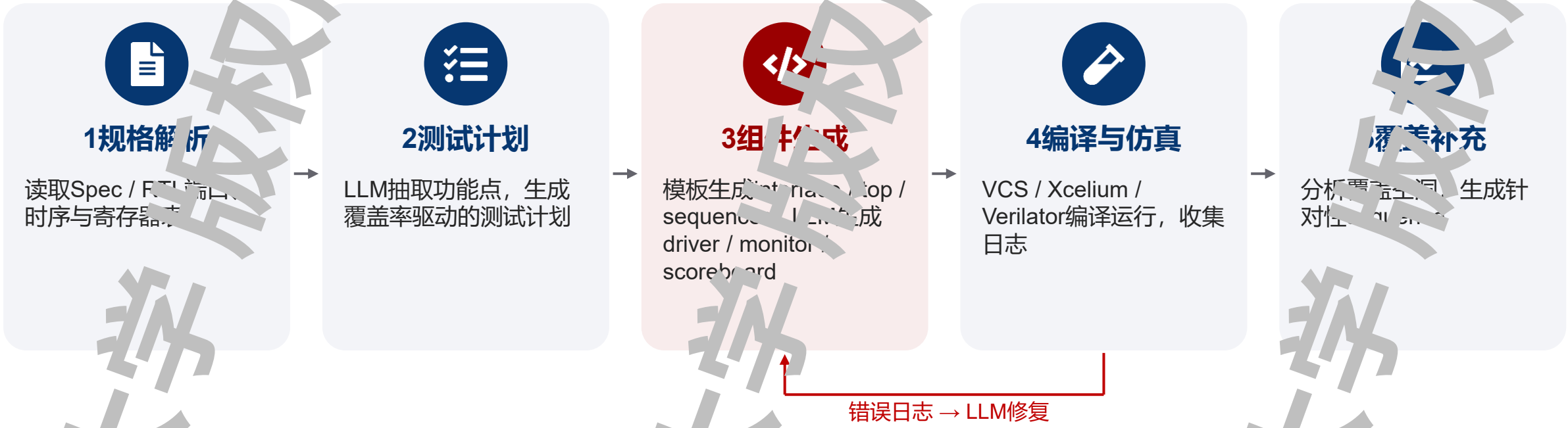
前沿探索

思想包容 兼容并包

北京大学 版权所有

LLM生成Testbench的整体流程

- 从规格到可运行的UVM环境：模板保证结构，LLM补全行为



代表性工作：UVM²（2025）

1个RTL设计（加密核、ALU、SPI、UART）上的实验结果

87.44%
平均代码覆盖率

89.58%
平均功能覆盖率

≈15.8×
环境搭建平均提速

来源：J. Ye et al., “From Concept to Practice: an Automated LLM-aided UVM Machine for RTL Verification”, arXiv:2504.19959；提速为相对资深工程师（约103分钟 vs 约27小时）

思想自由 兼容并包

输入准备：把规格结构化

- 自然语言规格先转成机器可读的JSON / YAML，再交给模板引擎与LLM

原始规格（自然语言）

sync_fifo为深度16、位宽32的同步FIFO。wr_en有效且未满时在时钟上升沿写入din；rd_en有效且非空时，下一拍dout输出数据。rst_n低有效异步复位，复位后empty=1。

为什么要结构化

- 消除自然语言歧义，缺项一目了然
- 模板引擎与LLM共用同一份“事实”
- 可与RTL端口自动比对，提前发现不一致

LLM抽取 + 人工确认的结构化规格（JSON）

```
{
  "module": "sync_fifo",
  "params": { "DEPTH": 16, "WIDTH": 32 },
  "clock": { "name": "clk", "edge": "posedge" },
  "reset": { "name": "rst_n", "active": "low", "async": true },
  "ports": [
    { "name": "wr_en", "dir": "in",  "width": 1  },
    { "name": "din",   "dir": "in",  "width": 32 },
    { "name": "rd_en", "dir": "in",  "width": 1  },
    { "name": "dout",  "dir": "out", "width": 32 },
    { "name": "full",  "dir": "out", "width": 1  },
    { "name": "empty", "dir": "out", "width": 1  }
  ],
  "timing": [ "rd_en & !empty -> dout valid at +1 cycle" ],
  "reset_values": { "empty": 1, "full": 0 }
}
```

测试计划自动生成

- LLM从规格抽取功能点，生成“功能点 → 场景 → 覆盖点 → 检查”的映射

结构化规格 → LLM抽取功能点 → 工程师评审确认 → 验证计划（Plan）

功能点	测试场景	覆盖点	检查方式
F1复位行为	复位期间及释放后的读写	cp_reset × cp_op	SVA：a_reset
F2基本读写	随机长度的读写序列	cp_op	scoreboard比对
F3满状态	写满后继续写入	cp_level.full	SVA + scoreboard
F4空状态	空时发起读操作	cp_level.empty × rd	SVA：empty保持
F5同时读写	各水位下同时读写	cross cp_op × cp_level	scoreboard比对
[illegible]	连续多拍读写不间断	transition bins	scoreboard比对

LLM只产出初稿：遗漏的功能点、错误的优先级都要由验证工程师在评审中纠正，确认后的Plan才进入下游生成。

思想包袱 兼容并包

北京大学出版社

Prompt与上下文构建

- 生成质量取决于“喂给模型什么”：结构化规格 + 检索增强 + 编码规范

结构化设计规格

端口列表、时序关系、协议、寄存器表——尽量表格化

UVM模板库 / VIP示例

RAG检索相似agent作为参考，减少“自由发挥”

团队编码规范

命名约定、宏用法、目录结构，保证可维护

历史错误与修复记录

作为few-shot示例，规避常见编译错误

Prompt示例

```
[角色] 你是资深 UVM 验证工程师
[背景] 模块: sync_fifo  DEPTH=16  WIDTH=32
       端口: clk, rst_n, wr_en, rd_en,
             din[31:0], dout[31:0], full, empty
       时序: rd_en 有效后下一拍 dout 有效
[任务] 生成 fifo_driver 与 fifo_monitor
[约束] 1. 遵循附带模板的类结构与宏
       2. 所有信号经 virtual interface 访问
       3. 只输出 SystemVerilog 代码
[参考] <RAG 检索到的 APB agent 示例>
```

经验：规格越结构化、约束越明确，生成代码的一次编译通过率越高

RAG知识库：让模型“照着样例写”

- 离线把模板、VIP与历史Testbench建成索引，在线按任务检索最相关的参考

离线：构建知识库



在线：检索增强生成



按语法单元切块

不要按固定字数切，避免把一个class切成两半

元数据过滤

协议类型、UVM版本不匹配的样例没有样例更糟

保留来源引用

生成代码标注参考出处，便于审查与追溯

思想包容 兼容并包

北京大学版权所有

模板 + LLM混合生成

- 结构固定的交给模板，行为相关的交给LLM，并按依赖顺序逐层生成

组件	生成方式	原因
interface / top / package	模板	结构固定，只依赖端口列表
sequencer / [illegible] / env	模板	连接关系规整，可参数化
sequence_item	模板 + LLM	字段来自规格，约束需理解语义
driver / monitor	LLM	协议时序行为，需读懂规格
scoreboard / 参考模型	LLM + 人工	功能语义，正确性最关键
sequence / test	LLM	测试场景由意图激励

被依赖者先生成并编译通过，再把其代码作为上下文生成依赖者，减少接口不一致。生成顺序参考UVM²（arXiv:2504.19959）。

依赖驱动的生成顺序

① **公共基础**
package · interface · transaction

② **叶子组件**
driver · monitor · sequencer

③ **组合组件**
agent（组装叶子组件）

④ **环境级**
scoreboard · env · test

思想自由 兼容并包

北京大学版权所有

示例：模板引擎生成tb_top

- Jinja2模板渲染同一份规格JSON：输出确定、可重复、可diff，无需LLM

模板：tb_top.sv.j2（Jinja）

```
{# tb_top.sv.j2 —— 输入：.json #}
module tb_top;
  import uvm_pkg::*;
  logic {{ clock.name }} = 0;
  always #5 {{ clock.name }} = ~{{ clock.name }};
  {{ prefix }}_if vif({{ clock.name }});
  {{ module }} #(
{%- for k, v in params.items() %}
    .{{ k }}({{ v }}){{ "," if not loop.last }}
{%- endfor %}
  ) dut (
    .{{ clock.name }}({{ clock.name }}),
    .{{ reset.name }}(vif.{{ reset.name }}),
{%- for p in ports %}
    .{{ p.name }}(vif.{{ p.name }}){{ "," if not loop.last }}
{%- endfor %}
  );
  initial begin
    vif.{{ reset.name }} = 1'b0;
    #20 vif.{{ reset.name }} = 1'b1;
  end
  initial begin
    uvm_config_db#(virtual {{ prefix }}_if)::set(
      null, "uvm_test_top.*", "vif", vif);
    run_test();
  end
endmodule
```

渲染（spec）

渲染结果：tb_top.sv（节选）

```
fifo_if vif (clk);
sync_fifo #(
  .DEPTH(16),
  .WIDTH(32)
) dut (
  .clk(clk),
  .rst_n(vif.rst_n),
  .wr_en(vif.wr_en),
  .din(vif.din),
  ...                    // rd_en, dout, full
  .empty(vif.empty)
);
```

为什么这部分用模板

- **确定性**：同一输入必得同一输出，无幻觉
- **同源**：参数、端口与规格JSON一一对应
- **可校验**：渲染后自动与RTL端口列表比对
- **可复用**：interface、agent、env骨架同理生成

示例：LLM生成事务类与约束

- 字段由模板从规格生成；约束要理解规格语义，交给LLM起草并逐条审查

fifo_txn.sv（LLM起草，经人审查）

```
typedef enum {IDLE, WRITE, READ, WRITE_READ} fifo_op_e;

class fifo_txn extends uvm_sequence_item;
  rand fifo_op_e op;
  rand bit [31:0] data;

  // 规格：读写为主，少量同时读写与空闲
  constraint c_op {
    op dist {WRITE := 4, READ := 4,
             WRITE_READ := 1, IDLE := 1};
  }
  // 规格：数据需覆盖全0、全1与一般值
  constraint c_data {
    data dist {32'h0 := 1, 32'hFFFF_FFFF := 1,
               [32'h1:32'hFFFF_FFFE] :/ 8};
  }

  `uvm_object_utils_begin(fifo_txn)
    `uvm_field_enum(fifo_op_e, op, UVM_ALL_ON)
    `uvm_field_int(data, UVM_ALL_ON)
  `uvm_object_utils_end
  function new(string name = "fifo_txn");
    super.new(name);
  endfunction
endclass
```

规格语句	生成的约束
读写比例约 4:4:1:1	`op dist {...}`
数据含全0、全1边界	`data dist {...}`
满时写入应被忽略	**不加约束！**（由测试覆盖）

审查要点：LLM常见错误

- **过约束：**把“非法/罕见”写成硬约束，如 (op==WRITE && full)，导致满状态永远测不到
- **dist误用：**:= 与 :/ 混淆，区间权重被放大
- **引用DUT状态：**事务类中不读vif；与水位相关的约束放到sequence的 with{} 中晚随机

示例：LLM生成的Driver

- 把“协议怎么驱动”写成时钟沿上的非阻塞赋值，并正确处理空闲周期

```
class fifo_drive  xtends uvm_driver #(fifo_txn);
  `uvm_component_uti s(fifo_driver)
  virtual fifo_if vif
  // new() 略
  function void bu ld_phase(uvm_phase phase);
    if (!uvm_co fig_db#(virtual fifo_if)::get(this, "", "vi ", vif))
      `uvm_f  ("NOVIF", "virtual interface not set")
  endfuncti
  task run_p ase(u m_phase phase);
    wait (vif.rs  n === 1'b1);
    forever begin
      @(p sedge vif.clk);
      se _ m_port.try_next_item(req);
        re  = null) begin                  // 无事务：驱 空闲
        if wr_en <= 1'b0;  vif.rd_en <= 1'b0;
      e d else begin
        vif r_en <= req.op inside {WRITE, WRITE_READ};
        vif.rd_en <= req.op inside {READ,  WRITE_REA
        if.din   <= req.data;
        s q_item_port.item_done();
      end
    e d
  nd sk
en as
```

1. **config_db获取vif**
   接口由top注入，driver不直接引用层次路径
2. **try_next_item处理空闲**
   无事务时拉低使能，有事务时与DUT背靠背驱动
3. **时钟沿 + 非阻塞赋值**
   在posedge用 <= 驱动，避免与DUT采样竞争

LLM常见错误（审查清单）

- 阻塞赋值 = 或在错误时钟沿驱动
- 漏写item_done导致sequence挂死

示例：Monitor与Scoreboard

- monitor只采样不驱动；scoreboard用队列实现FIFO的参考行为

fifo_monitor.sv（new/build_phase略）

```
class fifo_monitor extends uvm_monitor;
  `uvm_component_utils(fifo_monitor)
  virtual fifo_if vif;
  uvm_analysis_port #(fifo_txn) ap;
  bit rd_q;                          // 上一拍是否发生有效读
  task run_phase(uvm_phase phase);
    forever begin
      @(posedge vif.clk);
      if (rd_q) publish(READ, vif.dout);  // 读数据晚一拍
      if (vif.rst_n && vif.wr_en && !vif.full)
        publish(WRITE, vif.din);
      rd_q = vif.rst_n && vif.rd_en && !vif.empty;
    end
  endtask
  function void publish(fifo_op_e op, bit [31:0] d);
    fifo_txn t = fifo_txn::type_id::create("t");
    t.op = op;  t.data = d;
    ap.write(t);
  endfunction
endclass
```

fifo_scoreboard.sv（new略）

```
class fifo_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(fifo_scoreboard)
  uvm_analysis_imp #(fifo_txn, fifo_scoreboard) imp;
  bit [31:0] model_q[$];          // 参考模型：队列
  function void write(fifo_txn t);
    if (t.op == WRITE)
      model_q.push_back(t.data);
    else if (model_q.size() == 0)
      `uvm_error("SB", "read from empty model")
    else begin
      bit [31:0] exp = model_q.pop_front();
      if (t.data !== exp) `uvm_error("SB",
        $sformatf("exp %h, got %h", exp, t.data))
    end
  endfunction
endclass
```

本页要点：读数据晚一拍——LLM常在同一拍采样dout，漏掉“晚一拍”写法漏掉背靠背读

示例：从vPlan到sequence与test

- 每个功能点对应一个可回归的sequence；test只负责选择与配置

编号	sequence	关键约束 / 行为
F1	reset_seq	读写中途拉低rst_n
F2	rand_rw_seq	随机长度、随机op
F3	fill_full_seq	连续写 DEPTH+4 次
F4	read_empty_seq	空时连续读
F5	simul_rw_seq	op == WRITE_READ，遍历水位
F6	b2b_seq	多拍连续读写，无空闲

可追溯

sequence名、注释中的功能点编号与vPlan一一对应：覆盖率与回归结果可自动回填到计划

F3：fill_full_seq 与 full_test

```
class fill_full_seq extends fifo_base_seq;  // F3
  `uvm_object_utils(fill_full_seq)
  // new() 略
  task body();
    // 连续 DEPTH 次写满，后 4 次撞满
    repeat (DEPTH + 4) begin
      req = fifo_txn::type_id::create("req");
      start_item(req);
      assert(req.randomize() with {op == WRITE;});
      finish_item(req);
    end
  endtask
endclass

class full_test extends base_test;
  `uvm_component_utils(full_test)
  // new() 略
  task run_phase(uvm_phase phase);
    fill_full_seq s = fill_full_seq::type_id::create("s");
    phase.raise_objection(this);
    s.start(env.agt.sqr);
    phase.drop_objection(this);
  endtask
endclass
```

功能覆盖模型的自动生成

- 把vPlan的覆盖点翻译成covergroup：边界bins、transition与cross

fifo_cov.sv（LLM生成，经检查）

```
class fifo_cov extends uvm_subscriber #(fifo_txn);
  `uvm_component_utils(fifo_cov)
  fifo_op_e op;  int level;          // 软件维护的水位
  covergroup cg;
    cp_op : coverpoint op {          // F2
      bins wr = {WRITE}; bins rd = {READ}; }
    cp_level : coverpoint level {    // F3 / F4
      bins empty = {0};      bins one  = {1};
      bins mid  = {[2:14]};
      bins almost = {15};    bins full = {16}; }
    cp_trans : coverpoint op {       // F6
      bins wr_x = (WRITE [*3]);
      bins w2r = (WRITE => READ); }
    x_op_lvl : cross cp_op, cp_level; // F5
  endgroup
  function new(string name, uvm_component parent);
    super.new(name, parent);  cg = new();
  endfunction
  function void write(fifo_txn t);
    op = t.op;  cg.sample();         // 先采样
    if (t.op == WRITE) level++;      // 再更新水位
    else if (t.op == READ) level--;
  endfunction
endclass
```

1 **从vPlan到coverpoint**
每个功能点至少一个coverpoint，注释标注编号

2 **边界值单独成bin**
0、1、DEPTH−1、DEPTH 来自规格，不能

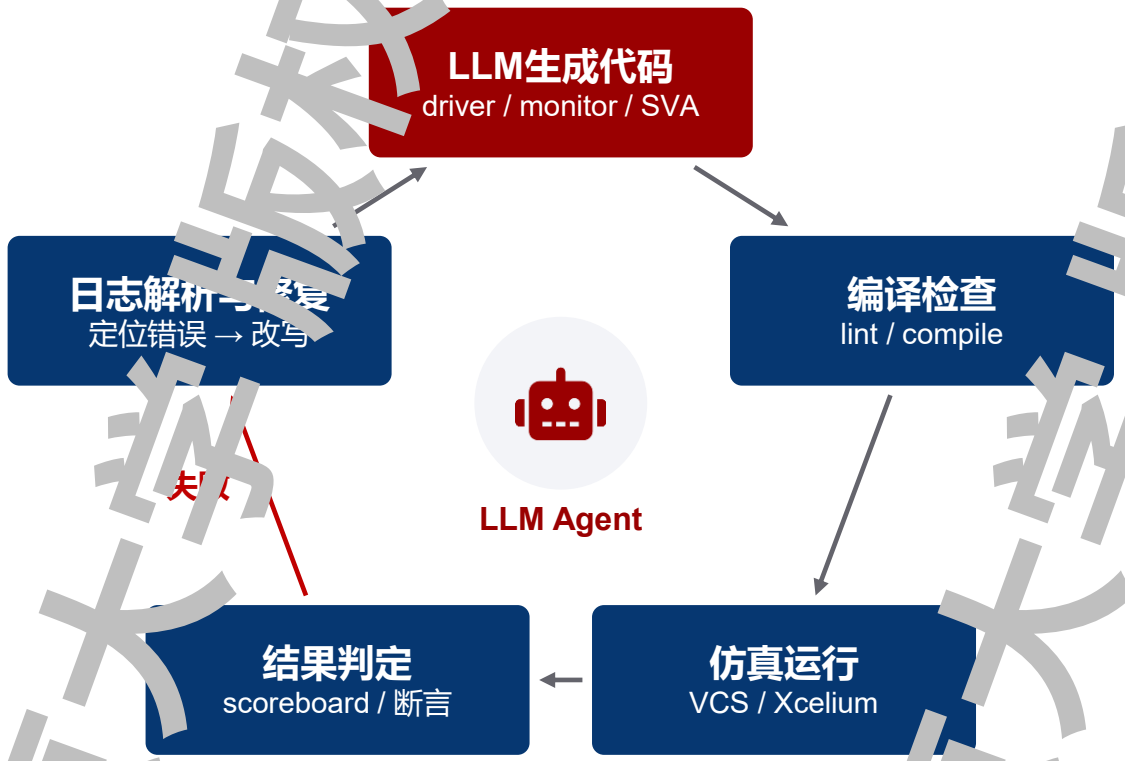
3 **transition与cross**
描述连续操作与组合场景；cross规模

LLM常见错误

- bins越界或漏掉边界值；cross组合爆炸
- 采样时机错误：先更新水位再采样，

生成—编译—仿真—修复闭环

- 让EDA工具当“裁判”：编译器与仿真器的反馈驱动LLM迭代自修复



闭环的工程要点

- 限定修复轮数，每轮保留版本快照
- 修复失败则回退到最近一次可编译版本
- 日志先过滤：只给首个错误及其上下文
- 每轮只改一个文件，便于diff审查

UVM²：最多两轮修复后的生成成功率

69.13% → 86.67%

首轮　　修复后

来源：UVM², arXiv:2504.19959（Testbench级生成成功率）

兼容并包

示例：编译错误自动修复

- 错误日志 → LLM定位根因 → 最小修改 → 重新编译验证

① 编译日志（示例，格式化）

```
Error-[IND] Ide      not declared
fifo_driver.sv,
  Identifier 'W      has not been
  declared yet.
```

② LLM定位与分析

根因：driver使用的枚举名WR_RD与fifo_txn中定义的WRITE_READ不一

策略：以公共类型定义为准，只改driver，不改transaction

③ LLM输出的最小修改（unified diff）

```
--- a/fi    driver.sv
+++ b/fi    driver.sv
@@ -17,    +17,2 @@
-       vif.wr_en <= req.op inside {WRITE, WR_RD};
-       vif.rd_en <= req.op inside {READ,  WR_RD}
+       vif.wr_en <= req.op inside {WRITE, WRITE_READ};
+       vif.rd_en <= req.op inside {READ,  WRITE_READ};
```

只喂首个错误 + 上下文

限定修改范围

改完立即重新编译

仿真失败的自动分诊（Triage）

- 回归中的大量失败先按“失败签名”聚类，再由LLM对每一类给出初步归因与分派建议

回归日志 → 提取签名 → 聚类 → LLM初判 → 分派复现

上千个失败用例

首个ERROR，去参数化

同签名 → 同一类

RTL / TB / 环境

owner + seed

签名提取

```
# 原始日志（两条不同用例）
UVM_ERROR @ 1530ns: uvm_test_top.env.scb [SB]
  exp 0000abcd, got 00001234
UVM_ERROR @ 8721ns: uvm_test_top.env.scb [SB]
  exp 00ff0000, got 00ff0000

UVM_FATAL @ 0ns: uvm_test_top.env.agt2.drv
  [NOVIF] virtual interface not set

# 去参数化后的签名
scb [SB] exp <HEX>, got <HEX>    // 第1类
[NOVIF] virtual interface ...   // 第2类
```

失败签名	数量	LLM初判	依据
scb [SB] exp <HEX>, got <HEX>	37	RTL：读指针	发生在空读之后
[NOVIF] virtual interface…	12	环境：config_db	仅新加的agt2
…	5	TB：objection	中途return

要点：先用规则聚类、再用LLM解释，避免对每个失败重复调用模型；LLM初判只是排序依据，结论须由owner复现确认。

表中数据为示意。

思想包袱 兼容并包

示例：仿真失败根因分析

- 把日志、相关RTL与波形片段一起交给LLM，要求给出“证据链”，而不只是结论

① 错误日志

```
UVM_ERROR fifo_sco  boar  (34) @ 55ns:
  uvm_test_top.env.  cb [SB  exp 0000000a, got 0000000b
```

② 波形片段（失败前后几拍，含内部信号）

拍	rd_en	rptr	empty	din	rptr	dout
1		0	1	—	0→1	—
2	0	1	1	0A	1	—
3	0	1	0	0B	1	
4	1	0	0	—	1→2	—
5	0	0	0	—	2	0B（期望0A）

③ LLM分析（证据链）

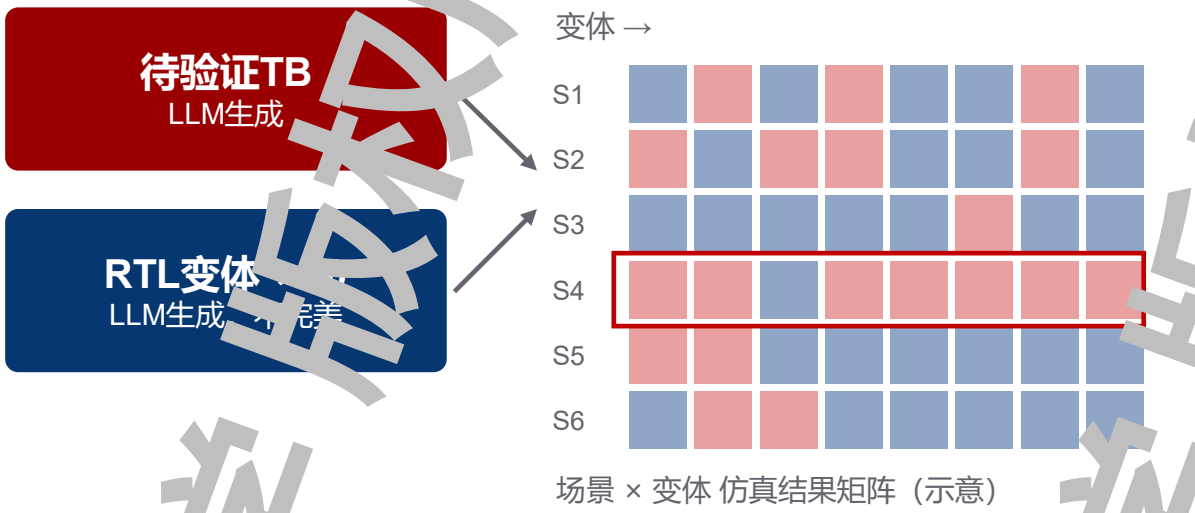
- 现象：第5拍读出0B，期望0A
- 证据：第1拍 empty=1 且 rd_en=1，rptr 仍由0变1；第2、3拍写入 mem[0]、mem[1]，第4拍读出mem[1]
- 判定：RTL缺陷——空时读未被屏蔽；TB行为符合规格
- 定位：sync_fifo.sv 读指针更新逻辑

④ 最小修改 + 回归断言

```
-  if (rd_en)            rptr <= rptr + 1;
+  if (rd_en && !empty)  rptr <= rptr + 1;
// 建议补充断言，防止回归
a_no_udf: assert property (@(posedge clk)
  disable iff (!rst_n) empty && rd_en |=> $stable(rptr));
```

功能正确性：Testbench自己也可能是错的

- 编译通过 ≠ 测对了——CorrectBench用多份RTL变体交叉验证TB



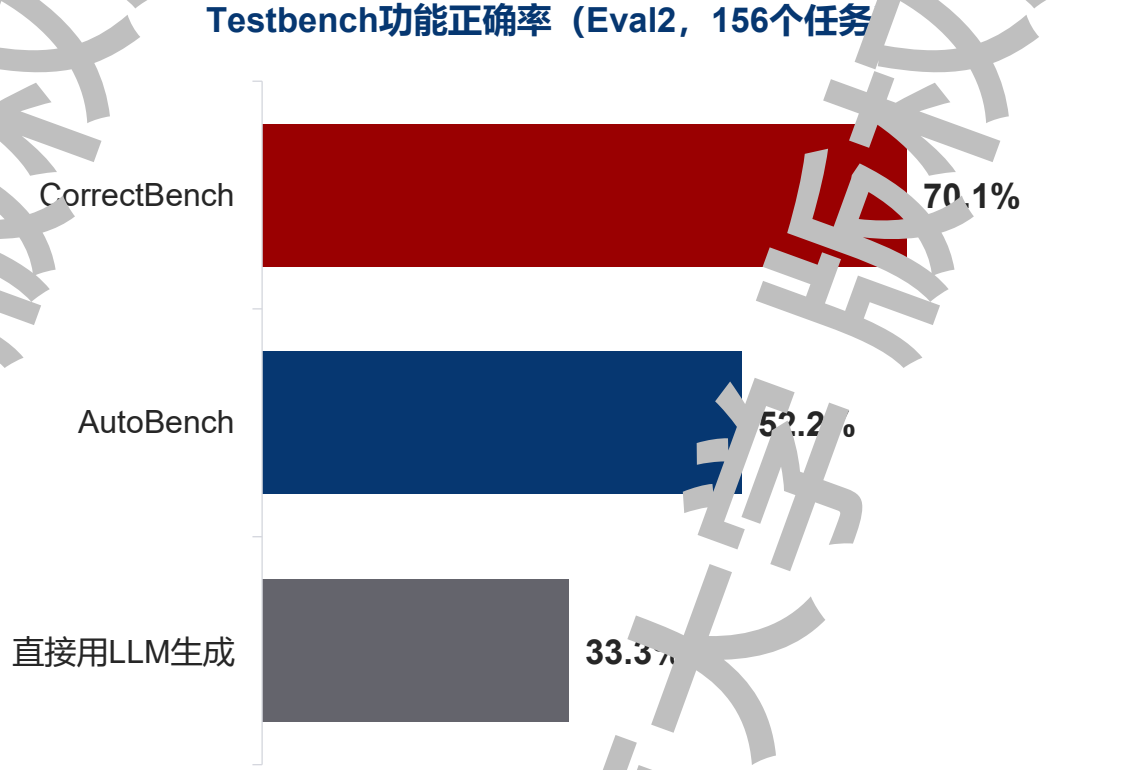
判定规则

- 某场景≥70%的变体与TB不一致 → 判为TB错误场景
- 全部变体完全通过 → 直接接受该TB

修正 ≤3次

重启生成 ≤10次

确认

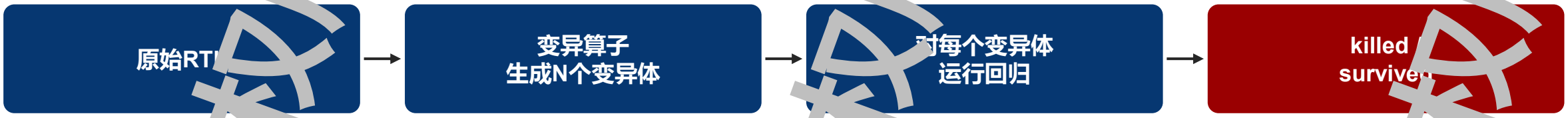


来源：R. Qiu et al., CorrectBench, arXiv:2411.085[illegible] AutoBench, arXiv:2407.03891。验证器判定准确率88.85%。

思想自由 兼容并包

变异测试：检验 testbench的检错能力

- 向RTL注入已知缺陷，统计被TB“杀死”的比例——覆盖率高不等于能发现错误



变异算子	原始代码	变异后
条件取反	`if (!empty)`	`if (empty)`
运算符替换	`rptr + 1`	`rptr - 1`
边界偏移	`count == DEPTH`	`count == DEPTH-1`
删除条件项	`rd_en && !empty`	`rd_en`
信号固定	`full`	`1'b0`

工具：开源MCY（Mutation Cover with Yosys）等；CorrectBench的RTL变体思路与此互补。
等价变异体（功能未改变）需剔除后再计算得分。

变异得分（示意）

46 / 50 = 92%

被杀死的变异体数 / 有效变异体数

存活变异体 → 交给LLM分析

- **例：** full判定偏移一位后仍存活——coreboard只比对已接受的写入，看不到写入被错误丢弃
- **建议：** 补断言 full == (count == DEPTH)，并补“写满后再读空”的计数检查

思想包袱 兼容并包

参考模型生成：LLM写C模型，经DPI-C接入

- 用与RTL不同的实现路径产生“期望值”，降低同源错误

C参考模型

```
// fifo_model.c  —— 生成的 C 参考模型
#include "svdpi.h"
#define DEPTH 16
static unsigned int q[DEPTH];
static int head = 0, tail = 0, cnt = 0;

void model_reset(void) { head = tail = cnt = 0; }

int model_push(unsigned int d) {     // 满则返回 0
  if (cnt == DEPTH) return 0;
  q[tail] = d; tail = (tail + 1) % DEPTH; cnt++;
  return 1;
}

int model_pop(unsigned int *d) {     // 空则返回 0
  if (cnt == 0) return 0;
  *d = q[head]; head = (head + 1) % DEPTH; cnt--;
  return 1;
}
```

SystemVerilog侧：DPI-C导入与调用

```
import "DPI-C" function void model_reset();
import "DPI-C" function int model_push(int unsigned d);
import "DPI-C" function int model_pop(output int unsigned d);

// scoreboard 中调用参考模型
function void write(fifo_txn t);
  int unsigned exp;
  if (t.op == WRITE) void'(model_push(t.data));
  else if (!model_pop(exp))
    `uvm_error("SB", "read from empty model");
  else if (t.data !== exp)
    `uvm_error("SB", "data mismatch");
endfunction
```

- **独立实现路径**：与RTL由不同prompt、不同语言产生
- **更接近算法规格**：C / Python描述行为，LLM生成正确率更高
- **可跨层复用**：同一模型可用于系统级、软硬件协同验证

示例：LLM生成FIFO断言（SVA）

- 自然语言性质 → SVA断言，可写入interface或通过bind挂到DUT

自然语言性质	生成的断言
复位释放时FIFO为空	`p_reset_empty`
满时只写不读，计数保持不变	`p_no_write_when_full`
full与empty不能同时为1	`p_full_empty_mutex`
full状态至少被观测一次	`cover property`

人工审查要点

- 时钟 / 复位语义与disable iff条件
- |-> 与 |=> 的一拍之差
- 配套cover，避免断言“空洞通过”

LLM输出（经人工审查）

```
// 1. 复位释放时 FIFO 为空
property p_reset_empty;
  @(posedge clk) $rose(rst_n) |-> empty;
endproperty

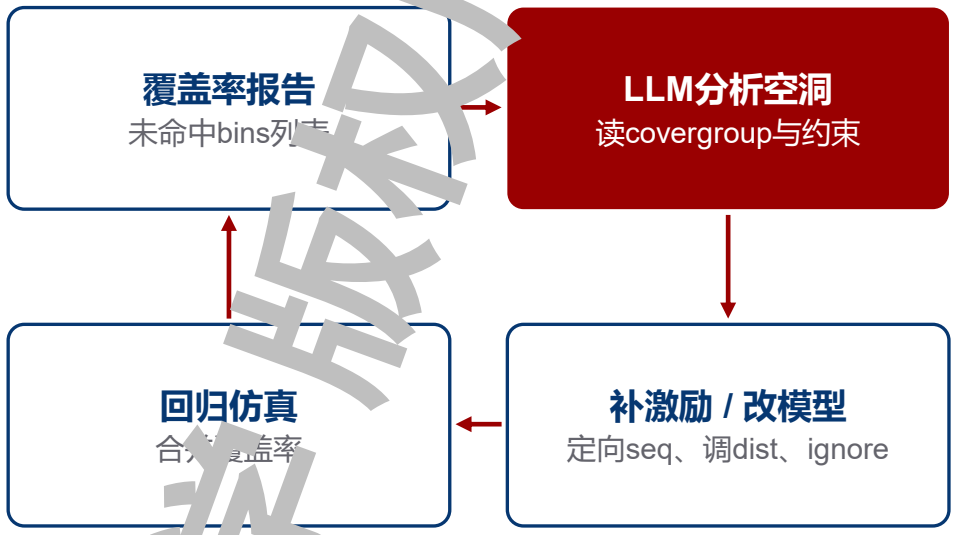
// 2. 满时只写不读：计数保持不变
property p_no_write_when_full;
  @(posedge clk) disable iff (!rst_n)
  (full && wr_en && !rd_en) |=> $stable(count);
endproperty

// 3. full 与 empty 互斥
property p_full_empty_mutex;
  @(posedge clk) disable iff (!rst_n) !(full && empty);
endproperty

a_reset : assert property (p_reset_empty);
a_full  : assert property (p_no_write_when_full);
a_mutex : assert property (p_full_empty_mutex);
c_full  : cover  property (@(posedge clk) full);
```

覆盖率驱动的激励补洞

- LLM先判断空洞是否可达，再决定补激励、调约束还是改覆盖模型



空洞（fifo_cov）	LLM判断	建议动作
x_op_lvl <wr, full>	**不可达**：monitor只发布成功写入，满时写不会被采样	改为采样“写尝试”，或ignore_bins（需评审）
cp_trans.wr_x5	**可达**：WRITE权重0.4，连续3次概率低	提高WRITE权重，或用b2b_seq定向
cp_level.lm st	**可达**：需连续15次净写入	生成n to_le el(15)定向序列

研究进展：LLM4DV（2023）

把未命中bins作为反馈写入prompt，迭代生成激励；在8个DUT上，最佳配置报告覆盖率89.74%–100%，但复杂设计上仍有困难

来源：Z. Zhang et al., arXiv:2310.04535

ignore_bins 与“不可达”结论必须人工确认：LLM可能把真正的设计缺陷（本应可达）误判为不可达，从而掩盖bug

智能回归：测试选择与排序

- 回归预算有限：根据代码变更、历史失败与覆盖贡献，预测哪些测试最值得先跑

代码变更
diff涉及哪些模块与文件

历史失败
近期失败越多，优先级越高

覆盖贡献
新覆盖命中的bins数

运行时长
按单位时间的价值排序

测试	独有覆盖	近期失败	时长	得分	本轮
full_test	18	3/20	4 min	0.82	✓
simul_rw_test	12	1/20	6 min	0.61	✓
reset_test	3	2/20	2 min	0.58	✓
rand_rw_test	2	0/20	15 min	0.12	—
smoke_test	0	0/20	1 min	0.05	—

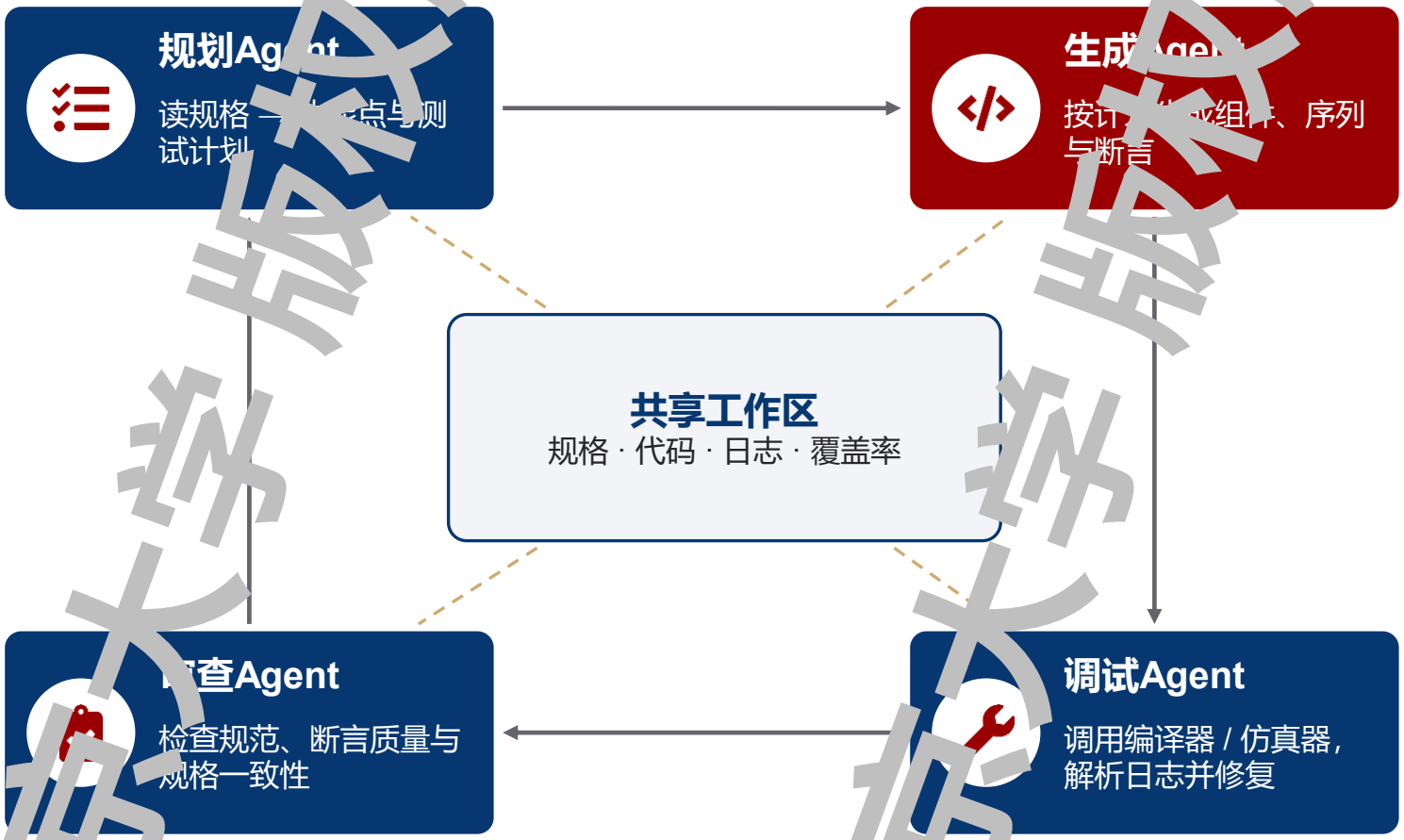
表中数据为示意；得分由各特征加权，权重可由历史回归数据训练。

LLM在回归中的角色

- **读commit diff / plan**：判断改动影响哪些功能点，给出“为什么选这些测试”的解释
- **保底机制**：每晚全量回归 + 随机种子轮换；被跳过的测试定期强制执行

多智能体验证框架

- 按角色拆分：规划、生成、调试、审查各司其职，围绕共享工作区协作



设计要点

- **角色分离**：生成者不自评，审查交给独立agent
- **工具即裁判**：编译、仿真、覆盖率结果作为客观反馈
- **共享状态 + 版本管理**：每步产物可追溯、可回退
- **人在回路**：测试计划、关键节点由工程师审批

实例：UVM² 采用分析（AgentA）、生成（AgentG）、优化（AgentC）分工（arXiv:2504.19959）

思想包袱 兼容并包

研究与产业进展一览

- 学术界聚焦TB生成与自校正，工业界把AI嵌入调试与回归平台

工作	年份	核心思路	代表结果
VerilogEval	[illegible]	Verilog设计题 + 参考实现的评测基准	常被用作TB生成的评测载体
AutoBench	2024	Verilog驱动 + Python检查器的混合TB [illegible]	Eval2通过率52.18%
CorrectBench	2024	RTL变体交叉验证TB，功能自校正	Eval2通过率70.13%
MEIC	2024	LLM迭代修复RTL的语法与功能错误	RTL调试自动化
UVLLM	2024	UVM环境 + LLM定位并修复RTL错误	语法修复率86.99%，功能修复率71.92%
UVM²	2025	多智能体生成完整UVM环境 + 覆盖率补充	代码 / 功能覆盖87.44% / 89.[illegible]%

工业界：EDA厂商推出生成式AI助手（如Synopsys.ai Copilot），并把AI嵌入调试与回归平台（如Cadence Verisium）[illegible]能力以厂商文档为准

来源：VerilogEval arXiv:2309.07544；AutoBench arXiv:2407.03891；CorrectBench arXiv:2411.08510；MEIC arXiv:2405.06840；UVLLM arXiv:2411.16238；UVM² arXiv:2504.19959

思想自由 兼容并包

局限、风险与人机协同

- LLM不是“验证审核者”：需要独立参考与人工把关

风险	对策
语法正确、语义错误 协议时序“幻觉”，编译通过却测错	**独立参考模型** C / Python golden model与DUT比对
同源偏差 TB与RTL基于同一份错误理解，互相“印证”	**变异测试（Mutation）** 向RTL注入缺陷，检验TB能否发现
覆盖过弱或过强 漏洞通过，或大量误报淹没真问题	**cover + vacuity检查** 确认每条断言的前件确实被触发
数据与IP安全 规格和RTL外泄到公有云模型	**私有化部署** 本地模型 + 本地RAG知识

北京大学版权所有

小结：AI辅助Te…Bench生成的最佳实践

- 模板负责结构，LLM负责行为，EDA工具负责裁判，工程师负责签核

实践	要点
结构化输入	规格JSON化；vPlan评审…才进入生成
模板 + LLM分工	结构交模板，行为交LLM，…依赖顺序逐层生成
工具做裁判	编译、仿真、覆盖率、变异测试提供客观反馈
人在回路	参考模型、断言、ignore_bins与签核由工程师把关

下一部分

时序逻辑的形式化验证与时态逻辑

仿真回答“跑过的场景对不对”，形式化回答“所有可达状态对不对”

北京大学版权所有

北京大学
PEKING UNIVERSITY

AI驱动的功能验证

时序逻辑的形式化验证与时态逻辑

北京大学版权所有

为什么时序逻辑需要形式化验证

- 仿真只能覆盖跑过的场景，形式化验证对所有可达状态给出证明或反例

$$2^{64} \approx 1.8 \times 10^{19}$$

64 位计数器可能的状态数

约 585 年

即使每秒遍历 10^9 个状态，穷举仍需的时间

	仿真	形式化验证
激励	人工或随机测试向量	无需激励，约束输入即可
覆盖范围	实际跑过的场景	所有可达状态（边界内）
结论	“没有发现错误”	证明成立，或给出反例
适用规模	可到整芯片	模块级、控制密集型逻辑
开源工具	Verilator、Icarus	SymbiYosys、ABC

FSM、仲裁器、握手接口、FIFO 控制逻辑——形式化验证的“甜区”

时序电路的形式化模型：状态迁移系统

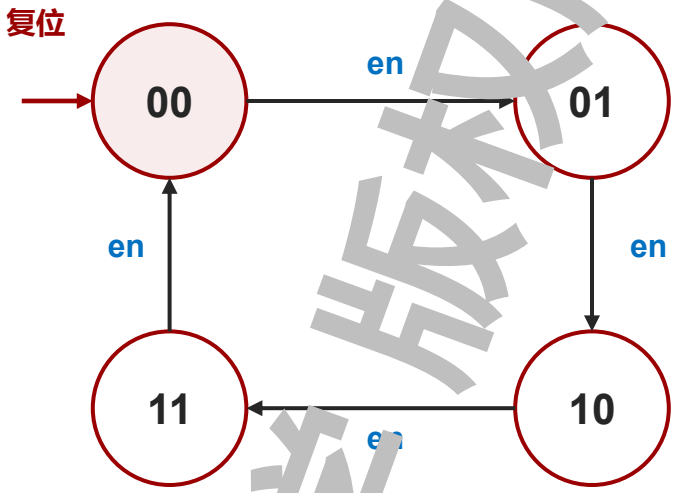
- 把RTL看作 M = (S, I, T, L)：寄存器取值构成状态，组合逻辑与时钟沿定义迁移

S　状态集合
全部寄存器取值的组合，n 位寄存器 → 2^n 个状态

I ⊆ S　初始状态
复位后的寄存器取值

T ⊆ S × S　迁移关系
组合逻辑 + 时钟沿：输入每拍任意取值

L　标记函数
每个状态上成立的原子命题，如 full、empty



en = 0 时保持自环状态（自环，图中略）

2位计数器

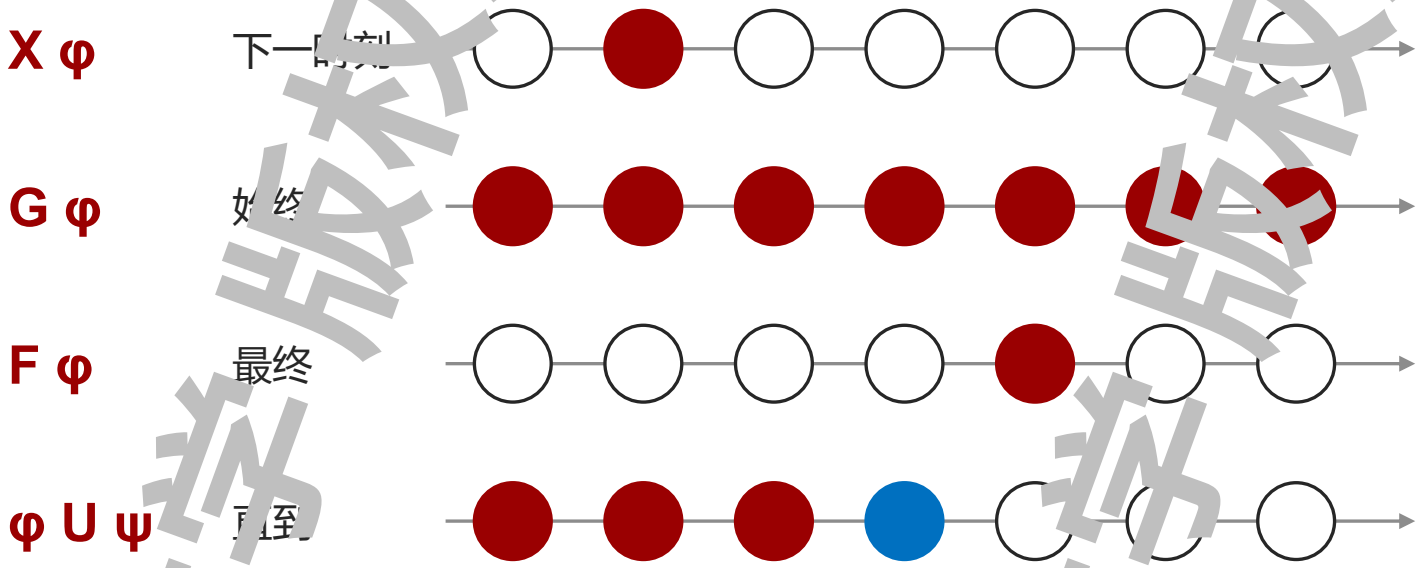
```
always @(posedge clk)
  if (!rst_n)  cnt <= 2'd0;
  else if (en) cnt <= cnt + 2'd1;
// S = {00,01,10,11}   I = {00}
```

RTL元素	模型中的含义
寄存器	状态变量（S）
复位值	初始状态（I）
组合逻辑	迁移关系（T）
原始输入	每拍非确定取值
输出 / 内部信号	原子命题（L）
assume	裁剪输入与迁移

可达状态
从 I 出发沿 T 能到达的状态，通常远少于 2^n；性质只需在可达状态上成立

线性时序逻辑（LTL）

- LTL 描述一条执行路径上性质随时间的变化，是 SVA 的理论基础



红点：φ 成立　蓝点：ψ 成立　从左到右为时间

G (req → F gnt)
每个请求最终都会被授权

G ¬(gnt0 ∧ gnt1)
两个授权不同时有效

安全性与活性

安全性：坏事永远不会发生

- 反例是一条有限长的路径
- 如互斥、FIFO 不溢出

活性：好事最终会发生

- 反例是一个无限循环
- 如无饥饿、无死锁

LTL语义与常用规范模式

- 性质在无限路径 $\pi = s_0\ s_1\ s_2\ \ldots$ 上判定；**常用模式可以直接翻译成SVA**

形式语义

$\pi \models p \iff p \in L(s_0)$

$\pi \models X\,\varphi \iff \pi^1 \models \varphi$

$\pi \models F\,\varphi \iff \exists\, i \geq 0 : \pi^i \models \varphi$

$\pi \models G\,\varphi \iff \forall\, i \geq 0 : \pi^i \models \varphi$

$\pi \models \varphi\ U\ \psi \iff \exists\, k \geq 0 : \pi^k \models \psi$ 且 $\forall\, j < k : \pi^j \models \varphi$

π^i：从 s_i 开始的后缀；$M \models \varphi \iff M$ 的每条初始路径都满足 φ

对偶与展开

$F\,\varphi \equiv true\ U\ \varphi$　　$G\,\varphi \equiv \neg F\,\neg\varphi$

$G(\varphi \wedge \psi) \equiv G\,\varphi \wedge G\,\psi$　　$F(\varphi \vee \psi) \equiv F\,\varphi \vee F\,\psi$

注意： $F(\varphi \wedge \psi)$ 与 $F\,\varphi \wedge F\,\psi$ 不等价

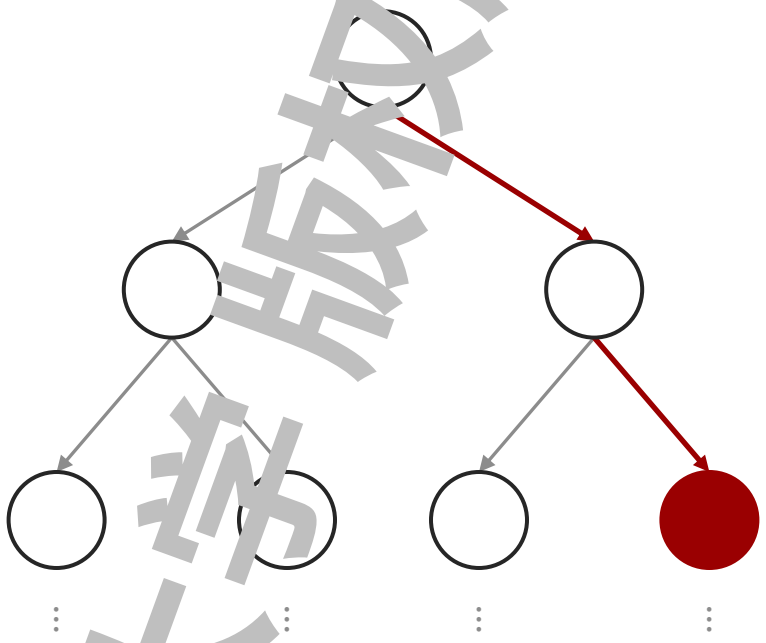
模式	LTL	SVA
不变式	G ¬bad	`!bad`
互斥	G ¬(g0 ∧ g1)	`!(g0 && g1)`
响应（活性）	G(req → F gnt)	`req |-> s_eventually gnt`
有界响应	G(req → X gnt ∨ X X gnt)	`req |-> ##1 gnt`
保持直到响应	G(req ∧ ¬gnt → X req)	`req && !gnt |=> req`

经验： 大多数工程性质都落在少数几种模式中；按模式生成断言，比自由翻译更不容易出错

模式分类参考 Dwyer et al., ICSE 1999

计算树逻辑（CTL）与路径量词

- CTL 在**分支的计算树**上推理：A 表示所有路径，E 表示存在某条路径



示例 EF p：存在一条路径（红）最终到达 p 成立的状态

AG p

所有路径上始终 p

不变式：FIFO 永不溢出

EF p

存在路径最终到达 p

可达性：错误状态可能出现

AF p

所有路径最终到 p

必然响应：请求必被处理

EG p

存在路径始终保持 p

可能永远停在某状态

LTL 与 CTL 的表达能力**互不包含**：工业界断言语言（SVA、PSL）主要基于线性时序

CTL 的 EF 常用于“这个状态能否到达”的覆盖检查

SystemVerilog 断言（SVA）

- SVA 用周期级语法书写时态逻辑：assert、assume、cover 各司其职

arbiter_prop_.sv

```
// 请求上升后 1~4 拍内必须授权
property p_req_gnt;
  @(posedge clk) disable iff (!rst_n)
  $rose(req) |-> ##[1:4] gnt;
endproperty
a_req_gnt: assert property (p_req_gnt);

// 授权互斥（最多一个有效）
a_mutex: assert property (
  @(posedge clk) $onehot0({gnt0, gnt1, gnt2}));

// 环境假设：请求在被授权前保持
m_req_hold: assume property (
  @(posedge clk) req && !gnt |=> req);

// 覆盖：同一请求者连续两次获得授权
c_b2b: cover property (@(posedge clk) gnt0 ##[1:4] gnt0);
```

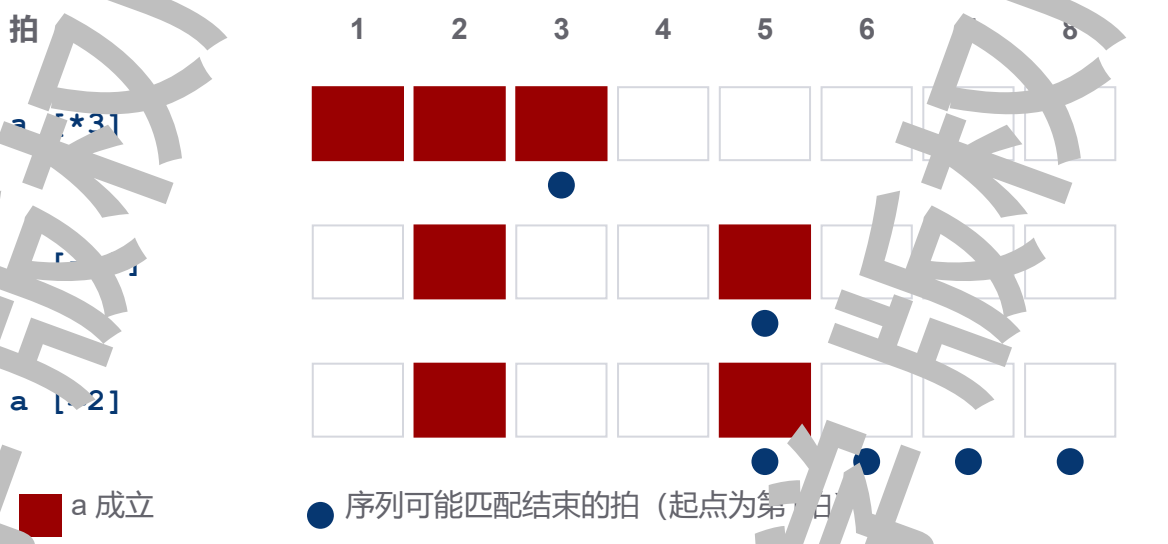
算子	含义
##n	延迟 n 拍
##[m:n]	延迟 m 到 n 拍
|->	同拍蕴含
|=>	下一拍蕴含
$rose / $fell	上升沿 / 下降沿
$stable / $past	保持不变 / 前一拍值
[*n]	连续重复 n 次

assert ≈ AG，cover ≈ EF

SVA序列算子进阶

- 重复、goto、非连续与约束算子：用序列精确描述多周期协议行为

算子	含义
`a [*m:n]`	a 连续成立 m 到 n 拍
`a [->n]`	goto：在 a 第 n 次成立的那一拍结束
`a [=n]`	非连续：a 第 n 次成立后可再延续
`a throughout s`	序列 s 期间 a 始终成立
`s1 within s2`	s1 发生在 s2 的区间之内
`s1 intersect s2`	两序列同起同止
`s_eventually p`	强性质：p 最终必须成立
`p until q`	q 出现前 p 一直成立（弱）



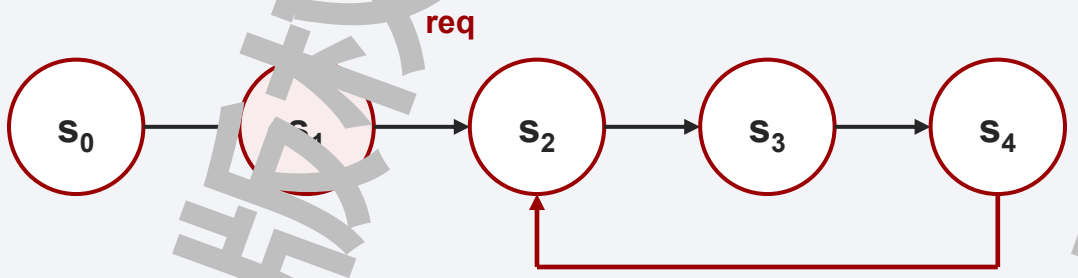
示例：突发传输

```
// 突发期间 valid 保持，第4次握手后下一拍 done
property p_burst;
  @(posedge clk) disable iff (!rst_n)
  $rose(start) |->
    (valid throughout hs[->4]) ##1 done;
endproperty
// hs = valid && ready
```

活性、公平性与弱性质

- 活性的反例是“套索”形路径：有限前缀 + 无限循环，证明活性常需要**公平性假设**

活性反例：套索（lasso）



SVA写法

```
// 强：gnt 必须最终到来（无界活性）
a_live: assert property (req |-> s_eventually gnt);
// 有界：可用 BMC 证明，反例更短更易读
a_bnd: assert property (req |-> ##[1:8] gnt);
// 公平性：环境不会永远阻塞
m_fair: assume property (s_eventually ready);
```

公平性假设

活性常依赖环境：若ready永远为0，任何设计都无法保证传输；
用 assume 排除“不公平”的路径

活性 → 安全性

工具把“存在不含gnt的无限循环”转化为“状态重复”检测，再用BMC/IC3求解（Biere等，2002）

强弱之分与工程建议

弱形式在仿真结束时不报错；能确定上界时优先写有界响应，无界活性用于发现死锁与饥饿

形式验证环境：assume、assert与cover

- 输入是“任意值”：assume刻画合法环境，cover检查是否过约束



assume 缩小的是路径集合：约束越多，证明越“容易”，也越可能失去意义

fifo_props.sv：检查器经bind挂载到DUT

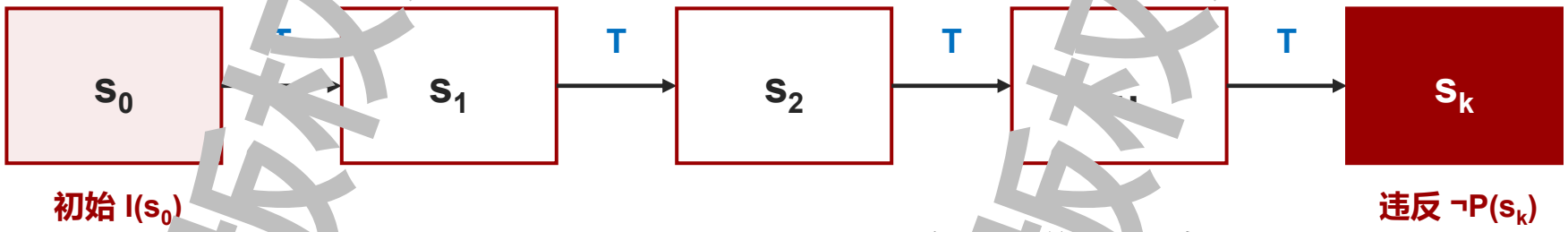
```
module fifo_p  #(parameter DEPTH = 16) (
  input logic clk, rst_n, wr_en, rd_en, full, empty,
  input logic [4:0] count);
  default clocking @(posedge clk); endclocking
  // 环境假设：复位释放后不再复位
  m_rst  : assume property (rst_n |=> rst_n);
  // 待证性质（复位期间不检查）
  a_cnt  : assert property (disable iff (!rst_n) count <= DEPTH);
  a_full : assert property (disable iff (!rst_n) full == (count == DEPTH));
  // 可达性：确认约束没有排除关键场景
  c_full : cover  property (full);
  c_wfull: cover  property (full && wr_en);
endmodule
bind sync_fifo fifo_props #(.DEPTH(DEPTH)) u_props (.*);
```

过约束陷阱

若再加 `assume property (full |-> !wr_en)`，“满时写入被忽略”就永远不会被检查——此时 c_wfull 变为不可达，正是报警信号

模型检测与有界模型检测（BMC）

- BMC 把电路展开 k 步交给 SAT：快速找到最短反例，但不是完整证明



展开深度 k
逐步加深

SAT Solver具体实现方式（学习可自行研究

$$I(s_0) \wedge T(s_0, s_1) \wedge \ldots \wedge T(s_{k-1}, s_k) \wedge (\neg P(s_0) \vee \ldots \vee \neg P(s_k)) \rightarrow \text{SAT 求解器}$$

1 模型检测思想

- 在状态空间上穷尽检查性质
- 符号化方法（BDD、SAT）缓解状态爆炸

2 可满足 → 反例

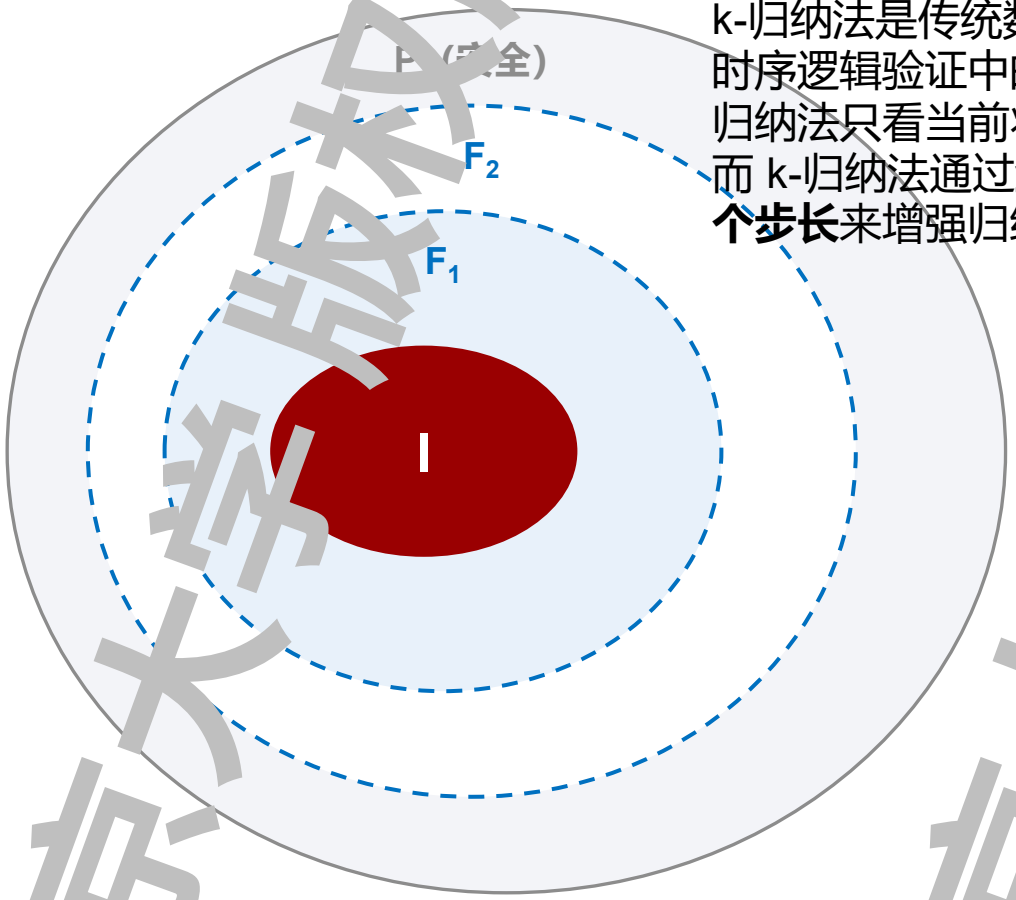
- 得到一条长度 ≤ k 的违例轨迹
- 导出 VCD 波形，便于调试

3 不可满足 → 界内安全

- k 步内不存在反例
- **不等于**完整证明，需要归纳

k 归纳与 IC3 / PDR

- **完整证明需要归纳不变式**：包含初始状态、对转移封闭、排除坏状态



F_i：i 步内可达状态的过近似；$I \subseteq F_1 \subseteq F_2 \subseteq \ldots \subseteq P$

k-归纳法是传统数学归纳法在时序逻辑验证中的扩展。传统归纳法只看当前状态（k=0），而 k-归纳法通过**查看连续的 k 个步长**来增强归纳假设。



k 归纳

- 基础：BMC 证明前 k 步都满足 P
- 归纳：任意连续 k 步满足 P，则下一步也满足
- 难点：不可达状态破坏归纳，需要补充归纳

IC3 / PDR

逐帧学习子句，阻断反例前驱；相邻两帧相同即得不变式

AI 机会：LLM 提出**候选不变式**（如“计数器不超过深度”），交给工具验证

读懂反例：从波形到根因

- 反例是一条**最短反例路径**：逐拍追踪哪个寄存器在哪一拍偏离了预期

两路轮询仲裁器（含缺陷）

```
reg [1:0] last;                          // 上次授权者
always @(posedge clk or negedge rst_n)
  if (!rst_n)                 last <= 2'd0;
  else if (gnt0)              last <= 2'd0;
  else if (gnt1)              last <= 2'd2;   // 缺陷：应为 2'd1
assign gnt0 = req0 && (!req1 || last != 2'd0);
assign gnt1 = req1 && (!req0 || last != 2'd1);
a_mutex: assert property (@(posedge clk) !(gnt0 && gnt1));
```

拍	rst_n	req0	req1	last	gnt0	gnt1	a_mutex
0	0	0	0	0	0	0	–
1	1	0	1	0	0	1	✓
2		1	1	**2**	**1**	**1**	✗

BMC在深度2给出最短反例。第0拍复位期间不检查；last在第1拍时钟沿被写为2，第2拍生效

1. **定位失败拍**
   第2拍 a_mutex 失败：gnt0 = gnt1 = 1
2. **回溯影响锥**
   gnt0、gnt1 只依赖 req0、req1 与 last
3. **找出异常值**
   last = 2，不在合法集合 {0, 1} 中
4. **追溯写入点**
   第1拍 gnt1 分支写入 2'd2 → 修复，并补 assert (last <= 1)

LLM可把VCD转成上表并按四步写出报告，但结论必须回到波形逐拍核对

状态空间爆炸与抽象技术

- 形式化能否收敛取决于问题规模：**先做减法，再求证明**

影响锥（COI）约

只保留性质所依赖的寄存器与逻辑；工具自动完成，是最[illegible]的抽象

黑盒化

把与性质无关的存储阵列、运算单元替换为自由输出，属于过近似

参数缩减

DEPTH 16→4、WIDTH 32→2：控制性质常与位宽无关，但需论证数据无关性

反例引导抽象（CEGAR）

先把多数寄存器切成自由变量；出现虚假反例再逐步加回

案例拆分

按工作模式、配置或操作码分别证明，每个子问题更小

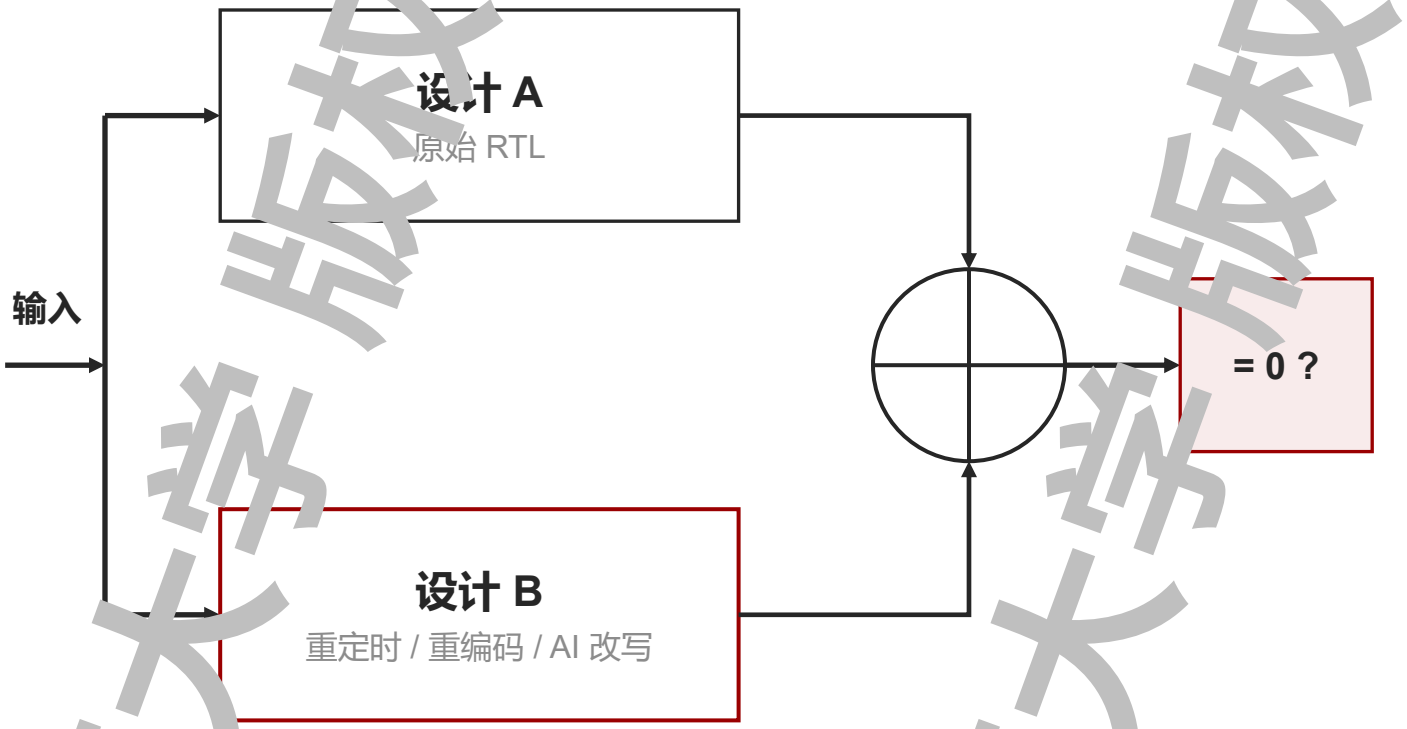
假设-保证

模块A已证明的assert作为模块B的assume，把系统级证明拆成模块级

[illegible]：抽象模型上证明成立 ⇒ 原设计成立；抽象模型上的反例可能是虚假的，必须在原设计上复现

时序等价性检查（SEC）

- 重定时、重编码或 AI 改写之后，用 SEC 证明新旧设计**输出始终一致**



Miter 电路：共享输入、比较输出，证明异或结果在所有可达状态下恒为 0

	组合等价	时序等价
前提	寄存器一一对应	不要求对应
方法	逐个比较组合锥	乘积机可达性 / 归纳
典型用途	综合前后检查	重定时、FSM 重编码、门控插入

与 AI 的结合

LLM 做“性能优化”改写后，SEC 是判断是否改坏功能的最后一道关

形式化签核：证明到底有多可信

- “全部PASS”之后还要回答三个问题：**约束是否过强、深度是否足够、性质是否足够**

约束是否过强

- 每个关键状态都配 cover；cover 不可达 ⇒ assume 可能过强
- 空洞成功：前件从不成立，断言“永远通过”
- 使用工具的 vacuity 检查，或为前件单独写 cover

深度是否足够

- 有界 PASS 只保证界内无反例
- k 要与设计的顺序深度相关，16深FIFO写满至少需16拍
- 报告写 “bounded k = 20”，而不是“PASS”

性质是否足够

- COI覆盖：哪些逻辑落在至少一条性质的影响锥内
- 证明核心：证明真正用到了哪些逻辑
- 变异：注入缺陷后应至少有一条断言失败

签核报告应包含： 每条性质的状态（证明 / 有界k / 反例）· 约束与深度 · assume 清单及审批人 · cover 结果 · 未覆盖逻辑说明

AI 方法：LLM 从规格生成 SVA 断言

- LLM 擅长把自然语言翻译成断言，但每条断言都要经过多重筛选才能入库

规格语句	LLM 生成的 SVA	需要人确认的点
请求后 4 拍内必须授权	$rose(req) |-> ##[1:4] gnt	“4 拍内”是否包含第 4 拍
FIFO 满时不允许写入	full |-> !wr_en	wr_en 指外部请求还是内部写使能
复位释放后 valid 为低	$rose(rst_n) |-> !valid	检查同拍还是下一拍

断言质量筛选流程

语法与编译检查 → 在参考设计上证明 → Cover 检查非空真 → 变异测试应能报错 → 人工审核入库

代表工作：AssertLLM 等用多个 LLM 从设计文档生成并评估断言（2024）；开源 RTL 与断言基准正在快速增加

AI 方法：LLM 辅助生成归纳引理

- k归纳失败时，反例常始于不可达状态：LLM提出辅助引理，形式化工具负责证明



例：Infineon工作中的计数器

```
// 两个同步4位计数器
always @(posedge clk)
  if (!rst_n) begin
    c1 <= 0;  c2 <= 0
  end else if (en) begin
    c1 <= c1 + 1;  c2 <= c2 + 1;
  end
// 目标性质：同时到达全1
assert property (&c1 == &c2);
```

归纳反例： c1=4'h[illegible]、c2=[illegible]'h3 满足 P，下一拍 c1=4'hF、c2=4'h4 违反 P（[illegible]达）

LLM 引理： `c1 [illegible]= c2`

它可归纳、蕴含 P，加入后证明立即完成

Kumar & Gadde（Infineon，2024）

LLM生成辅助断言并分析归纳失败；[illegible]与ECC设计上加速证明，强调人工审核

arXiv:2407.18965

Large Lemma Miners [illegible]25

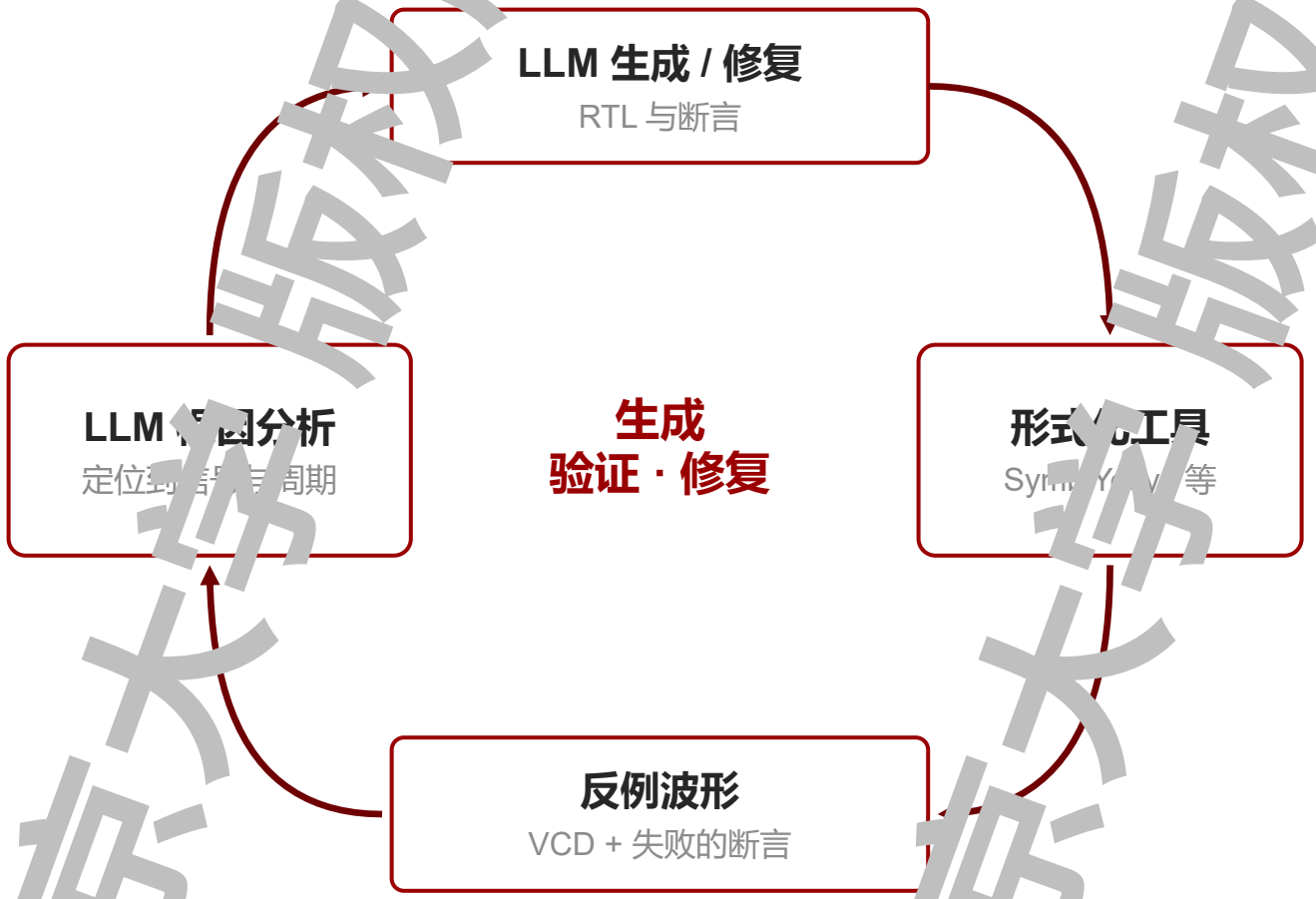
两种prompt框架；84%的[illegible]少一种设置得到可证明的归纳论证

Peled et al., arXiv:2511.02521

为什么安全：引理必须先被工具证明；错误引理只会被拒绝[illegible]证明“变假”

“生成—验证—修复”闭环

- LLM 负责提出方案，形式化工具负责裁决，**反例**是两者之间的通信语言



停止条件

- 全部断言证明通过，且 cover 均可达
- 达到迭代上限（如 5 轮）后转人工

风险与对策

- LLM 改断言“让它通过” → 断言文件只读
- assume 过强使证明空洞 → assume 需人工审批
- 有界证明被当成完整证明 → 报告中标注证明方式与深度