Modeling and Simulation 建模与仿真, 2026, 15(2), 1-8
Published Online February 2026 in Hans. https://www.hanspub.org/journal/mos
https://doi.org/10.12677/mos.2026.152028

Hans汉斯

基于模型的测试设计－面向复杂系统的软件与系统工程方法研究

张　妍[1]，尹溶森[2]，孙　波[2]

[1]西北工业大学计算机学院，陕西　西安
[2]北京空间飞行器总体设计部，北京

收稿日期：2025年12月30日；录用日期：2026年1月22日；发布日期：2026年2月2日

摘　要

基于模型的测试设计(Model-Based Test Design, MBTD)并不是单一的“自动生成用例”技术，而是一种把需求、架构边界、行为语义与环境假设统一到可计算资产中的测试设计方式。它强调在测试设计阶段就把关键知识写清楚：需求要能追踪，场景要能复现，判定要能落地，证据要能回溯。与传统以文档解释和经验枚举为主的用例编写相比，MBTD更关注：(1) 把自然语言需求转为可判定的验收条件；(2) 把行为模型转为可执行的场景骨架，并将断言与观测点嵌入模型；(3) 以覆盖与风险为约束，在有限预算内构造信息密度更高的用例集合；(4) 将执行结果回写到模型与差距清单，使回归选择与后续迭代有据可依。本文在梳理相关标准与研究进展的基础上，给出一套更贴近工程实践的MBTD流程与关键技术要点，包括模型资产构成、用例包组织、覆盖度量与选择策略、测试数据与Oracle设计、以及与持续集成相结合的落地方法。

关键词

基于模型，测试设计，需求追踪，覆盖度量，测试数据，判定逻辑(Oracle)，持续集成

Model-Based Test Design-Research on Software and Systems Engineering Methods for Complex Systems

Yan Zhang[1], Rongsen Yin[2], Bo Sun[2]

[1]School of Computer Science, Northwestern Polytechnical University, Xi’an Shaanxi
[2]Beijing Space Vehicle Master Design Department, Beijing

Received: December 30, 2025; accepted: January 22, 2026; published: February 2, 2026

文章引用: 张妍, 尹溶森, 孙波. 基于模型的测试设计－面向复杂系统的软件与系统工程方法研究[J]. 建模与仿真, 2026, 15(2): 1-8. DOI: 10.12677/mos.2026.152028

Abstract

Model-based Test Design (MBTD) is not a single "automatic generation of test cases" technique, but a test design approach that unifies requirements, architectural boundaries, behavioral semantics and environmental assumptions into computable assets. It emphasizes that key knowledge should be clearly stated during the test design stage: requirements should be traceable, scenarios should be reproducible, judgments should be implementable, and evidence should be traceable. Compared with the traditional use case writing mainly based on document interpretation and experience enumeration, MBTD pays more attention to: (1) converting natural language requirements into decidable acceptance conditions; (2) Transform the behavioral model into an executable scene skeleton and embed assertions and observation points into the model; (3) Constrained by coverage and risk, construct a set of use cases with higher information density within a limited budget; (4) Write the execution results back to the model and gap list to provide a basis for regression selection and subsequent iterations. Based on a review of relevant standards and research progress, this paper presents a set of MBTD processes and key technical points that are closer to engineering practice, including model asset composition, use case package organization, coverage measurement and selection strategies, test data and Oracle design, as well as implementation methods combined with continuous integration.

Keywords

Model-Based, Test Design, Demand Tracking, Coverage Metric, Test Data, Decision Logic (Oracle) Continuous Integration

1. 引言

测试设计常被放在“开发完成之后”去谈，但在很多项目里，真正昂贵的并不是执行，而是设计：怎么把分散的需求解释成可验证的场景，怎么在复杂交互中找到最可能出错的边界，怎么在版本快速迭代时保持回归既不过度也不漏测[1]。当系统行为跨越多模式、多组件、多时钟域，且接口有延迟、噪声、饱和等现实因素时，仅靠经验枚举往往会出现两类问题：一类是“覆盖看起来很高，但关键事故场景没被碰到”；另一类是“回归集合越堆越大，维护与执行成本失控”，最后反而拖慢交付节奏[2]。

实际工程里常见的尴尬场景是：需求文档写得很完整，但每个人理解的触发条件不一样；接口表列了字段，却没有单位、边界与采样周期；状态机图画得漂亮，却没有标出哪里要观测、哪里要判定、失败时应该抓取哪些证据[3]。于是测试脚本写出来后，执行结果经常要靠会议解释，一旦人员变动或版本迭代，旧脚本就成了“只能跑、跑完也说不清”的包袱。在安全关键或强合规的场景下，这种包袱还会扩散到审计与交付：证据链补不齐，返工成本会成倍增长[4] [5]。

在这种背景下，MBTD 的意义并不神秘：它试图把测试设计从“解释文本、猜测场景”的劳动，转为“围绕模型资产组织信息、围绕度量做选择”的工程过程。所谓模型资产，既包括 SysML/UML 等结构与行为视图，也包括接口契约、约束表达、仿真组件和环境剖面。关键不在于形式，而在于它能否被分析、被追踪、被复用。更直白地说：如果一个模型既不能告诉你该测什么，也不能在失败时解释为什么失败，那它对测试设计的帮助就十分有限[6]。

需要强调的是，MBTD 并不等同于“全自动生成测试”。全自动在很多团队里并不现实：模型不一定完备，接口与执行环境也不一定稳定。多数项目更可行的路径是“半自动 + 可度量”：用模型给出场景骨架、边界与断言模板，再由工程人员补齐关键数据点、执行细节与证据采集。这样做的收益往往立竿见影——变更发生后可以做影响分析，回归集合可按证据裁剪，失败定位能从“系统不对”收敛到“哪条契约被破坏”[7]。

1.1. 研究问题与目标

本文围绕三个更贴近工程的研究问题展开。第一，测试设计需要哪些模型信息才能真正“可计算、可复用”，并形成稳定的资产边界？第二，从模型到测试用例(更准确地说是用例包)应遵循怎样的流程，才能兼顾可读性与可执行性？第三，在资源预算有限的情况下，如何用覆盖度量与风险权重引导用例选择，并把执行反馈纳入闭环？对应地，本文目标是给出一套可落地的 MBTD 方法框架与实践要点，形成可直接迁移的写作与工程模板。

与纯粹的理论推导不同，本文更关注“把事情做成”的细节：稳定标识如何设定、哪些信息必须写进需求条目、断言如何参数化、回归集合如何分层、执行失败如何在报告里给出定位线索等。这些问题看似偏工程，却决定了 MBTD 在团队里能走多远[8]。

1.2. 论文结构安排

全文采用较为传统的论文结构组织。第 2 章介绍相关概念、标准与研究进展，并界定本文所说 MBTD 的边界；第 3 章给出面向工程落地的模型资产体系与测试设计流程，强调“用例包化”与“追踪闭包”；第 4 章进一步讨论覆盖度量、用例选择、测试数据与 Oracle 的关键技术细节；第 5 章从过程与工具链角度讨论持续集成场景下的落地方法、质量门禁与评估指标；第 6 章总结全文并给出后续研究方向。

2. 相关概念、标准与研究进展

“基于模型”在软件与系统工程里是一个容易被泛化的词。有人把画过状态机也叫基于模型，有人把自动生成脚本才算基于模型；还有人强调形式化语义和可证明性。为避免讨论失焦，本文把 MBTD 限定为测试设计阶段的方法：用模型来组织测试信息、推导场景结构、指导数据选择并支撑判定与追踪，而不是讨论模型本身如何被验证或证明。换句话说，MBTD 关心的是“用模型做测试设计”，而不是“用测试验证模型”[9]。

在标准层面，ISO/IEC/IEEE 29119-1 给出了测试活动的基本概念、术语与过程框架，强调测试应具备可追踪与可复现的证据链；ISO/IEC/IEEE 29148 进一步规范了需求工程，特别是需求的唯一标识、来源与验收准则，这些内容直接影响测试设计的可判定性。系统工程的 15288 则从生命周期过程角度要求把 V&V 活动嵌入开发流程，而不是事后补救。这些标准并不专门讨论 MBTD，但它们共同指向一个事实：如果缺少稳定的标识、验收尺度与过程证据，测试设计再“聪明”也难以工程化。

在建模语言与规范方面，SysML 用于表达系统结构、需求、行为与参数约束，是不少复杂系统项目的主流选择；UTP 则为测试构件、测试用例与判定提供了更明确的建模语义。但在真实项目里，模型往往同时承担多种目的：沟通、设计、仿真、验证。当模型被用于测试设计时，最常见的问题是“视图有了，但缺少可测试标注”：状态转换没有明确触发条件，接口没有范围与单位，观测点和断言也不清楚。因此，本文后续更强调“建模到可测试语义”的补齐原则，而非仅停留在语言介绍。

研究进展方面，基于模型的测试(MBT)在软件领域已经形成较为成熟的方法谱系，包括从状态机、时序图、决策表等模型出发生成测试序列与数据，并通过覆盖准则控制规模。Utting 与 Legeard 的专著系统讨论了工具链与覆盖策略，是这一方向的基础读物。在系统级建模语境中，早期 V&V 也在持续发展，近

年的系统性综述指出：越早在模型阶段引入可执行与可验证语义，越能减少后期集成与回归成本，并提升跨团队协作的效率。

从方法角度看，MBT 与 MBTD 之间的差异常常被忽略。MBT 更像是一个“生成与执行”的技术族群，而 MBTD 强调设计过程与工程闭环：用例为什么这样设计、覆盖缺口如何解释、数据策略为何选择、失败证据如何归档。这些内容不属于某一个算法，却决定了团队能否长期复用测试资产。尤其是在需求频繁变更的项目里，如果没有追踪矩阵与影响分析，回归就很容易退化为“全量跑一遍”，最后把时间花在重复验证上[1] [4] [5]。

另一个经常被低估的因素是执行环境的可迁移性。很多团队在仿真阶段做了大量测试，但进入在环或实物阶段后不得不“重新搭一套”，原因往往是接口与调度语义不一致，或者环境配置不透明。因此，把环境资产与接口契约纳入 MBTD 框架非常必要。FMI 3.0 对时钟与调度接口进行了更系统的规范，为跨工具链的模型复用与测试环境构建提供了可借鉴的基础[10]。

3. 基于模型的测试设计方法体系

本章给出本文推荐的 MBTD 方法体系。与其把 MBTD 理解为“从模型生成一堆路径”，不如把它看成一套围绕资产与闭环的工作方式：先把需求、接口、行为和环境沉淀成可计算资产，再将测试成果以“用例包”的形式交付，并通过追踪矩阵与覆盖度量持续修正。用例包的概念很关键，它把步骤、数据、断言与环境组合成闭包，避免测试脚本脱离上下文而难以复现。

3.1. 模型资产的构成与最小完备条件

工程上最容易走偏的地方是：把建模当作目标，而不是手段。为了让模型真正服务于测试设计，本文建议将模型资产分为四类，并给出每类资产的“最小完备条件”。需求与约束资产必须具备稳定 ID、来源与可判定验收条件；结构与接口资产必须明确 SUT 边界、变量集合、单位范围与异常处理契约；行为与模式资产必须覆盖关键模式与关键转移，并在关键处补齐触发、观测点与断言；环境与运行剖面资产必须把噪声、延迟、饱和等假设显式化，并能描述常见工况与高风险工况的权重。这些条件看似琐碎，却直接决定后续能否生成可执行场景、能否写出可自动判定的 oracle，以及能否在迭代中做回归裁剪。

在很多项目里，最缺的不是状态机，而是“可观测量”。需求写得很漂亮，但没有明确观测点和采样方式，测试人员只能通过日志猜测系统是否满足要求；一旦系统行为涉及窗口统计、抖动、漂移等非功能特性，这种猜测会迅速失效。MBTD 强调在模型层面就把观测点与判定尺度写清楚：哪些变量是判定依据，采样周期是多少，窗口如何定义，阈值容差怎么取。这样做不是让建模变复杂，而是把原本会在集成阶段爆发的争议提前解决。

除此之外，资产治理同样不可回避。若元素 ID 在迭代中频繁变化，追踪矩阵很快就会失效；若模型与测试资产分散在不同仓库、不同命名规则下，影响分析就会变成手工对照。因此，建议将 ReqID、模型元素 ID 与用例包 ID 统一纳入版本控制，并记录每次迭代的变化集(新增/修改/删除及原因)。在评审门禁上，可以设置最小规则：关键需求必须写清验收准则与可观测量；关键转移必须有触发、观测点与断言标注；若暂时无法实现，应给出原因与替代证据。门禁的目的不是“卡住人”，而是防止资产质量在迭代中逐步腐烂。

3.2. 用例包化：从“用例文本”到“可复现交付物”

传统用例往往是一段描述性文本或一段脚本，离开作者就难以维护。本文建议将测试设计交付物统一为用例包(Case Package)：它至少包含四部分内容。第一是场景步骤，描述事件与输入的顺序以及期望反应；第二是数据集，明确数据点的来源与生成策略；第三是断言集合，即硬判定规则与软判定监控项；

第四是环境配置，包含版本、初始化状态、接口连线、采样与日志设置等。在实践中，追踪链接与证据索引同样重要：用例包应能回指到 ReqID 与模型元素，执行结果也应能回指到断言 ID 与失败位置。

用例包化的一个直接好处是：它让“写用例”变成“组装资产”。当模型或接口变更时，影响的往往是某些触发条件、某些断言参数或某段环境配置，而不是整条用例从头重写。因此，用例包的维护成本通常低于纯脚本方式。更重要的是，当团队要求审计与复现时，用例包天然携带了复现所需的信息闭包，避免临近交付再“补材料”。

为了让用例包既能被人阅读，也能被流水线执行，建议在表达上保持克制：步骤部分尽量贴近工程语言，不追求过度形式化；数据与环境部分则尽量结构化，可被机器解析；断言部分用稳定 ID 与参数化模板表达。这种“人读的写法”和“机器跑的字段”并行的组织方式，往往比把一切都做成复杂模型更容易被团队接受。

3.3. MBTD 工作流程：切片 - 骨架 - 数据 - 判定 - 编排

在流程层面，本文将 MBTD 归纳为五步。第一步是模型切片：围绕目标需求或风险点，从全模型裁剪得到最小相关子模型，并明确外部依赖与假设。第二步是生成场景骨架：在子模型上导出可读的步骤序列(状态/事件/输入序列)，并与关键转移对齐。第三步是数据设计：为步骤中的输入域选择合适策略(等价类、边界值、组合覆盖、约束求解、扰动注入或剖面采样)。第四步是 oracle 绑定：把断言与观测点绑定到步骤、状态或窗口统计，并为失败输出定位字段。第五步是回归编排：基于追踪矩阵与覆盖增量，构造冒烟、变更与全量回归集合。这五步的共同目标是把测试设计做成“可审计的工程产物”，而不是“只能靠作者解释的脚本”。

这里的“切片”值得多说一句。切片不是把模型删薄，而是把边界写清楚：哪些组件在 SUT 内，哪些需要桩模拟；哪些外部输入可以控制，哪些只能观测；哪些延迟或噪声属于假设，哪些属于需求必须覆盖的范围。边界写清楚后，后续生成的场景骨架与数据集才不会在执行时陷入大量无效路径。一些工作提出基于依赖图的切片与增量更新思路，能够在模型变更时只重算影响域对应的用例包，这对高频迭代的工程场景尤其有价值。

4. 覆盖、选择、数据与 Oracle 的关键技术

如果说第 3 章解决了“怎么组织资产与流程”，那么本章更关注“怎么把资源花在刀刃上”。在真实项目里，测试预算总是有限的：时间有限、算力有限、在环资源有限、甚至可构造的环境也有限。因此，覆盖度量与选择策略必须能解释、能落地，而不能只是漂亮的理论指标。

4.1. 覆盖度量的多维视角

单一的结构覆盖(例如转移覆盖)在复杂系统中往往不够用。原因很简单：同一条转移在不同输入域、不同延迟与不同噪声下，可能表现出截然不同的动态响应；更不用说跨组件交互时，缺陷常常隐藏在消息顺序与竞争条件里。因此，本文建议采用多维覆盖视角：结构覆盖衡量状态/转移是否触达；交互覆盖衡量关键消息序列与接口组合是否出现；约束覆盖衡量边界值、时序窗口、禁止条件是否被击中；风险覆盖衡量高风险元素的缺口是否长期存在。多维覆盖并不是为了增加报表，而是为了让不同角色(开发、系统工程、质量、安全)能在同一套事实基础上讨论取舍。

在度量实现上，一个常见的误区是把覆盖当成“打点统计”。更合理的做法是把覆盖项设计成可追踪的对象：例如给每条关键转移、每条关键断言、每个关键交互序列分配稳定 ID，并把“被覆盖”定义为“触发且满足特定条件”。举例来说，边界覆盖不只是走到某条分支，而是要在阈值上下的狭窄区间内命中；时序覆盖不只是发生过某次切换，而是要在给定窗口内完成并满足稳态保持。只有这样，覆盖

指标才真正与验收准则对齐。

4.2. 风险权重与用例选择

覆盖与风险的关系需要说清楚：高覆盖不等于高安全，高风险也不等于只测异常。工程上更可行的做法是引入风险权重，把风险视为“覆盖项的优先级”。风险权重可以来自多种来源：需求优先级、历史缺陷密度、专家评审、甚至简化的 FMEA 打分(严重度 × 发生度 × 可探测度)。这些权重不必一开始就很精确，但要能稳定复用，并在迭代中随着证据更新。当风险权重与运行剖面结合后，就能自然形成策略：高频路径保证稳定性，高风险元素增加边界与扰动采样密度，同时把长期未覆盖的高风险缺口显式暴露在差距清单中。

用例选择可以分为两个阶段：先过滤，再选择。过滤剔除不可执行或不可判定的包；选择则在剩余集合中最大化覆盖增量。对多数团队而言，贪心策略足够实用：每次挑选“新增覆盖/新增风险收益”最大的用例包，直到预算用尽。关键在于把成本也纳入度量：一次在环执行可能比仿真执行贵几个数量级，同样的覆盖增量，在不同平台上的投入产出完全不同。如果把成本显式写进选择逻辑，就能避免回归集合在无意识中膨胀。

4.3. 测试数据：从“写几个数据”到“可追溯的数据集”

数据设计经常被低估。许多用例失败不是场景不对，而是数据点没打到关键区域。本文建议把数据集作为一等资产：每个数据点都应能追溯到生成策略与目标覆盖项。对连续输入，等价类与边界值仍然是最稳妥的基本功；对离散配置，组合覆盖可以显著压缩数量；对强约束场景，约束求解能够生成满足前置条件的有效数据，避免大量无效执行；对鲁棒性与容错，扰动注入(噪声、偏置、丢包、延迟、饱和)比“随机乱抖”更有意义；对长稳态或负载相关需求，运行剖面采样更接近真实运行。这些策略并不相互排斥，工程上常用的组合是：先求解进入条件，再在临界点附近加密边界采样，并在窗口内叠加扰动。

为了让数据集可复现，至少应记录三类信息：范围与分布(阈值、步长、概率权重)、生成过程(求解器/组合算法/脚本版本)、以及随机性控制(随机种子与重放参数)。不少团队在回归失败后无法复现，根源就是数据集缺少这些记录。当数据集版本化后，测试设计也会变得更透明：一次覆盖提升究竟来自“新增场景”还是“补齐边界数据点”，一眼就能看出来。

此外，还要正视现实中的非确定性：并发调度、网络抖动、传感噪声会让同一数据点出现不同输出。在这种情况下，数据策略应更偏向“分布与区间”而不是“单点命中”。比如对时延指标，不仅给出是否在 Δt 内完成，还应记录完成时间分布；对输出稳定性，不仅判定是否越界，还应记录窗口内的波动统计。这些记录会让报告更像“工程证据”，而不是“过没过”的简单打勾。

4.4. Oracle：把判定从“解释”变成“规则”

Oracle 是 MBTD 能否真正落地的核心。所谓 Oracle，不只是一个“期望值”，而是判定结构：哪些条件必须满足、在什么窗口内满足、允许多大容差、失败时输出什么证据。本文建议把 Oracle 分为硬判定与软判定两类。硬判定对应契约与断言，直接给出 Pass/Fail；软判定对应趋势与异常模式，用于捕捉退化、漂移、抖动变大等问题。二者配合可以减少“明明系统变差但测试全绿”的尴尬。

Oracle 设计有一个常见陷阱：规则写得很强，但观测点不足，导致要么无法自动判定，要么误报频繁。因此，Oracle 必须与可观测性一起设计。对关键规则，建议在模型或接口层面明确观测字段、采样周期与统计窗口；对非功能指标，建议输出完整的分布或统计量，而不是只输出一个是否通过的布尔值。此外，每条断言都应有稳定 ID，失败时输出定位字段：断言 ID、触发步骤/状态、失败时间戳、相关变量快照与期望区间，这样才能把失败从“系统不对”压缩到“哪个规则、在什么条件下被破坏”。

从形式上看，Oracle 既可以用简单阈值表达，也可以用更强的时序约束表达。但工程上往往没必要一开始就追求复杂形式化；更重要的是先把“判定边界”写清楚：容差从哪里来(传感器精度、数值积分误差、量化误差还是控制策略允许的死区)，窗口如何选取(进入稳态后才统计还是全过程统计)，以及在存在抖动时如何避免把正常波动误判为失败。把这些细节写清楚，往往比引入复杂逻辑更能减少争议。

5. 工程化落地与持续集成

方法再漂亮，如果不能嵌入日常流程，就会停留在汇报材料里。工程落地最关键的问题是：如何把 MBTD 融入持续集成，让它在每次迭代中产生可见收益。本章从过程、工具链与质量门禁三个角度讨论落地要点，并尽量使用项目中常见的语言来描述，避免把问题“讲成只有论文里才会发生”。

5.1. 过程与角色：把“谁负责什么”写清楚

MBTD 不是测试团队单方面能完成的事情，因为模型资产往往分布在多个角色手中。更现实的做法是明确分工：需求/系统工程负责需求条目的可判定性与追踪；架构与建模人员负责 SUT 边界、接口契约与关键行为标注；测试工程负责用例包、数据集与 Oracle 实现；平台/工具人员负责流水线、环境构建与证据归档。在评审机制上，建议把“用例包与模型标注”作为评审对象，而不仅是脚本。评审重点也应发生转变：从“步骤写得像不像”转为“触发条件是否充分、观测点是否可实现、断言是否可定位、覆盖缺口是否解释清楚”。

为了让协作顺畅，团队还需要一套“说同一种话”的约定。比如：状态与模式如何命名，关键转移如何编号，断言 ID 的组成规则是什么，日志字段与观测点如何对齐。这些约定一开始看起来像“规范文件”，但如果不做，后面出现问题时就会变成无穷无尽的对齐会议。从这个意义上讲，MBTD 不仅是技术方案，也是一种协作方式。

5.2. 工具链：开放接口与可替换结构

工程上最怕的是工具锁定与流程割裂。本文建议把工具链按三层拆分：建模层、转换层、执行观测层。建模层负责需求、接口、行为与环境资产；转换层负责切片、骨架生成、数据生成与覆盖计算；执行观测层负责仿真/在环/实物执行以及日志、轨迹、快照采集。三层之间最好通过开放格式或 API 连接，使团队可以按成熟度替换局部组件，而不至于推倒重来。当项目涉及协同仿真或跨工具链集成时，标准接口(例如 FMI)能够显著降低环境复用成本，并提高测试资产的可迁移性。

从持续集成角度看，最容易被忽略的是“可复现实验环境”。如果每次跑出来的结果都依赖某台机器、某个临时配置，那么测试就会变得脆弱：偶发失败越来越多，团队开始怀疑测试本身，最后把门禁放松。因此，建议把环境配置做成明确工件：版本号、依赖项、接口连线、采样周期、日志级别、随机种子等都应固化，并与执行产物一起归档。这样即使半年后回看，也能解释当时的“通过”意味着什么。

5.3. 质量门禁与评估指标：用事实驱动取舍

很多团队谈质量门禁，最后变成“卡流程”。MBTD 的门禁更应该围绕事实：追踪是否完整、覆盖缺口是否可解释、回归集合是否合理、失败是否可定位。本文给出一组较为常用的指标口径：资产指标(关键需求追踪率、关键转移断言标注率、断言库复用率)、效率指标(变更回归节省的执行时长、用例包维护工时)、质量指标(缺陷发现率、定位平均耗时、回流率)以及覆盖指标(高风险缺口数量及其连续存在的迭代数)。门禁规则应当允许裁剪与解释：例如允许少量缺口存在，但必须有风险评审结论和补齐计划；允许软判定波动，但应输出趋势与证据。这样，门禁就会从“拦人”变成“帮助团队更快对齐”。

在持续集成编排上，建议把回归分层：冒烟回归覆盖关键模式与关键接口；变更回归覆盖影响域用

例包；全量回归用于里程碑或发布前。冒烟与变更回归应成为日常门禁，全量回归则用于兜底。另外，建议单独管理“易波动用例”(Flaky Cases)：对偶发失败的用例包先进入隔离区，输出更多证据(例如更高频采样或更完整日志)，在定位后再回归到门禁集合。这样既能保持门禁的可信度，也不会因为偶发噪声把整个流程拖垮。

6. 结论与展望

本文以工程可落地为导向，对基于模型的测试设计进行了体系化讨论。核心观点可以概括为三点：其一，MBTD 首先是一种资产组织方式，模型必须具备可测试语义(触发、观测、断言)；其二，交付物应从“用例文本”升级为“用例包”，把步骤、数据、Oracle 与环境封装成可复现闭包；其三，覆盖度量与风险权重应共同驱动用例选择，并通过执行反馈持续修正剖面与差距清单。从实践角度看，MBTD 的短期收益往往体现在变更回归与定位效率上；长期收益则来自断言库、运行剖面与追踪矩阵的积累。

后续研究与实践仍有不少难点值得继续推进。其一是语义一致性：模型、仿真与实现之间如何保持一致的时序与单位语义；其二是规模复杂度：状态空间与连续变量耦合导致覆盖与选择问题迅速变大，增量切片与依赖图方法可能是可行方向；其三是 Oracle 质量：非功能与间歇性问题的判定仍然困难，需要更好的观测设计与统计判定方法。同时，学习型方法在探索优先级方面有潜力，但工程上必须兼顾可解释性与可审计性。

最后补一句更务实的结论：MBTD 的价值很大一部分来自“坚持”。它不是一次性导入某个工具就能完成的事情，而是在每次迭代里把需求写得更可判定一点、把模型标注得更可执行一点、把用例包做得更可复现一点、把证据链存得更完整一点。这些积累不会在一两周内立刻变成 KPI，但往往会在一次重大变更或一次关键故障回溯时体现出差距。

参考文献

[1] Utting, M. and Legeard, B. (2007) Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann.

[2] ISO/IEC/IEEE 15288:2015 (2015) Systems and Software Engineering—System Life Cycle Processes. ISO/IEC/IEEE.

[3] Object Management Group (OMG) (2019) OMG Systems Modeling Language (SysML) Version 1.6.

[4] ISO/IEC/IEEE 29119-1:2013 (2013) Software and Systems Engineering—Software Testing—Part 1: Concepts and Definitions. ISO/IEC/IEEE.

[5] ISO/IEC/IEEE 29148:2018 (2018) Systems and Software Engineering—Requirements Engineering. ISO/IEC/IEEE.

[6] Object Management Group (OMG) (2013) UML Testing Profile (UTP) Specification Version 1.2.

[7] Cederbladh, J., Cicchetti, A. and Suryadevara, J. (2024) Early Validation and Verification of System Behaviour in Model-Based Systems Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology*, **33**, 1-67. https://doi.org/10.1145/3631976

[8] Apvrille, L., De Saqui-Sannes, P., Hotescu, O. and Calvino, A. (2022) SysML Models Verification Relying on Dependency Graphs. *Proceedings of the* 10*th International Conference on Model-Driven Engineering and Software Development*, 6-8 February 2022, 174-181. https://doi.org/10.5220/0010792900003119

[9] Koo, J., Saumya, C., Kulkarni, M. and Bagchi, S. (2019) PySE: Automatic Worst-Case Test Generation by Reinforcement Learning. 2019 12*th IEEE Conference on Software Testing*, *Validation and Verification* (*ICST*), Xi’an, 22-27 April 2019, 136-147. https://doi.org/10.1109/icst.2019.00023

[10] Hansen, S.T., Gomes, C.Â.G., Najafi, M., Sommer, T., Blesken, M., Zacharias, I., *et al.* (2022) The FMI 3.0 Standard Interface for Clocked and Scheduled Simulations. *Electronics*, **11**, Article 3635. https://doi.org/10.3390/electronics11213635