南京大學

NANJING UNIVERSITY

研 究 生 毕 业 论 文

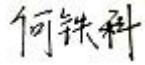
（申请工程硕士学位）

论 文 题 目	面向 Solidity 语言的变异测试系统设计与实现
作 者 姓 名	巫浩然
专 业 名 称	工程硕士（软件工程领域）
研 究 方 向	软件工程
指 导 教 师	刘嘉 副教授 何铁科 助理研究员

2020 年 5 月 23 日

学　　号：**MF1832175**

论文答辩日期：**2020 年 5 月 23 日**

指 导 教 师：  **（签字）**

Design and Implementation of Mutation Testing System for Solidity Language

By

Haoran Wu

Supervised by

Associate Professor **Jia Liu**

Research Associate **Tieke He**

A Thesis

Submitted to the Software Institute

and the Graduate School

of Nanjing University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Engineering

Software Institute

May 2020

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目： 面向 Solidity 语言的变异测试系统设计与实现

工程硕士（软件工程领域） 专业 2020 级硕士生姓名： 巫浩然

指导教师（姓名、职称）： 刘嘉 副教授 何铁科 助理研究员

摘 要

Solidity 是一种面向以太坊智能合约（Ethereum Smart Contract，ESC）的高级编程语言，用于在以太坊网络上实现智能合约。以太坊是当前最流行的智能合约开发与运行平台，已有超过百万款智能合约在以太坊中部署。然而研究表明，约 90% 的智能合约中存在缺陷，且由于区块链的不可修改性，这些缺陷难以被修复。因此，智能合约在部署前要尽可能充分地测试，这对保障以太坊用户资金安全以及构建可持续的以太坊生态具有重要意义。

变异测试是一种衡量测试套件充分性的软件测试方法，它通过将故障注入到源程序中来检查测试套件揭示这些故障的能力。变异测试已被证明适用于各种编程语言及编程范式，因此采用变异测试来评估 ESC 测试的充分性是可行的。然而，现有变异算子无法覆盖 ESC 特有的缺陷，为了提高变异测试的效果，需要针对 ESC 编程语言 Solidity 设计新的变异算子。本文设计并实现了一个面向 Solidity 语言的变异测试系统。具体来说，本文从关键字、全局变量/函数、异常检测及合约漏洞等方面研究了 Solidity 的语言特性，提出了 16 个 Solidity 特殊变异算子。随后，基于 Spring boot 框架实现了一个面向 Solidity 语言的变异测试工具 MuSC。MuSC 能够在抽象语法树的层次上高效且准确地生成大量变异体，并基于 Truffle 框架实现合约的自动化部署及测试。此外，MuSC 还提供变异体展示、自定义测试链创建、测试报告生成等功能，具有较强的可用性。

本文在 4 个真实以太坊分布式应用（DApp）中的 26 个智能合约上进行了实证研究，来证实变异测试以及新变异算子的有效性。实验结果显示，基于变异测试的方法在缺陷检测率上优于基于覆盖的方法（变异得分 63.1 比 53.6），表明变异测试相比代码覆盖率可以更好的衡量 ESC 测试的充分性。此外，本文对 729 个真实 ESC 错误报告进行浏览分类，发现其中有 117 个与 Solidity 特殊变异算子有关，表明本文提出的新变异算子能够有效揭示真实缺陷。这些结果揭示了变异测试在 ESC 质量保障中的巨大潜力。

关键词： Solidity，区块链，以太坊智能合约，变异测试，变异算子

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Design and Implementation of Mutation Testing System for Solidity Language

SPECIALIZATION: Software Engineering

POSTGRADUATE: Haoran Wu

MENTOR: Associate Professor **Jia Liu**
Research Associate **Tieke He**

Abstract

Solidity is a high-level programming language for Ethereum Smart Contract (ESC), it is used to implement the Smart Contract on the Ethereum network. Ethereum is the most popular platform for the development and operation of Smart Contracts. More than one million smart contracts have been deployed in Ethereum. However, research shows that about 90% of Smart Contracts have defects, which are difficult to be fixed due to the immutability of blockchain. Therefore, the Smart Contract should be tested as fully as possible before deployment, which is of great significance to ensure the funds security of Ethereum users and build a sustainable Ethereum ecosystem.

At present, the industry usually uses code coverage to measure the quality of testing, but coverage can not measure the error detection ability of test suite, mutation testing makes up for this disadvantage. Mutation testing is a fault driven software testing method, which checks the ability of test suite to reveal these faults by injecting faults into the source program. Mutation testing has been proved to be applicable to various programming languages and programming paradigms, so it is feasible to use mutation testing to evaluate the adequacy of ESC testing. The problem is that the existing mutation operators do not consider the characteristics of the ESC programming language Solidity, and cannot cover the unique defects of ESC. In order to improve the effect of mutation test for ESC, new mutation operators need to be designed. In this paper, we design and implement a mutation testing system for Solidity language. Specifically, we studies the language characteristics of solidity from the aspects of keywords, global variables/functions, exception detection and contract vulnerabilities, and

propose 16 special mutation operators for Solidity. Then, We implement a mutation testing tool named MuSC based on the Spring boot framework. MuSC can generate a large number of mutants efficiently and accurately at the level of abstract syntax tree, and supports automatic deployment and testing of contracts based on the Truffle framework. In addition, MuSC also provides many other functions such as mutant display, custom test chain creation, test report generation, etc.

In this paper, we conduct an empirical research on 28 smart contracts in 4 real Ethereum Distributed Applications (DApps) to confirm the effectiveness of mutation testing and new mutation operators. Experimental results show that the method based on mutation testing is superior to the coverage-based method in defect detection rate (63.1 vs. 53.6), indicating that mutation testing can better measure the adequacy of ESC testing than code coverage. In addition, we browses and classifies 729 real ESC error reports, and finds that 117 of them are related to the Solidity special mutation operator, indicating that the new mutation operator proposed in this paper can effectively reveal true defects. These results reveal the great potential of mutation testing in ESC quality assurance.

Keywords: Solidity, blockchain, Ethereum smart contract, mutation testing, mutation operator

目录

表 目 录

图 目 录

第一章　绪论

1.1　课题背景和意义

2008 年，中本聪发表文章《Bitcoin: A peer-to-peer electronic cash system》[1]，首次提出了区块链的概念。区块链具有去中心化、不可篡改、可以追溯、公开透明等特征，能够解决信息不对称问题，从而实现多个主体之间的协作信任与一致行动。基于这些特性，区块链被看作为下一代网络交互系统的核心技术，可以应用于医疗、能源、物联网等领域的建设 [2–4]。智能合约（Smart Contract, SC）是一种运行在区块链上的，控制用户数字资产的特殊程序。它可以由互不信任的节点组成的网络自动正确地执行，而无需外部可信机构 [5]。以太坊（Ethereum）是一个开源的具有智能合约功能的开放区块链平台，由 Vitalik Buterin 于 2014 年提出 [6]。作为当前最流行的智能合约开发与运行平台，以太坊上已部署了超过 100 万个智能合约。以太坊提供了四种智能合约开发语言：Solidity、Serpent、Mutan 和 LLL，其中 Solidity [1]是目前最流行的合约语言。

近年来，智能合约受到了越来越多的关注，其质量问题也日益突出。一方面，智能合约通常用于金融和信贷等安全关键领域，因此常常是攻击者的有吸引力的目标。另一方面，区块链的持久性决定了合约一旦部署上链就难以更改，因此合约一旦出现问题，其修复的代价极高。例如，2016 年 6 月，攻击者利用智能合约 splitDAO 函数中的重入漏洞盗取以太币，造成约 6000 万美元的损失并导致以太坊永久硬分叉 [7]。此外，与传统的软件系统相比，以太坊智能合约（Ethereum Smart Contract, ESC）的开发环境还不成熟，且开发人员大多缺乏经验，导致智能合约中缺陷普遍存在。研究表明：在已部署的合约中，约 90% 的智能合约隐含缺陷 [8]，为以太坊带来很大的安全隐患。因此，为了保障 ESC 的质量，必须对待部署的智能合约进行充分的测试。代码覆盖率通常被用来衡量测试的好坏，它通过计算测试用例执行的源代码的百分比来衡量测试的质量。但是代码覆盖率无法度量测试的缺陷检测能力，可以进行这种测量的一种技术是变异测试，它将缺陷注入到程序中，以测量测试用例检测这些缺陷的能力 [9]。

变异测试也称为变异分析，是一种对测试用例集的有效性、充分性进行评估的技术。在变异测试指导下，测试人员可以发现测试集的不足，从而创建缺陷检测能力更强的测试用例集 [10]。目前，变异测试已经适用于一系列编程语言，包括面向对象 [11]、面向函数 [12] 和面向方面 [13] 语言。因此，将变异测试应

[1]https://github.com/ethereum/solidity

用到 Solidity 语言的充分性是可取的。变异测试的效果主要取决于变异算子的设计，变异算子定义了如何将句法变化引入到原始程序中 [14]。设计变异算子的目标是尽可能完整地模拟潜在的威胁，如果采用不同的编程语言开发程序，通常需要不同的算子 [14]。然而，现有变异算子没有考虑到 Solidity 的语言特性，无法检测出一些 Solidity 特有的缺陷，为了提高变异测试的效果，需要设计新的变异算子。

为有效衡量以太坊智能合约测试的充分性，本文提出一种面向 Solidity 语言的变异测试系统，并设计实现了一个变异测试工具。具体来说，本文通过研究 Solidity 语言的特性，提出 16 个新的变异算子，涵盖了关键字、全局变量/函数、错误处理和合约漏洞等方面。然后基于 Spring boot、Nodejs、mysql 开发了面向 Solidity 语言的变异测试工具 MuSC，并在 MuSC 中实现了这些变异算子。MuSC 能够在抽象语法树的层次上进行进行缺陷注入，快速且正确地生成大量变异体，同时它还提供变异体查看、自定义测试链创建、自动化合约部署以及测试报告生成等功能。本文在来自 4 个以太坊分布式应用（Decentralized Application, DApp）的 26 个智能合约上进行实验来验证提出的变异测试系统的有效性。实验表明，（1）基于变异测试的方法在缺陷检测率上优于基于覆盖的方法（96.01% 比 55.68%）；（2）新提出的 16 个变异算子有效地揭示了真实缺陷（117 个真实缺陷报告与之相关）。这些结果显示了利用变异测试来保障以太智能合约质量的巨大潜力。

1.2　国内外研究现状

本文研究面向专用于以太坊智能合约的 Solidity 变异测试系统，主要涉及变异测试和智能合约测试两个方向，分别介绍国内外研究现状。

1.2.1　变异测试

变异测试是一种缺陷驱动的软件测试方法，用于帮助测试人员发现测试工作中的不足，改进和优化测试数据集。变异测试的思想最早由 DeMillo 等人提出 [15]，随后 Offutt 和其他人对此进行了广泛的探索 [16]。围绕本文中心，本节从编程语言、应用领域、智能合约等三个方面来介绍变异测试研究现状。

在面向不同编程语言的变异测试领域，Budd 等人针对面向结构语言 Fortran IV 首次设计了变异算子，并开发出变异测试工具 PIMS [17]。Offutt 等人在 Budd 的成果上进一步研究，针对 Fortran77 设计了 22 种变异算子，并在变异测试工具 Mothra 中实现了这些变异算子 [18]。C 语言作为一个代表性的面向结构编程

语言，也受到了研究人员的关注。Agrawal 等人针对 ANSI C 标准设计了 77 种变异算子，共分为变量变异、常量变异、操作符变异和语句变异四类 [19]。在面向对象编程语言领域，Kim 等人针对 Java 设计了 20 种变异算子，共分为类型/变量、名称、类和接口声明、语句块、表达式和其他六类 [20]。Ma 等人以面向对象缺陷模型为基础，设计出 24 种 Java 变异算子，并在变异测试工具 MuJava 中实现这些变异算子 [21]。Mirshokraie 等人针对 JavaScript 语言提出了一组特定于 Web 应用程序的变异算子，并实现了变异测试工具 MUTANDIS [22]。

在变异测试的应用领域方面，变异测试已成功应用于 Android 应用、数据库应用及 Web 服务测试等领域。在 Android 应用领域，Deng 等人定义了 11 种专用于 Adroid 应用程序的变异算子，涵盖了 XML 文件布局，事件驱动性质以及 Activity 生命周期结构等方面 [23]。在数据库应用领域，Chan 等人首先以增强关系实体模型为基础，针对全部 SQL 语句设计了 7 种变异算子 [24]。随后 Tuya 等人针对 SQL 查询语句设计了一组新变异算子，分为针对 SQL 语句、针对条件和表达式中操作符、针对标识符、处理控制四类，并在变异测试工具 SQLMutation 中实现了这些变异算子 [25]。在 Web 服务测试领域，Lee 等人首次对 Web 服务应用变异测试。他们提出交互规约模型对 Web 组件间的交互关系进行形式化描述，并以此为基础，提出一组用于 XML 数据变异的变异算子 [26]。Xu 等人提出新的 XML 数据变异策略，该策略通过对 XML Schema 进行搅动来生成无效 XML 数据，并在此基础上提出 7 种变异算子 [27]。

在智能合约变异测试领域，Joran 等人针对 Solidity 语言提出了两种新的变异算子，将 “+=” 替换为 “=+” 和删除函数修饰符，并实现了一个变异测试工具原型 Vertigo。随后他们在两个智能合约上进行了实验，表明变异测试可用于评估以太坊智能合约测试用例集 [28]。Pieter 等人考虑到以太坊地址变量以及变量存储位置，将特殊变异算子扩充到四个，同时提出使用 Gas 消耗的数量作为衡量变异体杀死与否的条件 [29]。此外，GitHub 上目前可获取两种能够用于智能合约的变异测试工具，eth-mutants [2]和 universalmutator [3]。然而，eth-mutants 只实现了边界条件变异算子，即只能用 “<=” 和 “>=” 来替换 “<” 和 “>”。universalmutator 是一个基于正则表达式的工具，可以对源码直接进行变异，缺点是可能产生大量无效的变异体。

当前针对以太坊智能合约变异测试的研究工作的一个主要不足之处在于没有对 Solidity 语言特性进行系统研究，因此所采用的变异算子无法覆盖 Solidity 中特有的缺陷，无法满足实际应用的要求。

[2]https://github.com/federicobond/eth-mutants

[3]https://github.com/agroce/universalmutator

1.2.2　智能合约测试

目前，智能合约测试的研究主要集中在智能合约测试数据生成、智能合约缺陷检测检测等领域、智能合约分析与优化等三个方面。本节分别针对这三个领域介绍国内外研究现状。

在智能合约测试数据生成领域，Luu 等人首先将符号执行技术用于智能合约测试数据生成，设计实现工具 Oyente [4]来检测重入攻击、交易顺序依赖、时间戳依赖、误操作异常等四种类型漏洞 [30]。Albert 等人基于 Oyente 设计实现 EthIR 框架，用以提供规则的以太坊智能合约中间表示，为开展以太坊智能合约属性分析提供了便利 [31]。之后，Albert 与 Correas 等人共同实现了 SAFEVM 工具，SAFEVM 基于 Oyente 将待测合约转化为控制流图，同时通过 EthIR 将其转化为更高层次的规则表示，从而将以太坊智能合约转化为 C 语言程序后再进行安全检测 [32]。为提高智能合约漏洞检测效率，Brent 等人实现了可将以太坊智能合约字节码转换为语义逻辑关系符的 Vandal 工具，并通过 Souffle 语言来检测合约是否包含漏洞 [33]。部分研究人员还提出了基于模糊测试的生成方法，Liu 等人基于模糊测试引擎设计并实现了一个以太坊智能合约测试生成工具 ReGuard，可以用于检测智能合约中的重入漏洞缺陷 [34]。Wang 等人在模糊测试生成的基础上提出了基于多目标优化的合约测试生成方法，在保持最大化覆盖率和最小化测试时间的同时，将 Gas 成本降到最低 [35]。

在智能合约缺陷检测领域，Sergey 等人对智能合约中的多交易行为与传统并发程序存在的共享内存访问问题进行了比较分析，总结了智能合约潜在的并发缺陷类型 [36]。Gao 等人设计实现了以太坊智能合约 overflow 缺陷检测工具 EasyFlow，在以太坊虚拟机层通过污点分析手段来发掘潜在的 overflow 缺陷模式 [37]。Grech 等人关注于以太坊智能合约中的 out-of-gas 漏洞，并提出一种结合基于控制流分析的反编译技术和陈述性程序结构查询技术的静态程序分析技术进行检测 [38]。Jiang 等人针对智能合约中的特殊类型漏洞 [39] 定义了一系列测试预言规则，并结合模糊测试技术自动检测合约中的安全漏洞缺陷 [40]。付梦琳等人使用符号执行技术辅助模糊测试，提高了模糊测试代码的覆盖率并缓解了符号执行的路径爆炸问题 [41]。sankov 等人使用领域特征语言来描述智能合约属性，在通过符号执行精确获取智能合约语义的基础上，通过判断语义是否违反来判断合约是否实现相关属性，从而分析合约的安全性 [42]。

在智能合约分析与优化领域化，Kolluri 等人归纳并总结了存在于区块链智能合约中的事件序列问题（event-ordering bugs），设计并实现了 EthRacer 工具进

[4]https://github.com/melonproject/oyente

行检测 [43]。特别的，针对事件序列组合带来的路径及状态组合爆炸问题，他们进一步使用了偏序约简技术和动态符号执行技术进行了性能优化。Kalra 等人实现了 Solidity 字节码到 LLVM 字节码的转换，并再此基础上结合抽象解释、符号模型检查以及限制性规则语言对以太坊智能合约进行形式化验证 [44]。Greth 等人提供了 Gigahorse 工具链，用于将以太坊智能合约字节码反编译为三地址码，便于研究人员在没有源码的场景下有效的开展程序依赖性分析 [45]。另外，由于以太坊特殊的 Gas 机制，每笔交易均需要消耗一定 Gas，因此从 Gas 机制出发对智能合约进行分析尤为关键。Chen 等人首先研究了导致智能合约燃油使用不当的代码模式，进而开发了工具 GasReducer 来实现缺陷代码的定位 [46]。Marescotti 等人首先对以太坊燃油耗费上界问题进行了研究，他们遍历分析函数中每一条路径的燃油消耗，同时考虑了协议版本、编译器版本以及智能合约编程语言类型的影响，实现对燃油耗费上界的有效预测 [47]。Jang 和 Kejsi 等人则从区块链信息中选择比特币的供需关系密切相关的特征，用于训练模型，以提高最新比特币定价过程的预测性能 [48]。

1.3　本文主要工作

本文设计并实现了一个面向 Solidity 语言的变异测试系统，主要工作如下：

在方法上，本文将变异测试的思想引入以太坊智能合约测试领域，通过变异测试来评估智能合约测试的充分性。为了确保变异测试能够涵盖智能合约的特性，本文从关键字、全局变量和函数、异常检测以及合约漏洞等方面入手，针对以太坊智能合约语言 Solidity 设计了一组全新的变异算子。接着本文通过对已有变异测试系统和以太坊智能合约测试方法的分析，提出了一种面向以太坊智能合约的变异测试框架，能够更好地保障以太坊智能合约的质量。

在系统上，本文在理论研究的基础上，实现了一个面向以太坊 Solidity 语言的变异测试工具——MuSC。MuSC 中实现了一系列传统以及 Solidity 特殊变异算子，可以在 AST（抽象语法树）级别上高效且精确地执行变异操作，并提供变异体展示、自定义测试链创建、合约自动化部署、测试报告生成等功能。此外，MuSC 还提供了一组简洁易用的图形用户界面，这些特点使 MuSC 成为以太坊智能合约开发人员的一个理想测试工具。

在实证研究上，本文在 4 个真实以太坊分布式应用中的 26 个智能合约上进行了实证研究，以评估变异测试的有效性。实验结果表明：（1）基于变异测试的方法在缺陷检测率上优于基于覆盖的方法（变异得分 63.1 比 53.6），这表明变异测试可以有效衡量衡量 ESC 测试套件的充分性；（2）通过查看 ESC 缺陷报告发

现 729 个真实的缺陷报告中有 117 个与新提出的算子相关，这表明本文提出的面向 Solidity 变异算子能够有效地揭示真实的缺陷。这些结果都体现了变异测试在以太坊智能合约质量保障中的巨大潜力。

1.4　本文组织结构

本文共分为六个章节，组织结构如下：

第一章，绪论。首先介绍本文研究背景并阐述该研究的重要意义，其次介绍了变异测试和智能合约测试的国内外研究现状，再次针对存在的问题说明本文的主要工作并总结本文贡献，最后给出本文的组织结构。

第二章，相关技术概述。首先介绍了区块链和智能合约的基本概念，其次详细介绍了以太坊智能合约的特点以及部署/执行过程，接着介绍了变异测试的定义、两个假设一般流程等基本知识，最后给出面向 JavaScript 的变异算子，为后文变异算子的设计做铺垫。

第三章，Solidity 变异测试。首先介绍了 Solidity 语言的特性；然后从变异算子的角度入手，针对关键字、全局变量及函数、错误处理和安全漏洞四个方面设计 Solidity 专用变异算子；最后从测试流程的角度入手，介绍了面向 Solidity 语言的变异测试完整流程。

第四章，系统的需求分析与概要设计。首先根据已有知识储备和研究现状对系统进行涉众和用例分析，其次，以此为基础确定功能与非功能需求，再次对系统进行概要设计并给出项目的整体架构，最后介绍 4+1 视图以及数据库设计。

第五章，详细设计和实现。首先详细介绍针对 Solidity 语言设计的新变异算子，其次根据概要设计对项目中的各个模块进行了详细设计，对每个模块都给出了其架构设计、核心类图、顺序图以及关键代码，最后以截图的方式展示系统各页面和测试报告。

第六章，系统的实验评估与分析。首先给出本文的研究问题，其次介绍实验对象，再次详细介绍了针对实验问题所设计的两组实验，最后对实验结果进行研究分析并得出结论。

第七章，总结与展望。总结实现系统与完成论文期间所做的工作，并展望项目的后续研究工作。

第二章　相关技术概述

本章主要介绍与系统相关的以太坊和变异测试两类技术。介绍以太坊时，首先对区块链和智能合约的基本概念进行简要介绍，然后重点介绍以太坊智能合约。介绍变异测试时，首先介绍变异测试的基本概念，然后详细介绍 JavaScript 变异算子，为后文变异算子设计作铺垫。

2.1　以太坊

2.1.1　区块链概述

区块链可以看作为一系列带有时间戳的不可变交易记录，这些交易记录被封装在一个个块中，并使用哈希值相互绑定。当一个新的区块添加到区块链中时，使用前一个区块的内容生成的哈希值作为引用来链接到前一区块，从而保证链条中的每一个块都是可追溯的。区块链中的每一笔交易均需要通过网络上的多个计算机节点进行验证，这些计算机协同工作，以确保每个交易在添加到区块链前都是有效的。区块链的结构决定了它具有去中心化、不可篡改性、匿名性、可验证性等特点 [49]。1）去中心化。区块链真正实现了点对点交易而无需中央可信机构的认证，从而降低了交易的额外消耗，且不会受到中央机构性能的影响。2）不可篡改性。区块链中的每个块都会广播给所有节点，并被各个节点所验证和记录，因此区块链中的数据篡改几乎是不可行的。3）匿名性。区块链系统中的每个用户按照一定规则自动生成单个或多个地址来与其他用户进行交易，其真实信息并未体现在其中，一定程度上保护了用户隐私。4）可验证性。由于区块链上的每笔交易都经过验证并带有时间戳记录，用户可以通过访问分布式网络中的任意节点轻松地验证和跟踪以前的交易。

区块链架构示意图如图2.1所示，区块链系统的架构通常由数据层、网络层、共识层、激励层、合约层和应用层六个层次组成。

(1) **数据层**：封装了数据区块及相关数据加密、时间戳等技术，是区块链最底层的数据结构。这一层描述了区块链的物理形式，即一个由规格相同的区块通过链式结构组成的链条。

(2) **网络层**：通过点对点组网、特定的信息传播协议和数据验证机制，保证各个节点可以平等地参与共识与记账，是区块链平台信息传输和去中心化特征的基础。

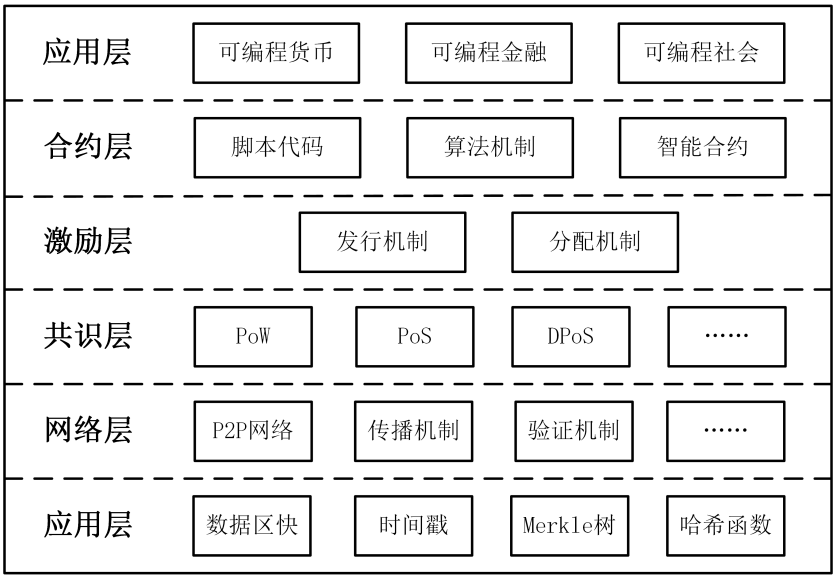


图 2.1: 区块链架构示意图

(3) **共识层**：封装了网络节点和各类共识算法，决定了各个节点的记账权利，包括记账权利的分配过程和理由。基于共识，区块链系统可以有效减少双花交易等欺诈行为，建立信任并保持区块链的完整性。

(4) **激励层**：负责设计合理的众包激励机制，使得每个节点最大化自身收益的个体行为与保障去中心化区块链系统的安全和稳定的整体目标相吻合。

(5) **合约层**：封装了区块链系统的各类脚本、算法和智能合约，可以在满足某个约束条件时自动执行，是实现区块链系统灵活编程和数据操作的基础。

(6) **应用层**：封装区块链的各种应用场景和案例，包括各式去中心化应用以及公共链、联盟链、私有链等不同的区块链应用模式。

2.1.2　智能合约概述

在区块链发展初期，其主要应用是比特币为代表的虚拟货币。此时区块链仅支持简单的交易，难以进行复杂的逻辑处理。随着区块链技术的发展，出现了图灵完备的智能合约编程语言，支持用户实现更为复杂且灵活的数据操作功能，使区块链能够支持金融和社会的诸多应用。智能合约这个概念由 Nick Szabo 于 1995 年首次提出，表示一套数字形式定义的承诺，以及执行这些承诺的相关协议，用于在分布式化环境中灵活地创建和管理数字资产 [50]。最初由于缺乏可编程合约的数字系统，智能合约并未引起研究人员的广泛关注。区块链技术的出

现使得智能合约成为了可能，区块链不仅支持合约编程，还具有去中心化、数据不可篡改、交易透明可追踪等优点，提供了履行智能合约承诺的信任环境。

智能合约中封装了若干预定义的触发条件、状态转换规则等。当合约的各签署方就合约内容、违约责任、外部数据源等达成一致，并对合约代码进行检查测试以确保无误后，将其以智能合约的形式部署上区块链，即可无需任何中心机构、自动化地执行合约。智能合约的诞生大大拓宽了区块链的应用场景，也标志着区块链进入 2.0 时代。智能合约可看作区块链的激活器，它赋予了静态的底层区块链可编程的机制。另外，智能合约可自动化运行的特性使它成为区块链网络中的软件代理人，使基于区块链技术构建去中心化应用成为可能，从而促进了区块链技术在各类分布式系统中的应用 [51]。

2.1.3　以太坊智能合约

以太坊是一个开放的区块链平台，允许任何人在平台中建立和使用通过区块链技术运行的去中心化应用。与之前的区块链系统不同，以太坊除了使用其内置加密货币以太币（ether）来实现数字货币交易功能外，还提供了图灵完备的编程语言以构建智能合约，从而首次将智能合约应用于区块链领域 [52]。目前，全世界有成千上万名开发者正在以太坊上构建应用程序，如加密货币钱包、金融应用、去中心化市场、游戏等。

以太坊中所有操作都是交易驱动的。以太坊的交易由 To、Value、Nonce、gasPrice、gasLimit、Data 和交易签名七个属性组成 [53]。其中，To 表示接受者的账户地址，Value 表示转账的以太币金额，Nonce 表示发送者对本次交易的计数，gasPrice 表示交易时的 gas 单价，gasLimit 表示执行该交易所允许消耗的最大 gas 数，Data 表示消息数据，交易签名表示发送者对本次交易的 ECDSA 签名。两个外部账户之间的交易可以看作一次简单的转账，而与合约账户有关的交易会触发智能合约的代码，实现预定义的功能。合约账户无法自己发起一个交易，只有在接收到一个交易之后 (从一个外部账户或另一个合约账户处)，为响应此交易而触发另一个交易。因此，以太坊上的操作总是始于某一外部账户发起的交易。

以太坊的数据模型与比特币不同，比特币采用基于交易的数据模型，而以太坊采用了基于账户的模型。在比特币中，每笔交易由交易输入和交易输出组成，并通过上笔交易的哈希指针构成以交易为节点的链表。针对某一地址，其交易输入与交易输出之差即为该账户的比特币余额。基于交易的模型可以有效防范双花、伪造等针对数字货币的攻击，但却无法快速查询账户余额以及其他状态。为了方便地处理更复杂的业务逻辑，以太坊采用了基于账户的模型 [53]。以太坊

中的账户分为外部账户（External Account, EA）和合约账户（Contract Account, CA）两类。外部账户由具有私钥的用户控制，用于表示一个普通账户的以太币余额，合约账户用于表示一个以太坊智能合约。每次将交易发送到合约账户时，以太坊网络中的每个节点都会对此合约中的代码执行过程进行评估。由于以太坊网络的去中心化性和透明性，与智能合约交互时较少出现腐败和保密等问题。

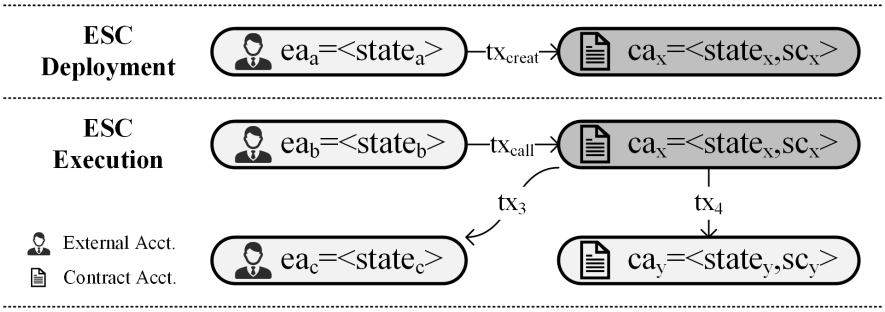


图 2.2: 以太坊智能合约部署及执行示意图

以太坊智能合约的部署和执行过程如图2.2所示。ea_a、ea_b 和 ea_c 表示三个外部账户，ca_x 和 ca_y 表示两个合约账户，tx 表示交易。本文使用四元组 $tx = \langle from, to, value, data \rangle$ 来表示一个交易，其中 $from$ 和 to 分别表示发送方账户和接收方账户的地址，value 表示要传输到的 Wei 的数量。当 $to = \varnothing$ 时，此次交易是一个合约创建交易，此时 $data$ 表示待部署的合约代码。智能合约部署和执行过程详细描述如下：

(1) **合约部署**。外部账户 ea_a 发起交易 $tx_{create} = \langle addr_a, \varnothing, value_1, sc_x \rangle$。一旦 tx_{create} 被打包到一个块中，并且该块被持久化到以太坊中，合约账户 ca_x 的创建和合约代码 sc_x 的部署完成。

(2) **合约执行**。外部账户 ea_b 发起交易 $tx_{call} = \langle addr_b, addr_x, value_2, input_x \rangle$。一旦 tx_{call} 被打包到一个块中并且该块被持久化到以太坊中，智能合约 sc_x 将 $input_x$ 作为输入并开始执行。在本图的示例中，执行智能合约 sc_x 时触发了两个附加交易 tx_3 和 tx_4。

2.2　变异测试

2.2.1　变异测试概述

变异测试是一种对测试套件的有效性和充分性进行评估的技术，能为开发或测试人员开展单元测试、集成测试等提供有效的帮助 [14]。变异测试的工作原

理是通过对源代码进行变异将故障引入系统，并通过对比源程序和变异程序在运行同一测试用例时的差异来评价测试套件的缺陷检测能力。这些所谓的变异是基于定义良好的变异运算符，它们要么模拟典型的应用程序错误，要么强制进行有效性测试，目标是帮助测试人员识别测试过程或测试套件中的限制。在变异测试过程中，一般利用与源程序差异极小的简单变异体来模拟程序中可能存在的各种缺陷。该方法的可行性主要基于两点假设:

(1) **熟练程序员假设 [15]**。该假设指出，由于熟练程序员编程经验丰富，即使其编写的代码中包含缺陷，也十分接近正确代码，仅需要做微小的修改即可修复代码中的缺陷。基于此假设，变异测试仅需要在原程序上做微小的修改即可模拟熟练程序员实际编程行为。

(2) **变异耦合效应假设 [15]**。该假设指出，如果测试用例能够杀死简单变异体，那么它也易于杀死更复杂的变异体。这里的简单变异体指的是在原程序上进行单一语法修改所形成的变异体，复杂变异体指的是对原程序进行多次单一语法修改所形成的变异体。耦合效应为变异测试时仅考虑简单变异体提供了理论依据，增加了变异测试的可行性和价值。

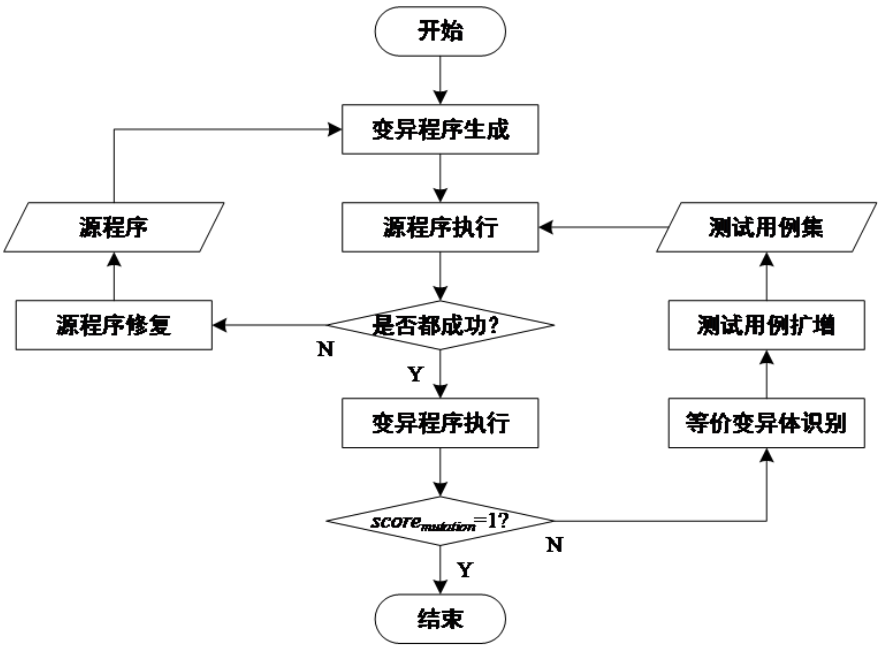


图 2.3: 变异测试基本流程图

图2.3给出了变异测试的基本流程图。给定源程序 P 和测试用例集 T，通过预先设计的变异算子生成一组变异程序 M。首先基于 T 运行 P，若存在测试用例使得 P 运行失败，则表明 P 存在缺陷，需要修复 P 并重复上述过程。若 P 均运行成功，则运行每一个变异程序。所有的变异程序运行结束后，若有变异程序存活，则表明：(1) 存在等价变异程序；或 (2) 当前测试用例设计不充分。此时，应当识别并去除当前存在的等价变异体，并扩增测试用例集，从而提高测试用例集的错误检测能力。不断重复上述过程，直至测试用例集变异得分为 1。

变异测试使用变异得分（Mutation Score）来评价测试用例集的缺陷检测能力。变异得分按照公式 $score_{mutation} = \frac{num_{killed}}{num_{total} - num_{equivalent}}$ 来计算，其中 num_{killed} 表示被杀死的变异体的数目，num_{total} 表示所有变异程序的数目，$nume_{quivalent}$ 表示等价变异程序的数目。变异得分 $score_{mutation}$ 的值介于 0 与 1 之间，数值越高，表明测试用例集杀死的变异程序越多，其缺陷检测能力越强，反之则越低。当 $score_{mutation}$ 的值为 0 时，表明测试用例集没有杀死任何一个变异程序；当 $score_{mutation}$ 的值为 1 时，表明测试用例集杀死了所有的非等价变异程序。

2.2.2　JavaScript 变异算子

表 2.1: JavaScript 变异算子

类型	变异算子	描述
通用变异算子	AOR	算术运算符替换
	AOI	算术运算符插入
	ROR	关系运算符替换
	COR	条件运算符替换
	LOR	逻辑运算符替换
	ASR	分配运算符替换
	SDL	语句删除
	RVR	返回值替换
	CSC	条件语句更改
特殊变异算子	ARV	添加/删除 var 关键字
	RSF	删除 replace A 中的全局搜索标志
	RIA	删除 parseInt 中的整数基本参数
	CSF	更改 setTimeout 函数
	RNU	用 null 替换 undefined
	RTK	删除 this 关键字
	RFS	用 (function()) 替换 (function()!==false)

变异测试已经被证明可以通过添加适当的变异算子来纳入新的的测试标准 [54–56]。因此，当变异测试应用到一个新领域时，设计有效的变异算子是最重要的任务之一 [23]。考虑到 Solidity 语言的语法与 JavaScript 相似，它们的变异算子也有共通之处，本文在设计 Solidity 变异算子前研究了面向 JavaScript 的变异算子 [22]。表 2.1展示了这些变异算子，包括 9 个通用变异算子和 7 个特殊变异算子。

通用变异算子中，AOR 又分为两类变异算子，AOR_B（二进制算术运算符替换）和 AOR_S（快捷算术运算符替换）。AOR_B 讲一个基本二进制算术运算符 (+，−，∗，/ 和 %）替换为另一个，而 AOR_S 将一个快捷算术运算符（op++，++op，op−− 和 −−op）替换为另一个。ROR 关系运算符替换变异算子将一个关系运算符（>，>=，<，<=，== 和 !=）替换为另一个，或将整个谓词部分更改为 *true* 和 *false*。COR 条件运算符替换变异算子将一个二进制条件运算符（&&，||，&，|）替换为另一个。ASR 分配运算符替换变异算子将一个分配运算符（+=，∗=，/=，%= 和 &=）替换为另一个。SDL 语句删除运算符可以通过注释删除任意一条可执行语句，CSC 条件语句更改运算符可以强制将条件判断中的语句更改为 *true* 或 *false*。这些通用变异算子均可直接应用于面向 Solidity 语言的变异测试中。

除了上述通用变异算子外，作者还提出了一组特定于 JavaScript 的变异算子。这些变异算子是通过研究从各种资源收集的 JavaScript 错误报告，基于程序员编程中的常见错误而设计的，涵盖了 JavaScript 特有关键字、全局变量和函数等方面。本文尝试将这些变异算子应用到 Solidity 变异测试中，发现除了 RTK（删除 this 关键字）变异算子外，其余变异算子均不适用于 Solidity 语言。因此，直接使用 JavaScript 变异算子来对 Solidity 语言进行变异是不合理的，要想提高 Solidity 变异测试的效果，必须针对该语言设计新的变异算子。

2.3　本章小结

本章对以太坊和变异测试的相关技术进行了介绍。介绍以太坊相关技术时，首先介绍区块链的基础知识，说明了区块链的特性、层次和智能合约的基本概念，然后介绍以太坊智能合约的特点以及其部署和执行过程，最后对以太坊智能合约编程语言 Solidity 的特性进行详细介绍。介绍变异测试的相关概念时，首先从变异测试基本假设、一般流程以及变异得分三个方面来介绍变异测试的基础知识，最后详细介绍面向 JavaScript 的变异算子，为后续面向 Solidity 的变异算子的设计打下基础。

第三章　Solidity 变异测试

本章主要介绍 Solidity 变异测试相关研究工作。首先介绍 Solidity 的语言特性，然后针对这些特性设计变异算子并给出每个变异算子的详细介绍，最后从从变异测试流程入手，给出了 Solidity 程序变异测试的完整流程。

3.1　Solidity 语言特性

```
1.  pragma solidity^0.6.0;
2.  contract HelloWorld{
3.   function helloWorld() external pure returns(string memory){
4.    return "Hello, World!";
5.   }
6.  }
```

图 3.1: Solidity 语言代码示例

Solidity 是一门面向合约的高级编程语言，设计的目的是能在以太坊虚拟机（EVM）上运行智能合约。这门语言主要受 JavaScript 语言的影响，因此语法与 JavaScript 相似。Solidity 将代码封装在 Contract 中，使用 Solidity 编译器将源码转化成能够在 EVM 上执行的机器级代码。图3.1展示了一个 Solidity 语言编写的”Hello World” 程序。其中，第 1 行表示代码使用的 Solidity 编译器版本为 0.6.0，第 2 行表示合约名为 HelloWorld，第 3-5 行是函数 helloWorld 的具体内容。尽管 JavaScript 程序员可以轻松读懂 Solidity 代码，但还是要注意到它们两者之间的差异，例如这段代码中有两个 JavaScript 中不存在的特殊关键字 external 和 pure。本文通过阅读 Solidity 文档[1]，从变量类型、变量单位、关键字、全局变量/函数、错误处理五个方面总结了 Solidity 的语言特性，具体如下：

(1) 变量类型。与 JavaScript 不同，Solidity 是一种静态的强类型语言，因此需要在编译期间指定变量类型，并且不允许隐式类型转换。此外，为了便于处理区块链数据，Solidity 中添加了一些新的变量类型（例如账户地址 address）。

(2) 货币及时间单位。以太坊使用其内在的货币以太币来激励网络内的计算 [57]。Solidity 中定义了一系列以太币货币单位，它们之间的转换为公式如下：1 ether = 1e3 finney = 1e6 szabo = 1e9 Gwei = 1e18 wei。类似的，Solidity 还支持一系列时间单位，包括 seconds, minutes, hours, days, weeks, years。

[1]https://solidity.readthedocs.io/en/v0.6.4/

表 3.1: Solidity 中特殊关键字

类型	关键字	描述
函数可见性	public	任何用户均可调用或访问
	external	仅外部用户可调用或访问
	internal	可被本合约及子合约调用或访问
	private	仅在本合约中调用或访问
函数状态	pure	函数无法查看或修改状态变量
	view	函数无法修改状态变量
	payable	函数可以接收以太币
变量存储位置	memory	生命周期仅限于一个函数调用
	storage	长期存储于区块链上
	calldata	用于外部函数参数
变量左值	delete	为变量赋予类型初始值

表 3.2: Solidity 中全局变量/函数

类型	全局变量/函数名	描述
区块和交易属性	block.number (uint)	当前区块号
	now (uint)	当前区块时间戳
	block.coinbase (address)	当前区块的矿工地址
	block.gaslimit (uint)	当前区块的 gas 限制
	msg.data (bytes)	交易的调用数据
	msg.value (uint)	随消息发送的 wei 数量
	msg.sender (address)	当前调用的交易发起者
	tx.origin (address)	完整调用链的交易发起者
	tx.gasprice (uint)	交易的 gas 价格
	blockhash(uint blockNumber)	指定区块的区块哈希
	gasleft() returns (uint)	剩余的 gas 数
ABI 编码函数	abi.encode(...) returns (bytes)	对给定参数进行编码
	abi.encodePacked(...) returns (bytes)	对给定参数执行紧打包编码
数学和密码学函数	addmod(uint x, uint y, uint k) returns (uint)	在任意精度下计算 (x + y) % k
	keccak256(...) returns (bytes32)	计算参数的 Keccak-256 哈希
	sha256(...) returns (bytes32)	计算参数的 SHA-256 哈希

(3) 关键字。Solidity 中有一些特殊的关键字，本文将这些关键字分为函数可见性、函数状态、变量存储位置和变量左值四类，具体内容见表 3.1。

(4) 全局变量/函数。Solidity 提供了一系列特有的全局变量和全局函数以便访问区块链数据或进行算术运算。本文将这些全局变量和函数进行了整理，分为区块和交易属性、ABI 编码函数以及数学和密码学函数三类，具体内容见表3.2。

(5) 错误处理。除了 assert 函数外，Solidity 还提供了另一个错误处理函数 require(*condition*)。当 *condition* 满足时，ESC 继续运行，否则停止并回退状态。函数 require 用于变量检查，assert 主要用于检测未知错误。

3.2　Solidity 变异算子

本节围绕3.1中总结的 Solidity 语言特性来设计 Solidity 特殊变异算子。为了使变异算子能够体现真实缺陷，本文从 GitHub、StackExchange 等网站上搜集了与以太坊智能合约相关的问题报告，并模拟这些真实缺陷设计了一组特定于 Solidity 语言的变异算子。这些变异算子根据其特征被分为关键字、全局变量及函数、错误处理、安全漏洞四类。为了便于理解，本文为每个变异算子都给出了一个示例，其中多数例子来自于 Liquid Democracy [2]。这是一个由智能合约实现的以太坊分布式应用程序，包含三个主要功能：vote(), delegate() and revoke()。

3.2.1　关键字变异算子

本文针对 Solidity 中含有的特殊关键字设计了六种变异算子。

(1) 函数状态关键字改变 (Function State Keyword Change, FSC)

Solidity 中的函数可以声明为 view，这意味着该函数不会修改智能合约的状态，即既不修改变量，也不发出交易。函数也可以声明为 pure，这意味着函数既不能读取也不能修改状态。也就是说，函数的返回值只取决于函数的参数。pure 函数不能访问 this.Balance 或 <address>.Balance，不能访问 block、tx、msg 的任何成员，不能发送或接收以太币，只能调用其他 pure 函数 [58]。

FSC 变异方法主要包括：1) 为一个未给定函数状态的函数添加状态关键字 view；2) 为一个未给定函数状态的函数添加关键字 pure；3) 将函数状态关键字 view 替换为 pure。需要注意的是 FSC 不会将 pure 替换为 view，原因是这样的变异体虽然引起了程序内部状态的改变，但这种改变并未传递到程序输出，因此无法被测试检查。本文基于可观察的测试输出判断变异体是否杀死，意味着

[2]https://github.com/DemocracyEarth/dapp

我们只考虑将错误传播到输出的强突变体。下面给出了两个 FSC 变异体的示例，第一个示例（liquiddomacy.sol 的第 236 行）将 createNewBalot () 函数的 view 关键字替换为 pure。因此，即使该函数内部代码实现了增加了 ballotData.number 的功能，它的实际值也不会改变。第二个变异体示例（liquiddomacy.sol 第 48 行）使函数 getvoctors 从 view 函数变为 pure 函数，导致其变得无法读取状态，这可能会引起编译错误。但是如果该变异体存活，则意味着开发人员使用了错误的函数关键字 **(ESE[3]#28504)**。

+ function createNewBallot(...) view private {
− function createNewBallot(...) pure private {

+ function getVoters() public view returns (...) {
− function getVoters() public pure returns (...) {

(2) 数据位置关键字替换 (Data Location Keyword Replacement, DLR)

Solidity 中的每一个引用类型，即数组和结构，都有一个附加的注释关键字，即“数据位置关键字”。Solidity 中总共有三种数据位置：memory（该位置的数据生存周期限制在一个函数调用内）、storage（存储状态变量的数据位置，在函数调用之间保持不变）和 calldata（包含函数参数的特殊数据位置，仅适用于外部函数调用参数）[59]。

数据位置不仅与数据的持久性相关，还与赋值的语义相关。例如，storage 和 memory（或 calldata）之间的数据分配总是会创建一个独立的副本，而从 memory 到 memory 的数据分配只创建引用 **(ESE#1231)**。智能合约中未设定数据位置关键字的本地存储变量可能被存储于意外的存储位置，可能导致有意或无意的漏洞。DLR 变异示例（liquiddomacy.sol 第 238 行）将 ballot data 的数据位置从 storage 替换为 memory，从而将 ballot 从引用转换为副本。

+ Ballot memory ballot = Ballot(from, ballotData...
− Ballot storage ballot = Ballot(from, ballotData...

(3) payable 关键字删除 (Payable Keyword Deletion, PKD)

payable 是 Solidity 中的一个函数修饰符，如果一个函数需要执行货币操作，它必须有一个 payable 关键字，这样它才能正常接收以太网货币。本变异算子的示例中，函数 deposit () 的 payable 关键字被删除。在变异之后，任何试图通过该函数发送 ether 的交易都将被拒绝。

+ function deposit(...) payable {
− function deposit(...) {

[3]Ethereum Stack Exchange: https://ethereum.stackexchange.com/

(4) delete 关键字删除 (Delete Keyword Deletion, DKD)

Solidity 使用 delete 关键字来释放空间。区块链作为一种公用资源，资源十分宝贵，为了鼓励主动对空间的回收，释放空间将会返还一些 gas。需要注意的是，delete a 只是将 a 重新赋值为初始值，并不是真正意义上的清除内存。对于整数而言，相当于 a=0；对于动态数组而言，相当于将 a 初始化为长度为 0 且不包含任何元素的数组；对于静态数组而言，相当于将 a 初始化为长度不变且不包含任何元素的数组。对数组中某个下标的元素进行 delete 操作时，该下标元素的值变为初始值，其后的元素并不会往前移动。下面的示例中展示了一个 DKD 突变体，该示例中使用注释的方式删去了 delete 语句，这将导致数组 A 中的最后一个元素没有被初始化。

```
+ delete A[length - 1];
- //delete A[length - 1];
```

3.2.2　全局变量/函数变异算子

Solidity 的全局命名空间中存在一些特殊的变量和函数 [60]，本文针对这些全局变量/函数设计了五个变异算子。

(1) 全局变量改变 (Global Variable Change, GVC)

Solidity 使用多个全局变量来表示区块链状态相关的信息。例如，now 表示当前区块的时间戳（block.timestamp 的别名），block.number 表示当前区块的编号，msg.value 表示发送到智能合约的 wei 的数量。GVC 操作符通过给全局变量分配一个符合其格式的随机值来更改它。例如，第一个 GVC 变异体示例将 block.number 更改为 1，而第二个变异体示例将其更改为随机值 85162。这两个突变体通过直接给全局变量赋固定值来创建不同的测试条件。

```
+ if (blocklock + BLOCK_HEIGHT > block.number) {
- if (blocklock + BLOCK_HEIGHT > 1) {
- if (blocklock + BLOCK_HEIGHT > 85162) {
```

(2) 数学函数替换 (Mathematical Functions Replacement, MFR)

智能合约常常需要处理数字，因此经常出现数学或加密函数，例如 addmod()、max256()、keccak256() 等。很多数学函数具有相同的参数类型和返回值类型，为了防止开发人员意外地使用错误的函数，本文设计了 MFR 操作符来替换这些函数。如 MFR 变异体示例将 addmod() 替换为 mulmod()。在 Solidity 中，addmod(x, y, k) 计算 $x + y\%k$，而 textttmulmod(x, y, k) 计算 $x * y\%k$。它们的函数输入和返回值均相同，但所表示的含义不同。

+ return addmod(x, y, k);
− return mulmod(x, y, k);

(3) 地址变量替换 (Address Variable Replacement, AVR)

如前所述，以太坊是一个面向交易的区块链平台，每笔交易都与账户地址相关。因此，地址是智能合约的重要组成部分。通常，在测试链上测试时，可以使用 address(n) 来引用测试链上节点的虚拟地址。此外，还有三个与 Solidity 中的 address 相关的全局变量：block.coinbase、msg.sender 和 tx.origin。

block.coinbase 代表打包当前区块的矿工的地址，msg.sender 是消息（当前调用）的发送者，而 texttttx.origin 是交易（完整调用链）的发起者。msg.sender 和 tx.origin 之间的不同之处在于，msg.sender 可以是协定，但 tx.origin 永远不能是协定。在一个简单的调用链中，$A \rightarrow B \rightarrow C \rightarrow D$，msg.senderIn D 将是 C，而 tx.origin 将始终是 A（**ESE#1891**）。AVR 运算符用另一个地址替换当前地址变量，如在 AVR 变异体示例（Liquid Democracy 第 181 行）中，msg.sender 被替换为 tx.origin 或 address(n)。要杀死这个变异体，需要设计一个测试用例，使 msg.sender 不等于 tx.origin。

+ createNewBallot(msg.sender, ipfsTitle);
− createNewBallot(tx.origin, ipfsTitle);
− createNewBallot(address(0), ipfsTitle);

(4) 以太币单位替换 (Ether Unit Replacement, EUR)

以太坊使用以太币 (Ether) 作为官方货币，除了可以作为虚拟货币直接进行交易之外，开发者们需要支付以太币来支撑应用的运行。不论是为以太网货币的传输构建交易，还是为令牌的发行调用智能合约，都要消耗以太币。Solidity 中以太币的单位有 wei、finney、szabo、Ether。它们之间的转换为 $1ether = 1 * 10^3 finney = 1 * 10^6 szabo = 1 * 10^18 wei$[61]。EUR 突变体示例将 finney 替换为 szabo，以模拟不经意间使用错误的以太币单元。

+ uint public constant MIN_INVEST = 10 finney;
− uint public constant MIN_INVEST = 10 szabo;

(5) 时间单位替换 (Time Unit Replacement, TUR)

除了以太币单位外，Solidity 中还有一个特殊的常用单位，即时间单位。时间是智能合约的特征之一，很多智能合约中都包含时间约束，作为合约触发的条件。solidity 中支持的时间单位有 seconds、minutes、hours、days 和 weeks，其中 seconds 是默认的单位 solidity time units。TUR 算子将一个时间单位后缀替换为另一个时间单位后缀，TUR 变异体示例如下所示。

+ if (now < startTime.add(2 days)) {
− if (now < startTime.add(2 hours)) {

3.2.3　错误处理变异算子

Solidity 中使用状态回滚异常来处理错误，即出现异常将回滚当前调用中对状态所做的所有更改。Solidity 提供函数 assert() 和 require() 来检查条件 [62]，本文针对这两种错误处理函数设计了四种变异算子。

(1) Require 语句删除 (Require Statement Deletion, RSD)

require() 函数通常用于确保输入或合约状态变量的条件有效，或验证调用外部合约的返回值。如果 require() 中的条件不满足，以太坊将撤消合约状态更改以检查由输入或外部组件引起的错误，同时可以提供错误消息。RSD 变异算子通过注释整个 Require 函数来起到删除的效果，变异体示例（liquiddomacy.sol 第 136 行）如下所示。

+ require(position < ballotData.number);
− //require(position < ballotData.number);

(2) Require 语句更改 (Require Statement Change, RSC)

与 RSD 运算符不同，RSC 运算符将 Require 函数中的语句强制更改为 false。该变异体的目的是使错误处理函数之后的语句永远不会执行，如下面的变异体示例所示，程序执行到 Require 处就会报错并强制回滚。

+ require(position < ballotData.number);
− require(false);

(3) Assert 语句删除 (Assert Statement Deletion, ASD)

assert() 函数的工作方式与 require() 类似，但 require() 用于检查输入上的条件，而 assert() 函数用于检测内部错误。Assert 语句与分析工具结合，可以评估智能合约是否按预期运行。下面的变异体示例通过注释的方式删除 assert() 函数，使其后的语句可以始终执行。

+ assert(balances[account] >= amount);
− //assert(balances[account] >= amount);

(4) Assert 语句更改 (Assert Statement Change, ASC)

ASC 变异算子与 RSC 类似，通过将 Assert 函数中的语句强制更改为 false 来使程序执行到此处强制发生异常。ASC 变异体示例如下所示。

+ assert(balances[account] >= amount);
− assert(false);

3.2.4　安全漏洞变异算子

安全漏洞是智能合约一个无法避免的问题。以太坊曾多次遭遇严重的安全漏洞问题，其中很多都是由仅一行代码所引起的。本文研究了数起著名的以太坊安全事件，并针对这些安全漏洞提出了 4 种变异算子。

(1) 传输函数替换 (Transfer Function Replacement, TFR)

以太坊智能合约的一个特点是合约之间可以进行外部调用。此外，以太坊的交易也不限于外部账户之间，合约帐户可以存储以太币并执行交易。如果一笔交易的目的地址是合约帐户且未调用合约中的任何函数时，将自动执行合约中的回退函数 (fallback function)。但如果合约代码中没有定义回退函数，则这笔交易将失败（并引发异常）。回退函数的定义在官方文档[4]中的描述如下：合约可以有一个未命名的函数，这个函数既不能有参数也不能有返回值。如果在一个到此合约的调用中，没有其他函数与给定的函数标识匹配（或没有提供调用数据），那么这个函数（fallback 函数）会被执行。除此之外，每当合约接收到以太币（没有附带任何数据），这个函数就会执行。

目前已知的很多与 Solidity 相关的安全问题都涉及回退函数，其中最著名的就是可重入漏洞 [39]。可重入漏洞与传输函数有关。solidity 中有三个传输函数：<address>.transfer()、<address>.send() 和 <address>.call.vale()。它们都可以用来发送以太币到一个地址，但有如下区别：

- <address>.transfer() 在交易失败时抛出异常并回滚状态，最多使用 2300gas 进行外部调用；
- <address>.send() 在交易失败时抛出异常并返回 false，最多使用 2300gas 进行外部调用；
- <address>.call.vale()() 在交易失败时仅返回 false 不抛出异常，可以使用全部的 gas 进行外部调用。

在使用传输函数向一个合约账户直接转账时，由于这个行为没有发送任何数据，所以接收合约总是会调用 fallback 函数。当使用 call.value()() 进行转账操作时，会将所有可用的 gas 提供给外部调用（即回退函数），如果此时回退函数中有递归调用，则会不断重复调用直到 gas 耗尽。为了更直观，本文使用一段示例代码来展示可重入性漏洞。

[4]https://solidity.readthedocs.io/en/v0.5.8/

```
1.  contract SimpleDAO {
2.    function deposit(address to) public payable {
3.      balances[msg.sender] += msg.value;
4.    }
5.    function queryCredit return (uint) {
6.      return credit[to];
7.    }
8.    functionwithdraw(uint amount) public payable {
9.      if(credit[msg.sender] >= amount) {
10.       msg.sender.call.value(amount)();
11.       credit[msg.sender]= amount;
12.     }
13.   }
14. }
15. contract Attack {
16.   SimpleDAO public dao = SimpleDAO(0x354...);
17.   function() payable { //fallback function
18.     dao.withdraw(dao.queryCredit(this));
19.   }
20. }
```

图 3.2: 重入漏洞代码示例

图3.2的展示了一个重入漏洞的示例代码。其中，SimpleDAO 是漏洞合约，Attack 是攻击合约。在本例中，攻击者通过恶意递归合约 Attack 来取出 SimpleDAO 中的所有以太币。攻击者先存一笔钱，然后调用取款方法。只要满足第 9 行中的 credit[msg.sender] >= amount，就可以成功执行 msg.sender.call.value(amount)()。由于传输函数的性质，在传输完成后会自动调用 Attack 合约的回退函数，因此会再次调用 withdraw 函数。因为此时 credit 还没有更新，仍然可以正常取款。这样就进入了一个递归循环：不断地从 SimpleDAO 中提取以太币，直到剩余的钱少于 *amount*。臭名昭著的 The DAO 事件就是由可重入漏洞引起的。2016 年 6 月，攻击者通过递归调用 splitDAO 函数从 The DAO 合约中窃取了价值 6000 万美元的以太币。为了避免此类漏洞，可以将扣款操作放在传输函数之前，或者使用 transfer() 来代替 call.value()()。这是因为 transfer() 只提供 2300 个 gas 用来传输，没有多余的气体来递归执行回退函数。为了模拟开发人员无意中使用错误的传输函数而导致的安全漏洞，本文设计了传输函数替换 (TFR) 变异算子来替换传输函数，下面是一个 TFR 变异体的示例。

+ msg.sender.send(msg.value);
− msg.sender.transfer(msg.value);
− msg.sender.call.value(msg.value)();

(2) 访问域关键字替换 (Access Domain Keyword Replacement, ADR)

在 Solidity 中有四种变量或函数访问域关键字：private、public、external 和 internal。合约中实例方法默认可见状态为 public，而合约实例变量的默认可见状态为 private。public 标记的函数或变量可以被任何账户调用或获取，可以是合约里的函数、外部用户或继承该合约里的函数；external 标记的函数只能从外部访问，不能被合约里的函数直接调用，但可以使用 this.func() 外部调用的方式调用该函数；private 标记的函数或变量只能在本合约中使用（注：这里的限制只是在代码层面，以太坊作为公链，任何人都能直接从链上获取合约的状态信息）；internal 一般用在合约继承中，父合约中被标记成 internal 的状态变量或函数可供子合约进行直接访问和调用（外部无法直接获取和调用）。

错误地使用访问域关键字会引起访问控制漏洞，这一漏洞曾经导致奇偶校验钱包（the Parity Wallet）事件。2017 年 7 月，由于合同可见性设置错误，Parity Multi-signature Wallet 合约损失约 3000 万美元。该漏洞是由于 initWallet 函数被错误地设置为 public 函数所致。攻击者可以通过调用 initWallet 函数来重新初始化钱包并覆盖先前的钱包所有者。通过对代码的分析，可以确定导致此次事件核心问题是超权限的函数调用，因此合约接口必须经过精心设计，并明确定义访问权限。本文设计访问域关键字替换（ADR）变异算子的目的是避免开发人员不小心使用错误的访问域关键字。ADR 突变体示例（liquiddomacy.sol 第 164 行）展示了由此变异算子生成的三个不同突变体。

+ function resetVoter() external {
− function resetVoter() internal {
− function resetVoter() public {
− function resetVoter() private {

(3) 值类型关键字替换 (Variable Type Keyword Replacement, VTR)

布尔型、整形、定长浮点、定长字节数组等类型被称为值类型，因为这些类型的变量将始终按值来传递。也就是说，当这些变量被用作函数参数或者用在赋值语句中时，总会进行值拷贝。与 JavaScript 不同，Solidity 是一种静态的强类型语言，因此需要在声明时指定每个变量的类型。Solidity 支持三种固定大小的类型：固定大小整数、定点数和固定大小字节数组，所有这些值类型都有一个限制范围。例如，int8（8 位有符号整数）可以存储 −128 到 127 之间的值，uint256

（256 位无符号整数）可以存储 0 到 $2^{256}-1$ 之间的值。当某些运算的结果超出支持的范围时，会发生溢出。

Solidity 中的整数分为各种范围的有符号整数和无符号整数，整数溢出的结果是最高位的丢失，这会引起整数溢出漏洞，如 batchOverflow 事件 [63]。除整数外，定点数和固定大小字节数组也同样有溢出的可能，本文针对这些类型定义了一系列值类型关键字替换 (VTR) 运算符。下面给出的变异体示例中将 uint256 替换为 uint8，若 block.number 的值大于 uint8 的最大值，则会导致无限循环。

\+ for(uint256 i=0; i < block.number; i++){
\- for(uint8 i=0; i < block.number; i++){

3.2.5　特殊变异算子小结

表 3.3: Solidity 特殊变异算子

类型	变异算子	描述
关键字	FSC	函数状态关键字更改
	DLR	数据位置关键字替换
	PKD	Payable 关键字删除
	DKD	Delete 关键字删除
全局变量/函数	GVC	全局变量更改
	MFR	数学函数替换
	AVR	地址变量替换
	EUR	以太币单位替换
	TUR	时间单位替换
错误处理	RSD	Require 语句删除
	RSC	Require 语句更改
	ASD	Assert 语句删除
	ASC	Assert 语句更改
合约漏洞	TFR	传输函数替换
	ADR	访问域关键字替换
	VTR	值类型关键字替换

综上所述，如表3.3所示，本文提出了 16 种 ESC 特定的变异运算符，其中四个与关键字有关，五个与全局变量/函数有关，四个与错误处理有关，另外三个与合约漏洞有关。每种变异算子都对应一种合约编程中出现的真实错误，运用这些变异算子可以在代码中模拟这些缺陷。

3.3　Solidity 变异测试流程

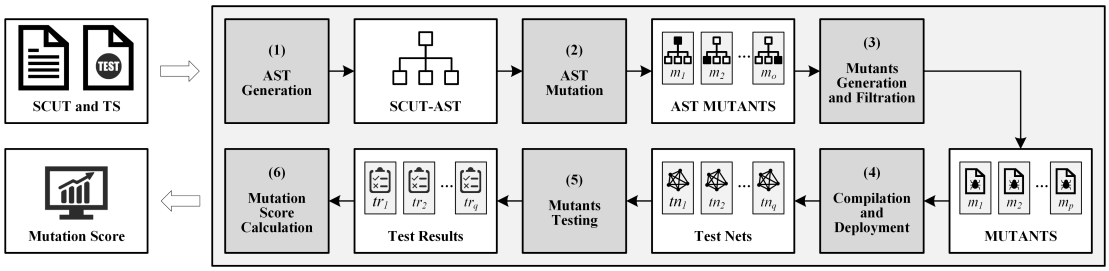


图 3.3: Solidity 变异测试流程

本文研究了传统变异测试流程，在此基础上提出了面向 Solidity 语言的变异测试流程。如图 3.3所示，系统的输入为被测智能合约（Smart Contract Under Test, SCUT）和一个包含 n 个测试用例的测试套件（Test Suite, TS），输出为一份测试报告。下面对本系统流程进行详细描述。

(1) **AST 生成（AST Generation）**。首先，SCUT 被解析为 AST 格式，以减少由于注释、空行等冗余语句而导致的变异精度损失。本系统使用 GitHub 上开源的 Solidity 解析器 solidity-parser-antlr [5]来生成 AST，它在 ANTLR4 语法基础上构建，解析结果准确。

(2) **AST 变异（AST Mutation）**。对 AST 格式的代码应用变异算子，生成一系列 AST 格式的变异体。由于变异操作是在 AST 格式上进行，因此需要面向 AST 实现变异算子。

(3) **变异体生成和过滤（Mutants Generation and Filtration）**。每一个 AST 格式的变异体将被转换成一个源文件版本，该版本与 SCUT 具有相似的语义，但是注入了一个预定义的错误。随后，进行等价性检查以过滤出等价变异。

(4) **编译和部署（Compilation and Deployment）**。ESC 在以太坊区块链上工作，其执行结果取决于区块链状态。因此，为了避免区块链状态影响测试结果，无论是 SCUT 还是每个源文件版本的变异体，系统都会在编译和部署之前构建一个初始状态相同的以太坊测试链。另外，有的变异体无法通过编译，系统会将这类变异体直接丢弃，不会体现在变异得分中。

[5]https://github.com/federicobond/solidity-parser-antlr

(5) **变异体检测（Mutants Testing）**。当 ESC 被成功部署后，即可执行 TS 中的测试用例。对于每个变异体，系统记录其每次测试的执行结果（即通过或失败）。给定一个变异体 m_i 和一个测试用例 t_j，如果 t_j 在 m_i 上执行失败，则 m_i 被标记为被 t_j 杀死，否则 m_i 存活。

(6) **变异得分计算（Mutation Score Calculation）**。在所有的变异体检测完成后，可以得到存活的变异体数以及被杀死的变异体数。此时，按照被杀死的变异体数与全部变异体数的百分比来计算变异得分。

3.4　本章小结

本章介绍了面向 Solidity 的变异测试的研究工作。首先从变量类型、关键字、全局变量/函数、错误处理等方面介绍 Solidity 语言的特性，为设计新的变异算子打下基础。接着，针对这些特性设计了 16 个 Solidity 变异算子，这些变异算子可以模拟合约编程中实际出现的错误。最后，本文为 Solidity 语言设计了完整的变异测试流程，为后文系统的需求分析与概要设计打下基础。

第四章　　需求分析与概要设计

本章介绍面向 Solidity 的变异测试系统的需求分析与概要设计。首先按用例分析、功能性需求和非功能性需求的顺序对系统进行需求分析，随后从系统架构、4+1 视图以及持久化设计三个方面对系统进行概要设计。

4.1　需求分析

4.1.1　用例分析

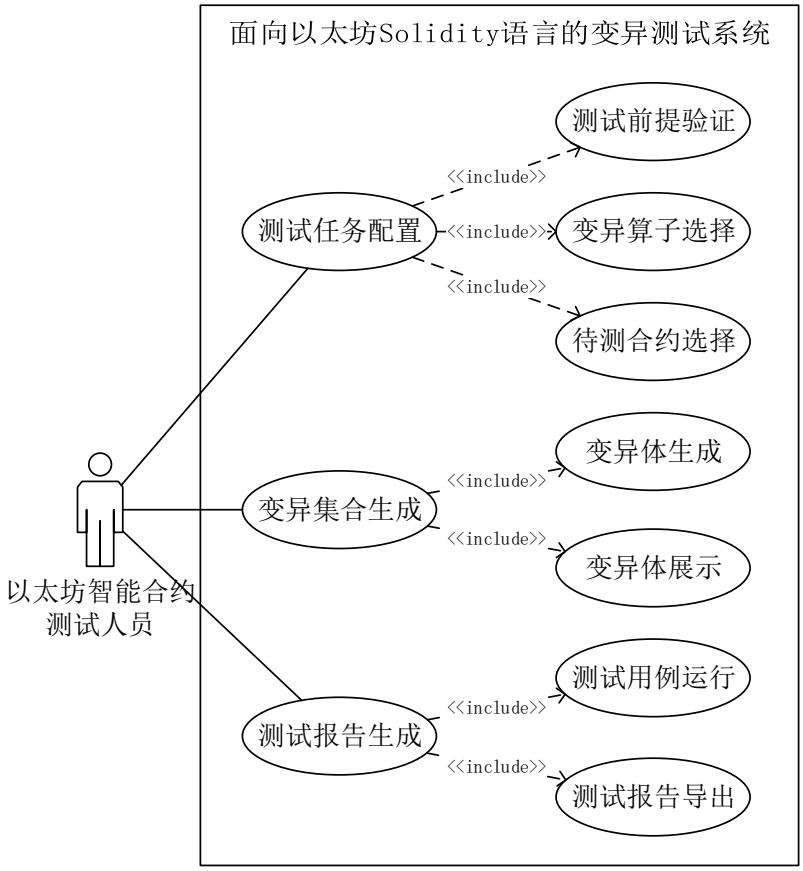


图 4.1: 系统用例图

图4.1展示了系统整体用例图。本系统作为一个面向以太坊 Solidity 语言的变异测试工具，用户一般为以太坊智能合约测试人员，主要可以分成测试任务配置、变异集合生成和测试报告生成三个用例。在测试任务配置用例，用户输入

待测项目路径，系统对该项目进行测试前提验证，通过后用户可以选择变异算子以及待测合约。在变异集合生成阶段，系统使用用户选择的变异算子对待测合约进行变异体生成，随后用户可以在前端查看生成的变异体。在测试报告生成阶段，用户选择使用默认或自定义测试链命令，系统会自动在测试链上部署原始合约及变异合约并运行测试套件，当所有变异体测试完毕后系统根据测试结果生成一份测试报告供用户导出。下面对三个用例进行详细的用例描述。

表 4.1: 测试任务配置用例描述

ID	UC1
名称	测试任务配置
参与者	以太坊智能合约测试人员
触发条件	以太坊智能合约测试人员启动本系统
前置条件	以太坊智能合约开发人员已完成项目开发并提供部分测试用例
后置条件	无
正常流程	1. 用户输入待测项目的本地路径 2. 用户点击 [项目检测] 按钮，系统自动运行项目中的测试用例 3. 测试用例全部通过，系统获取并展示项目中的合约文件 4. 用户选择要使用的变异算子 5. 用户选择待测试的合约文件
扩展流程	1a. 本地项目路径错误或非 Truffle 框架构建 1. 系统提示无效项目路径，请输入正确路径 3a. 项目中测试用例有报错 1. 系统提示测试用例无法全部通过

表4.1给出了测试任务配置的用例描述。测试任务配置主要包括测试前提验证、变异算子选择和待测合约选择三个子用例。测试前提验证用于验证输入项目路径是否基于 Truffle 框架构建且项目中的测试文件是否全部通过。待测项目必须基于 Truffle 的原因是这便于获取项目内合约文件以及测试文件的位置，并且后续测试用例运行时也需要使用框架自带的测试命令。测试文件必须全部通过的原因是本系统通过对比测试用例在原程序和变异体上的运行结果来判断变异体是否被杀死。如果测试前提均符合，系统会将项目内的合约名展示出来，用户可以在前端选择想要测试的合约文件以及想要使用的变异算子。

表4.2给出了变异集合生成的用例描述。在测试任务配置完成后，用户点击变异体生成按钮，系统使用选定的变异算子对待测合约生成变异体。变异体生成完成后，系统会自动跳转到变异体展示界面。在该界面中，用户可以按待测文件和变异算子来选择想要查看的变异体，变异体展示时在右侧对比显示原始合约，并将变异行标红，以便定位变异位置。

表 4.2: 变异集合生成用例描述

ID	UC1
名称	变异集合生成
参与者	以太坊智能合约测试人员
触发条件	以太坊智能合约测试人员点击 [变异体生成] 按钮
前置条件	以太坊智能合约测试人员已完成测试任务配置
后置条件	无
正常流程	1. 用户点击 [变异体生成] 按钮 2. 系统生成变异体并跳转到变异体展示页面 3. 用户选择原始合约文件 4. 系统展示该文件生成的所有变异体列表 5. 用户选择待某个变异体 6. 系统展示该变异体的内容，并将变异行标红
扩展流程	无

表 4.3: 测试报告生成用例描述

ID	UC1
名称	测试报告生成
参与者	以太坊智能合约测试人员
触发条件	以太坊智能合约测试人员切换到测试用例运行界面
前置条件	以太坊智能合约测试人员已完成变异体生成步骤
后置条件	无
正常流程	1. 用户选择执行全部变异体 2. 用户选择使用默认测试链命令 3. 用户点击 [开始运行] 按钮 4. 系统启动本地测试链，开始运行变异测试 5. 系统运行完毕，展示简要测试结果 6. 用户点击 [测试报告导出] 按钮，系统导出详细测试报告
扩展流程	1a. 用户选择自定义测试链并输入测试链启动命令 1. 系统使用用户输入的测试链部署变异体并执行变异测试 2a. 用户选择仅执行部分变异体 1. 系统仅对选型的变异体进行变异测试

表4.3给出了测试报告生成的用例描述。测试报告生成用例分为测试用例运行和测试报告导出两个子用例。在测试用例运行子用例中，用户首先切换到测试用例运行界面，接着选择使用默认测试链命令，最后点击开始运行按钮，系统即可开始运行变异测试。由于区块链状态会对合约运行结果产生影响，为了满足不同的测试需求，系统提供一个输入框以供用户输入自定义测试链命令。用户点击开始运行按钮后，系统会自动创建本地测试链，并将原始合约及变异体部署上链，随后基于 Truffle 框架在各个合约上执行测试用例。每一个变异体测试完成后，系统根据测试用例执行时的输出来判断该变异体是否被杀死。当所有变异体测试完成后，系统根据测试结果生成一份详细的测试报告，，显示总变异得分和每个变异体的测试结果。测试报告以 html 的形式展示，使用表格或高亮代码来显示不同的变异体，便于测试人员进一步分析。

4.1.2　功能性需求

根据用例分析，本系统可以分为测试任务配置、变异体生成与展示、测试报告生成三个模块。

测试任务配置。智能合约测试人员向系统输入本地待测 DApp 项的目路径，系统能够自行获取项目中的各个智能合约文件以及测试文件。在变异测试前，系统需要自行检验项目现有测试用例是否可以全部通过。如测试用例中包含错误，系统应弹出提示。在测试用例能够全部通过后，测试人员可以自行选择使用哪些变异算子，并针对哪些合约文件进行测试。

变异体生成与展示。在测试任务配置完成后，系统可以按照配置的参数进行变异操作。生成的变异体保存在本地，命名需体现原始文件名、变异算子名以及变异的行号。在变异体生成后，测试人员可以查看每个变异体。由于变异体数量较多，系统应将变异体按变异算子分类或按测试文件分类。测试人员选择某一变异体时，系统应在界面上展示变异体，并将变异行用醒目的颜色标注以便定位。为了验证变异操作是否正确，系统应同时展示原始合约文件来进行对比。

测试报告生成。测试报告生成模块包含了测试用例运行的完整流程。由于智能合约的测试需要部署在测试链上进行，因此在测试开始前需要确定测试链启动命令。系统支持使用默认测试链，同时为了满足不同的测试需求，系统还需支持用户输入自定义测试链启动命令。由于以太坊智能合约测试步骤较为繁琐，在每个变异体测试时系统应提供自动化测试链搭建、合约部署以及测试用例运行的功能。在所有变异体测试完成后，系统可以根据测试结果生成一份详细的测试报告。报告应包涵测试用例的变异得分及各个变异体运行结果。

4.1.3　非功能性需求

本系统目的是帮助以太坊智能合约测试人员进行测试套件充分性评估，从而提高以太坊智能合约的质量。为了便于使用，本系统需要满足以下非功能性需求。

性能。系统前端各个界面的响应时间不应高于 1 秒，系统执行变异体生成的时间不应超过 1 分钟，每个变异体执行变异测试的时间不应超过 1 分钟。

可靠性。变异测试的效果与变异体生成阶段的准确度直接相关，因此对系统可靠性有较大需求。系统可靠性还要求当用户输入异常数据或系统运行出现故障时有相应的提示信息或者操作。

易用性。由于本系统整体流程专业性较强，用户可能未接触过变异测试的概念，本系统需要对变异算子和使用方法提供详细的介绍说明以降低使用难度。同时，由于变异测试耗时可能长达数十分钟，为了提升用户体验，系统应在测试报告生成步骤给出当前正在运行的变异体信息以及整体运行进度。

可扩展性。由于以太坊和 Solidity 语言在不断的迭代更新，语言特性等可能会更改，因此对现有变异算子的修改和新增变异算子的添加有必要的需求。为了适应变更的版本，系统需要保证良好的可扩展性。

4.2　概要设计

概要设计部分以上述需求分析总结出的功能性需求和非功能性需求为基础，对系统进行总体设计。首先介绍系统的整体架构，然后从 4+1 视图中的逻辑视图、进程视图以及开发视图三个角度详细描述系统，最后介绍系统的持久化设计。

4.2.1　系统架构

系统整体架构如图4.2所示。基于对系统需求以及实现技术的分析与预测，可以将系统整体分为三层：用户界面层、业务逻辑层和数据存储层。从而确定本系统基于 C/S（Client/Server）架构，设计模式为 MVC（Model View Controller）。用户界面层主要负责数据展示以及与用户的交互，前端使用 Html5+JavaScript 实现静态 UI 界面，并通过 JSON 数据的格式与后端 api 交互。该层主要通过客户端界面的 UI 控件等实现交互，并采用 JavaScript 编写的事件处理函数来接受响应以及传送请求。业务逻辑层的所有服务均使用 Spring boot 框架实现，选用 Spring boot 框架的原因是它的可扩展性强且支持众多服务组件。本系统中业务服务可以分为测试前提检测、变异算子选择、AST 解析、AST 变异、AST 还原、变异体

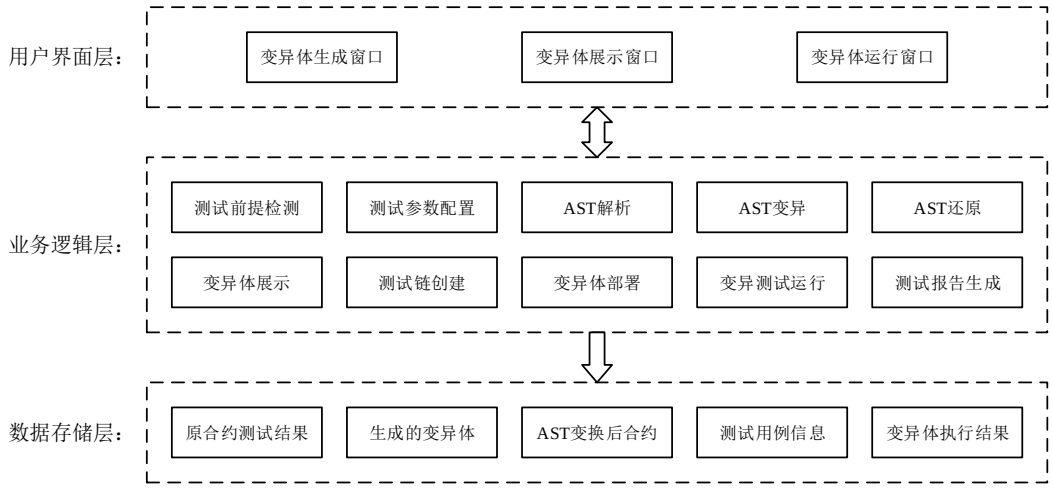


图 4.2: 系统架构图

展示、测试链创建、变异体部署、变异测试运行和测试报告生成。所有的业务服务均采用传统的分层设计，可以较大程度减少业务耦合且具有良好的可拓展性。数据存储层使用 mySQL 数据库存储持久化数据，使用 JDBC 连接数据库。其中生成的变异体保存在磁盘，其他的测试信息等数据存入数据库。

4.2.2　4+1 视图

本文采用“4+1”视图法 [64] 展示系统的架构设计，该方法由 Philippe Kruchten 于 1995 年提出，是目前软件架构设计领域的结构标准。本节分别从 “4+1 视图” 中的逻辑视图、进程视图和开发视图三个方面对系统的架构进行描述。

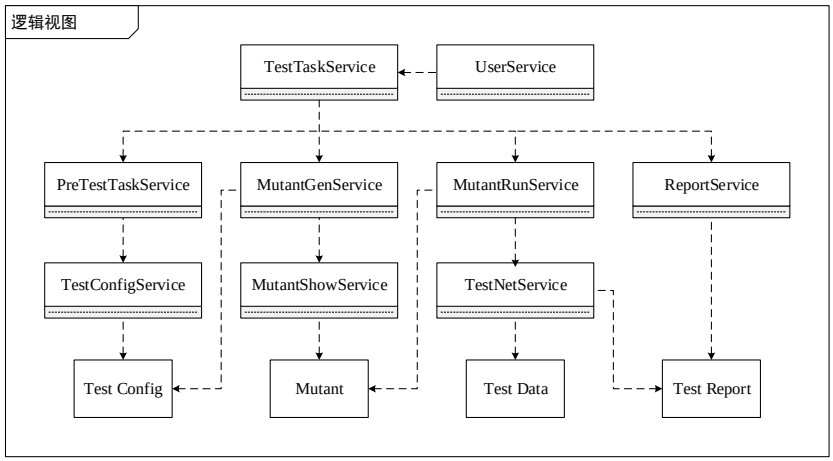


图 4.3: 逻辑视图

逻辑视图。图4.3展示了面向以太坊 Solidity 语言的变异测试系统的逻辑视图。逻辑视图从功能性需求上描述系统，通常以面向对象的方法为基础，将系统抽象为类或对象进行描述。本系统中的主要实体可以抽象为测试配置（Test Config）、变异体（Mutant）、测试数据（Test Data）和测试报告（Test Report）等，系统中主要服务可以抽象为用户服务（UserService）、测试任务服务（TestTaskService）、预测试任务服务（PreTestTaskService）、测试配置服务（TestConfigService）、变异体生成服务（MutantGenService）、变异体展示服务（MutantShowService）、变异体运行服务（MutantRenService）、测试链服务（TestNetService）和报告服务（ReportService）等。其中预测试任务服务主要提供待测项目的测试前提验证功能，测试任务服务提供变异算子和待测合约选择功能，变异体生成服务负责执行变异操作，变异体展示服务提供对生成的变异体的展示功能，变异体运行服务提供对所有变异体的测试功能，测试链服务提供默认或自定义测试链的创建功能，报告服务提供测试报告生成功能。

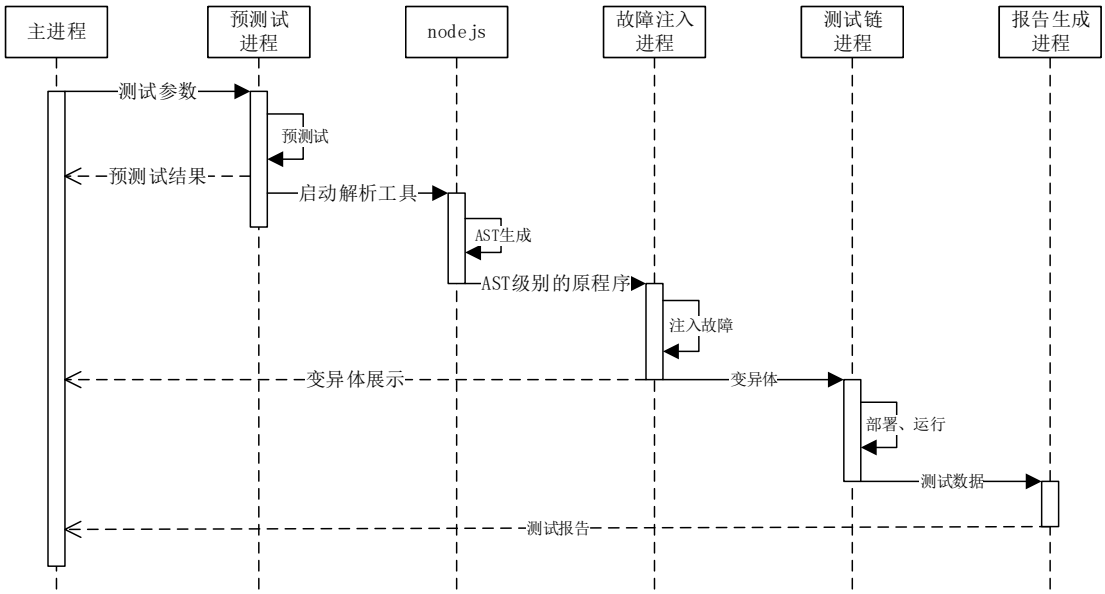


图 4.4: 进程视图

进程视图。图 4.4是本系统的进程视图。进程视图用于描述进程、线程、对象等运行时概念以及他们相关的并发、同步、通信问题。本系统中，当用户配置好测试任务之后，测试参数（包括项目路径、变异算子和待测合约等）以 JSON 的格式传递给预测试进程，开始对变异测试前提进行验证。验证通过后启动 Node.js 进程，该进程主要是运行 solidity-parser-antlr，以获取合约 AST。合约解析完毕后 AST 信息以 JSON 格式传递给故障注入进程进行具体的变异操作，生成的变

异体以供展示和后续运行。测试用例运行时，测试链进程首先创建本地测试链，然后在测试链上部署变异体并运行测试套件。测试用例运行获得的测试数据被传递给报告生成进程，经过分析输出测试报告并输出。

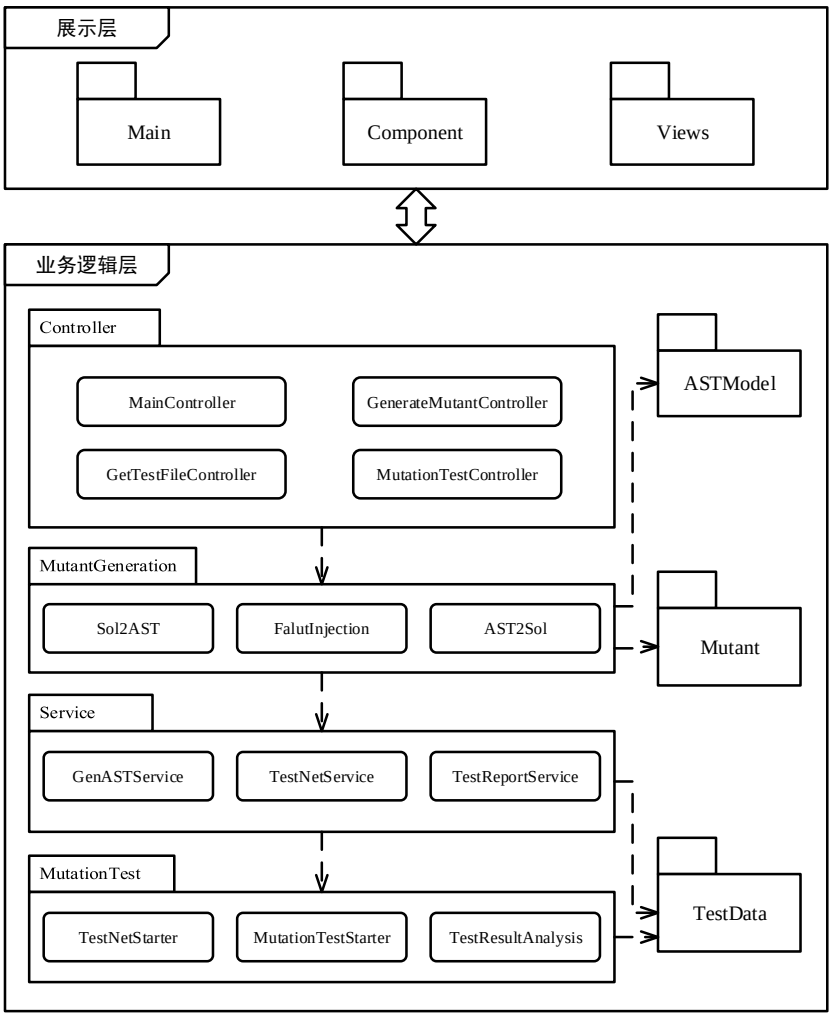


图 4.5: 开发视图

开发视图。图 4.5是本系统的开发视图。开发视图从系统的模块构成和开发过程的角度来描述系统，展示了软件包的层级结构。在展示层中，Main 包存放的是系统入口代码，Component 包存放的是可复用组件，Views 包是项目页面目录。业务逻辑层基于 Springboot 框架实现，主要包含 Controller、MutantGeneration、Service 和 MutationTest 四个包。Controller 包作为控制层，主要处理前端传回的外部请求，调用相应的系统服务。MutantGeneration 包负责变异体生成，主要包含源码至抽象语法树转换（Sol2AST）、故障注入（FaultInjection）和抽象语法树还原（AST2Sol）三个部分的代码。Service 包主要包含抽象语法树解析

(GenASTService)、测试链服务 (TestNetService) 及测试报告生成服务 (TestReport-Service) 三个部分，其中 GenASTService 负责启动 nodejs 服务并调用合约解析工具 solidity-parser-antlr。MutationTest 包主要包含测试链启动（TestNetStarter）、变异测试启动（MutationTestStarter）和测试结果分析（TestResultAnalysis）部分的代码。ASTModel、Mutant、TestData 是系统中重要的三个对象，分别对应抽象语法树格式的合约对象、源码格式的变异体对象以及变异测试相关数据对象。

4.2.3　持久化设计

本文提出的面向以太坊 Solidity 语言的变异测试系统使用 MySQL 数据库来持久化对象，按照需求分析以及后续实验需求，定义了如下持久化对象：

表 4.4: Mutant 表主要字段

字段名	字段类型	描述
ID	INT(11)	自增编号
MutantName	VARCHAR(200)	变异体名
OriSCName	VARCHAR(200)	原智能合约名
AST	MEDIUMTEXT	抽象语法树
Line	INT(11)	变异行号
OpName	VARCHAR(200)	变异算子名
Content	MEDIUMTEXT	变异体内容

表4.4展示了 Mutant 的数据库字段，每一条数据都对应一个变异体。该数据在变异体生成步骤获取并写入数据库，记录生成的变异体详细信息。通过数据库中的变异体名和变异体内容可以还原成一个个独立的变异体。

表4.5展示了 TestResult 的数据库字段，每一条数据都对应变异体的一次检测。在执行变异测试时，每次先在测试链上部署一个变异体，再执行测试文件。变异体状态 MutCondition 是根据测试输出 TestOutPut 得到，有编译失败、杀死、存活三种状态。其中变异体存活与否是通过对比变异体和原程序在同一测试文件上的测试输出来判断变异体状态，若测试输出不同则变异体被杀死，若相同则变异体存活。BranchCoverage 和 CoveredLine 字段是实验设计需要，使用 solidity-coverage [1] 工具获取。

[1] https://github.com/sc-forks/solidity-coverage

表 4.5: TestResult 表主要字段

字段名	字段类型	描述
ID	INT(11)	自增编号
MutantName	VARCHAR(200)	变异体名
TestFileName	VARCHAR(200)	测试文件名
TestNet	MEDIUMTEXT	测试链命令
BranchCoverage	DOUBLE	分支覆盖率
CoveredLine	MEDIUMTEXT	所有覆盖行
TestOutPut	MEDIUMTEXT	测试输出
MutCondition	VARCHAR(200)	变异体状态
TimeCost	INT(11)	测试耗时

4.3　本章小结

本章节首先对系统进行用例分析，根据用例分析的结果，将系统划分为测试任务配置、变异体生成及展示、测试报告生成三个模块；其次从功能性需求和非功能性需求两个方面对系统进行需求分析；再次介绍系统概要设计，介绍了系统整体架构并从逻辑、进程和开发三个角度对系统进行详细描述；最后给出系统的持久化设计。

第五章　详细设计与实现

体本章主要介绍面向以太坊 Solidity 语言的变异测试系统的详细设计与实现。根据第三章进行的需求分析与概要设计，本系统被划分为测试任务配置、变异集合生成和测试报告生成三个模块，本章分别对三个模块进行详细介绍。

5.1　测试任务配置模块

5.1.1　详细设计

测试任务配置模块主要负责变异测试前的准备工作，包括测试前提验证子模块和测试参数配置子模块。测试前提验证子模块主要负责验证用户输入的项目是否符合标准，即是否是 Truffle 框架构建的项目，且是否测试用例全部通过。用户首先在前端提供的文本框中输入本地 DApp 项目路径，接着后端使用 @RequestBody 接受前端返回的参数，然后进入该目录下检测是否包含合约文件夹 contracts、部署命令文件夹 migrations、测试文件夹 test 以及 truffle 配置文件 truffle-config.js。如果包含上述文件或文件夹，则初步判断项目为 Truffle 框架构建，可以进入到测试用例检测阶段。

测试用例检测阶段首先要在项目根目录启动测试链命令，本系统选择 Truffle 框架配套的 Ganache 测试链作为默认测试链。Ganache 是一个快速启动的个人以太坊区块链，可用于运行测试、执行命令和检查状态。它与可以作为 Truffle 框架的 develop 链一键启动，并支持一键运行测试。系统通过重定向将测试结果保存在本地 TruffleInfo.txt 文件中，然后用流的方式读取该文件，使用关键字匹配的方法判断测试用例是否全部通过。若测试用例全部通过，则可以进入到测试参数配置子模块。

参数配置子模块首先从本地项目路径获取全部待测合约并将合约名称展示到前端 Html 界面。用户可以在前端勾选需要测试的合约文件以及使用的变异算子，这些数据随后被封装成 JSON 格式并使用 post 传递给后端。后端获得参数后使用 ArrayList<T> 保存这些参数，至此测试任务配置模块完成。

5.1.2　核心类图

图5.1展示了测试任务配置模块的核心类图。该模块所涉及的类大部分包含在 Controller、Preparation 和 Analysis 文件夹中。其中 GetTestFileController 主要作用是接受前端输入的项目路径，并检测该项目是否符合 Truffle 框架格式。

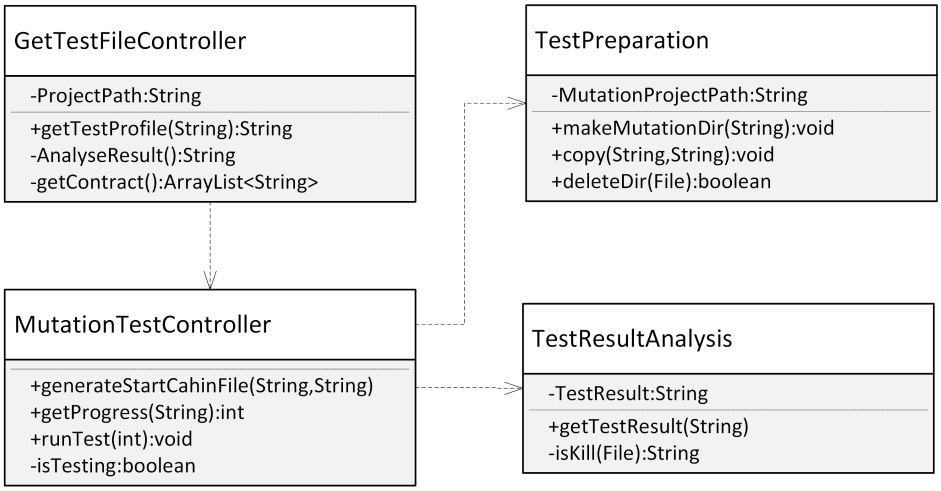


图 5.1: 测试任务配置模块类图

MutationTestController 负责启动本地测试链以及运行项目中的测试用例，同时它还负责解析从前段传回的 JSON 数据，作为测试任务的参数。TestResultAnalysis 负责解析测试结果，判断测试用例是否全部通过。TestPrearation 负责创建变异测试文件夹、备份原始合约文件及测试文件，为后续步骤做准备。

5.1.3　顺序图

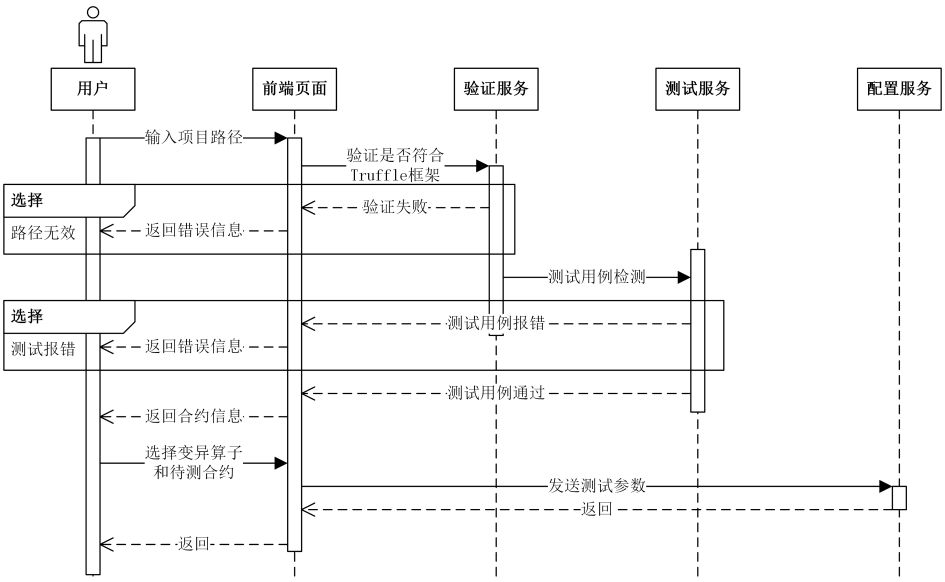


图 5.2: 测试任务配置模块顺序图

图5.2是测试任务配置模块的顺序图。用户输入待测项目路径，系统首先验证其是否为 Truffle 项目，若符合则开始检测测试用例检测，否则返回错误信息。

在测试用例检测过程中，若测试用例报错，系统会返回错误提示，若测试用例全部通过，会在前端中展示项目中的全部智能合约文件以供用户选择。用户选择待测合约及变异算子后，前端会将数据发送到配置服务，完成参数配置。

5.1.4　关键代码

```
1.  public static String getTestProfile(@RequestParam("path") String projectPath){
2.    ProjectPath=projectPath;
3.    MutationTestStater.ProjectPath=projectPath;
4.    File fileDir=new File(ProjectPath);
5.    if(!fileDir.exists()){
6.      return JSON.toJSONString("Fail: Folder directory doesn't exist.");
7.    }else if(!TestPreparation.checkFramework(ProjectPath)){
8.      return JSON.toJSONString("Fail: Require Truffle Project.")
9.    }else{
10.     File logFileDir = new File(ProjectPath+"\\MuSC_MutationTestLog");
11.     if(!logFileDir.exists()){logFileDir.mkdir();}
12.     try {
13.       //省略测试进程启动代码
14.       outputStream.start();
15.     } catch (IOException ioe) {
16.       ioe.printStackTrace();
17.     }
18.     boolean isEnd=false;
19.     while(!isEnd){
20.       //省略测试运行状态检测代码
21.     }
22.     String temp=AnalyseResult();
23.     if(!temp.equals(""))return JSON.toJSONString("Fail: "+temp);
24.     ArrayList<String>cons=getContract();
25.     return JSON.toJSONString(cons);
26.   }
27. }
```

图 5.3: 测试任务配置模块关键代码

图5.3展示了测试任务配置模块的关键代码，由于代码长度较长，省略了部分非核心代码。该函数通过 @ResponseBody 获取前端传回的参数，即用户输入的项目路径。代码第 5-9 行的作用是先检测项目路径是否存在，再调用 checkFramework() 函数来验证项目是否是 Truffle 框架。验证通过后则在第 10-11 行创

建 log 文件夹，并在项目路径下启动测试链并使用“Truffle test”命令运行测试文件。控制台中的测试结果通过流的方式重定向到 log 文件夹下，当测试输出为空时表明测试运行完成，随后在第 22 行调用 AnalyseResult() 函数来判断测试结果，该函数按行读取 log 文件，并通过关键字匹配的方式来检查其中是否含有报错信息。若测试结果为失败则向前端返回错误信息，若测试结果为通过则在第 24 行调用 getContract() 方法获取项目中的合约文件路径，最终这些合约路径被封装在 JSONString 中返回给前端。

5.2　变异集合生成模块

5.2.1　详细设计

变异集合生成模块主要负责待测智能合约的变异操作以及对生成的变异体展示。系统在测试任务配置模块完成了变异算子和待测文件的选择，在变异体生成阶段将针对选定的待测文件，运用每一个变异算子进行故障注入。本系统中变异操作在抽象语法树（Abstract Syntax Tree, AST）级别上进行。AST 是代码语法结构的一种抽象表示，如图5.4所示，AST 使用树状形式来表示源代码的语法结构，树上的每一个节点都对应源代码的一种结构。AST 作为程序代码的一种中间表现形式，在代码分析、代码重构、语言翻译等领域有广泛地应用。

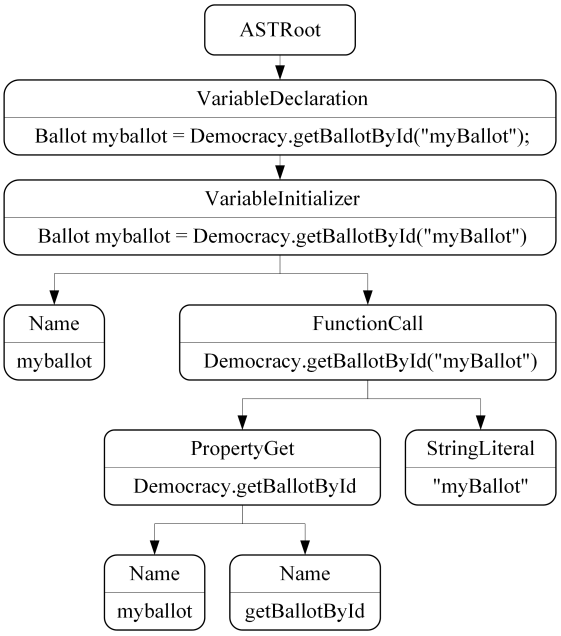


图 5.4: 抽象语法树示例图

MuSC 使用基于 ANTLR 语法的 Solidity 语言转换器 solidity-parser-antlr 来进行 AST 的生成。该工具在 Nodejs 服务端运行，输出的抽象语法树为 JSON 格式。系统在获取 AST 后首先将 JSON 内容转换为相应的 Java 对象，以便于注入故障以及还原成源码形式。选择在 AST 级别上变异再还原而不是直接在源码级别进行变异操作的是因为源文件中的冗余状态，如注释、空行和空格等均可能影响变异的准确性。通过将源码解析成 AST 可排除与程序运行无关的因素，如图5.5所示的原始合约经过 AST 转换再还原可以得到如图5.6所示的合约。

```
1.  pragma solidity ^0.4.11;
2.
3.  import "./StandardToken.sol";
4.  /**
5.   *  SkinCoin token contract. Implements
6.   */
7.  contract SkinCoin is StandardToken, Ownable {
8.    string public constant name = "SkinCoin";
9.    // Constructor
10.   function SkinCoin() {
11.       totalSupply = 1000000000000000;
12.       // Send all tokens to owner
13.       balances[msg.sender] = totalSupply;
14.   }
15. }
```

图 5.5: 原始合约示例图

```
1.  pragma solidity ^0.4.11;
2.  import "./StandardToken.sol";
3.  contract SkinCoin is StandardToken,Ownable{
4.    string public constant name = "SkinCoin";
5.    function SkinCoin() {
6.      totalSupply = 1000000000000000;
7.      balances[msg.sender] = totalSupply;
8.    }
9.  }
```

图 5.6: AST 转换后的合约示意图

Listing 5.1: MuSC 变异体存储形式

```
//Mutant Format
8  ROR if(makeAddress != takerAddress){
13 ROR if(expiration <= now){
24 ROR if(takerToken != takerAmount){
25 ROR if(msg.value != takerAmount){
37 ROR if(msg.value > 0){
42 ROR if(takerAddress > msg.sender){
```

对 AST 执行变异操作时，每个变异算子针对 AST 对应的 Java 对象中的某个成员，通过更改该 Java 对象中的成员变量对智能合约注入缺陷。例如，对于 FVC 变异运算符，通过更改其可见性成员变量来变异原始内容。之后，将每个 AST 变异体转换为源文件版本，该源文件版本与 SCUT 具有相同的语义，只是注入了预定义的故障。为了提高变异测试的效率，系统优化了变异体的存储形式。系统不是存储完整的变异体，而是存储每个变异操作的行号，类型和变异行的代码。表5.1展示了 MuSC 中变异体的存储形式，每行称为一条变异信息，对应于一个变异体。这种存储格式可以有效地减少存储空间和文件读取时间。此外，在执行变异体的过程中，系统只需要记录每个变体信息和相应的执行结果，就可以方便地对测试结果进行计数。

在变异体生成完毕之后，服务端将生成的全部变异体名、对应的原始合约名、变异算子、变异行号以及具体的变异行返回给前端。前端可以根据变异算子或原始合约名展示生成的变异体，同时将源码级别的变异体在页面上展示。展示变异体时依据变异行号将该行字体用红色显示，便于定位缺陷注入位置。

5.2.2　核心类图

变异集合生成模块的类图如图5.7所示。该模块是系统的关键模块，涵盖了完整变异操作。由于对 AST 进行变异的步骤涉及到的类较多，为了不影响阅读体验，这里仅展示了最核心的类。

首先，系统将先前设定的变异算子以及待测合约作为参数传给 GenerateMutantController。接着，通过 Nodejs 启动 GenASTServiceClient，并通过 genAST() 方法将待测合约全文输入到 solidity-parser-antlr 中，获取该合约的 AST 格式。然后对该 AST 按顺序应用选定的变异算子，其中每个变异算子均可以搜索 AST 中的特定 node 节点，并对该节点进行变异。每次注入故障后，都需要调用 ASTMutant 中的 Repair() 方法将 AST 修复以便执行下一个变异体。若有变异算子成功在 AST 中注入了故障，则进入 AST 至源码的转换步骤，该步骤将在顶级类 SourceUnit 的对象中调用 output 方法，以输出还原的 Solidity 语句。项目中的每

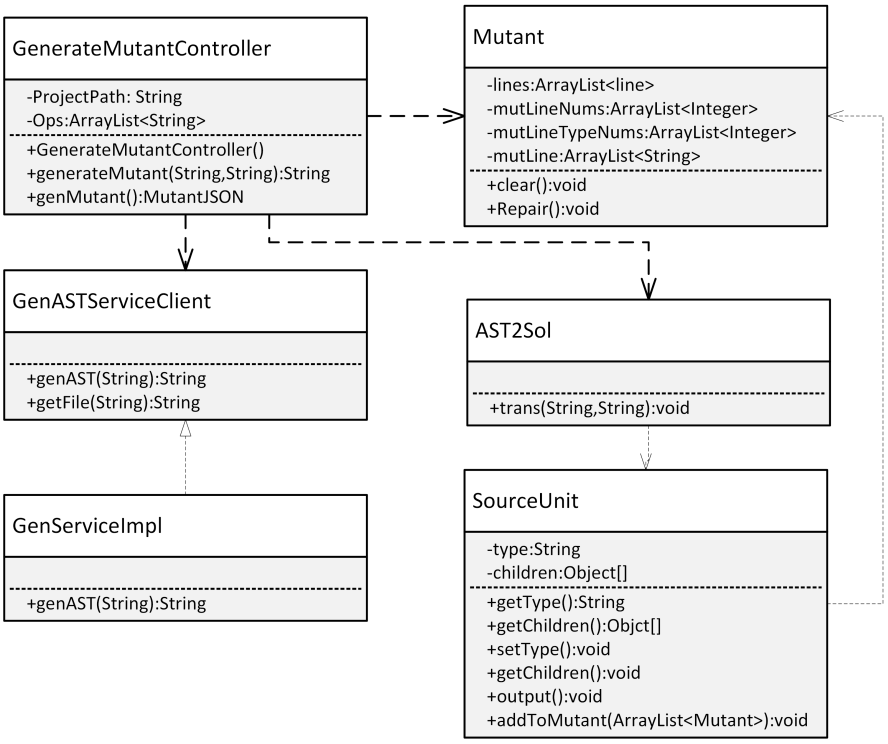


图 5.7: 变异集合生成类图

个模型类都有一个输出方法，用于输出还原的 Solidity 语句，该语句从上到下递归调用，最后输出完整的 Solidity 文件，完成变异体的生成。在所有变异体生成完成后，GenerateMutantController 中的 generateMutant() 方法将全部变异体信息发送给前端以供展示，包括按变异算子或原合约显示变异体和变异体全文展示，具体的展示逻辑使用 JavaScript 实现。

5.2.3　顺序图

图5.8是变异集合生成模块顺序图。该步骤的前提是测试任务配置已完成，当用户在界面上选择开始生成变异体，就进入到本模块。首先后台会启动 Node.js，以运行合约抽象语法树解析工具 solidity-parser-antlr。若 Node.js 未安装或启动失败，则返回错误信息，若成功运行，则将待测合约解析为 AST 格式，并将 AST 级别的原程序以 JSON 格式传递给故障注入服务。在故障注入服务中，AST 被封装成一个个对应的对象，对这些对象运用选中的变异算子会改变对象的特定成员变量，即注入了故障。所有变异算子均执行完后，注入故障的 AST 被发送给 AST 还原服务，以还原成源码级别的变异体，至此变异体生成步骤完毕，这些生成的变异体信息同时被返回给前端用于展示。

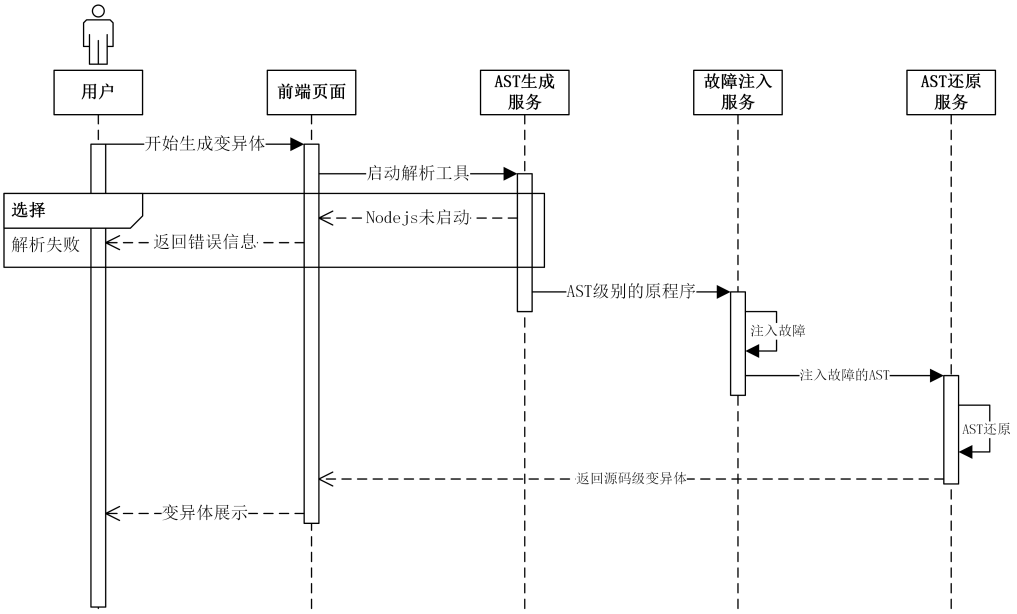


图 5.8: 变异集合生成模块顺序图

5.2.4　关键代码

```
1.  var thrift = require("thrift");
2.  var GenAST= require('./GenAST.js');
3.  var parser=require("solidity-parser-antlr")
4.  //RPC 接口的实现
5.  var genASTImpl = {
6.    genAST: function(sol) {
7.      console.log(JSON.stringify(parser.parse(sol)))
8.      console.log("receive: "+JSON.stringify(parser.parse(sol)));
9.      return JSON.stringify(parser.parse(sol));
10.   }
11. }
12. //启动服务器，默认只支持 TBufferedTransport 和 TBinaryProtocol
13. var server = thrift.createServer(GenAST, genASTImpl);
14. server.listen(9898);
```

图 5.9: 启动 AST 解析工具关键代码

图5.9、图5.10和图5.11是变异体生成与展示模块的关键代码片段。其中，图5.9所示代码为启动 AST 生成服务的主要实现，该代码为 JavaScript 编写，作用是在本地 9898 端口启动 node.js 服务以运行合约解析工具 solidity-parser-antlr。图5.10所示代码片段使用 TSocket 来连接 9898 端口，在第 9 行中将待测合约路径发送到该端口，并调用 solidity-parser-antlr 的 genAST 方法来获取待测合约的抽象语法

树。图5.11展示了将 AST 还原成源码的关键函数，该方法从 AST 根节开始，点自上向下递归调用每一个每个节点的 output 方法，最后输出完整的 Solidity 文件。代码中的三个 if 分别表示 Solidity 合约代码的三种开头方式，分别是合约定义、Solidity 版本声明以及导包指令。

```
1.  public static String genAST(String SolPath){
2.    System.out.println("客户端启动....");
3.    TTransport transport = null;
4.    try {
5.      transport = new TSocket("localhost", 9898, 30000);
6.      TProtocol protocol = new TBinaryProtocol(transport);
7.      GenAST.Client client = new GenAST.Client(protocol);
8.      transport.open();
9.      String result = client.genAST(getFile(SolPath));
10.     System.out.println(result);
11.     transport.close();
12.     return result;
13.   } catch (TTransportException e) {...}//省略异常处理
14. }
```

图 5.10: 获取 AST 关键代码

```
1.  public void output(){
2.    for(int i=0;i<children.length;i++){
3.      if(((JSONObject)children[i]).getString("type").equals("ContractDefinition")) {
4.        JSON.parseObject(children[i].toString(), Contract.class).output();
5.      }
6.      if(((JSONObject)children[i]).getString("type").equals("PragmaDirective")) {
7.        JSON.parseObject(children[i].toString(), Pragma.class).output();
8.      }
9.      if(((JSONObject)children[i]).getString("type").equals("ImportDirective")) {
10.       JSON.parseObject(children[i].toString(), Import.class).output();
11.     }
12.   }
13. }
```

图 5.11: AST 还原关键代码

5.3　测试报告生成模块

5.3.1　详细设计

在此阶段，MuSC 首先将智能合约部署到区块链上，然后在测试目录中执行测试。ESC 在以太坊区块链上运行，其执行结果取决于区块链的状态，例如账户余额和区块高度。为避免区块链状态的影响，应确保 SCUT 与变异体具有相同的初始测试链状态。为此，MuSC 在每个测试文件的开头重新部署了所有 migration，以确保有一组新的测试合约。

MuSC 支持用户定义的测试网创建。只要测试网的端口与项目文件夹下的 truffle configuration 文件中的端口号配置相同，用户就可以创建自定义的测试网。在测试过程中，会导致编译错误的变异体被立即丢弃，并且不会在随后的测试中使用。对于每个变异体，我们记录其执行结果（通过或失败）。最后，根据每个变异体的测试情况，生成一份测试报告，显示总变异得分和每个变异体的测试结果。

5.3.2　核心类图

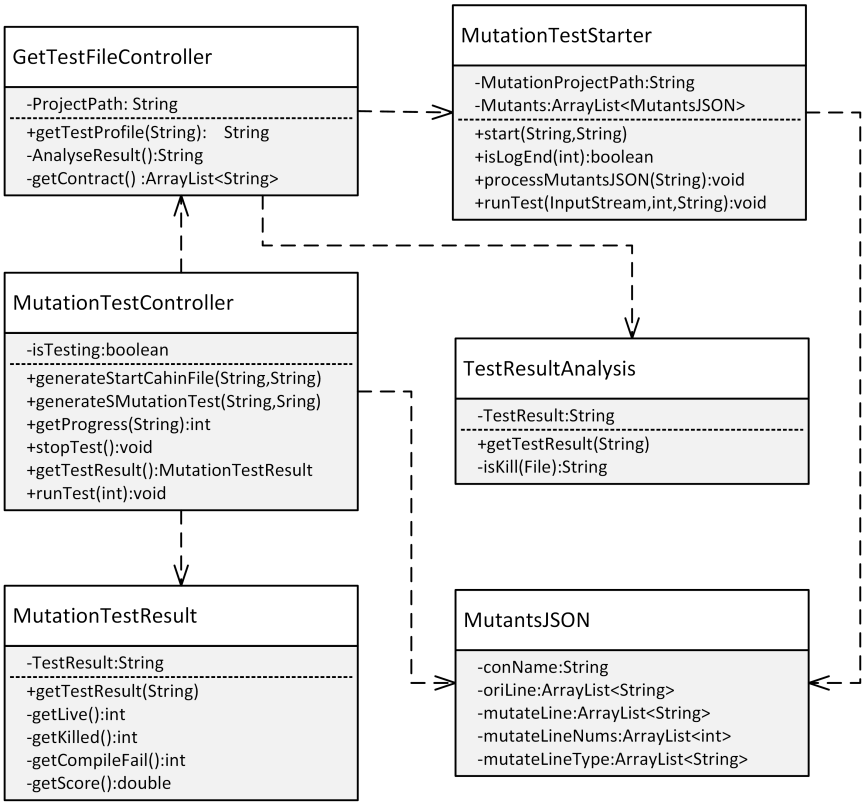


图 5.12: 测试报告生成模块类图

图5.3.2展示了测试报告生成模块的核心类图。其中 GetTestFileController 负责变异体替换原合约等文件相关工作。MutationTestController 负责对从前端传回的用户操作做出相关响应，包括自定义测试链命令、暂停测试等。MutationTestStarter 是变异测试启动类，包括自动化启动测试链、部署变异体以及运行测试套件等。MutationTestResult 是变异测试结果相关类，负责重定向测试输出，从中获取并保存测试结果。TestResultAnalysis 中实现了具体的测试结果的分析方法，包括使用正则匹配关键字以及对比变异体和原合约的测试输出，从而判断变异体是否被杀死。

5.3.3　顺序图

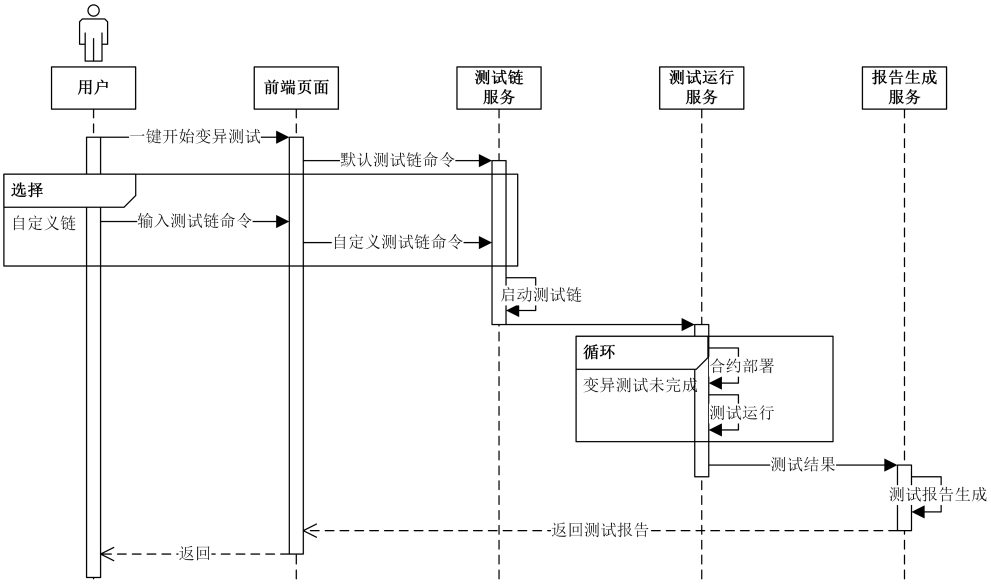


图 5.13: 测试报告生成模块顺序图

图5.3.3是测试报告生成模块的顺序图。该步骤的前提是变异体生成步骤已完成，当用户在界面上选择开始运行变异体，就进入到本模块。用户可以选择不输入自定义测试链命令，直接运行变异体，此时系统使用命令“testrpc-sc –gasLimit 0xfffffffffff –port 8555”来创建默认测试链。系统会在 8555 端口创建一个包含 5 个地址的测试链，将 gasLimit 设置为 0xfffffffffff 的目的是防止在测试过程中发生 gas 不足异常导致状态回滚。用户也可以在前端输入测试链命令，前端会将用户的输入传入到测试链服务端并创建自定义测试链。测试链启动完毕后进入测试运行阶段。对于每一个合约变异体，首先在测试链上部署合约，再运行测试套件。每次测试结果将被传入报告生成服务，最终生成一份测试报告。

5.3.4　关键代码

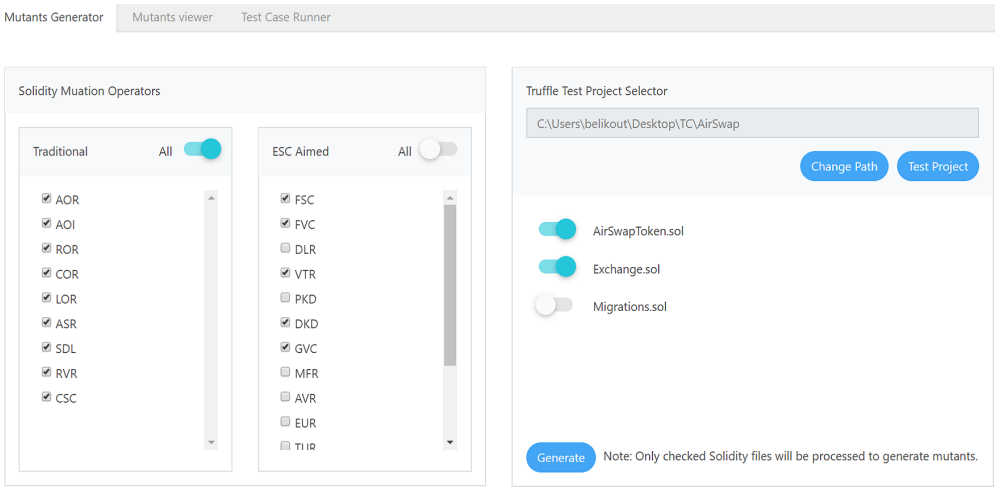
```
1.  public static ArrayList<MutationTestResult> start(String path,String mutantsJSON) {
2.    ArrayList<MutationTestResult>res=new ArrayList<MutationTestResult>();
3.    if(Mutants.size()>0){
4.      int index=0;
5.      for(int i=0;i<Mutants.size();i++){
6.        for(int j=0;j<Mutants.get(i).mutateLine.size();j++){
7.          replaceFileContent(ProjectPath+"\\contracts\\"+Mutants.get(i).conName,geMutant
    string(Mutants.get(i).mutateLine.get(j),Mutants.get(i).mutateLineNums.get(j),Mutants.g
    et(i).oriLine));
8.          if(MutationTestController.isTesting) {
9.            generateMutationTest(index, Mutants.get(i).mutateLineType.get(j));
10.           index += 1;
11.         }
12.       }
13.       replaceFile(ProjectPath+"\\contracts\\"+Mutants.get(i).conName,ProjectPath+"\\Mu
    tants\\ori_"+Mutants.get(i).conName);
14.     }
15.     while(!isEnd){...}//省略测试运行状态检测代码
16.     MutationTestResult tRes=TestResultAnalysis.getTestresult(ProjectPath);
17.     res.add(tRes);
18.   }else{
19.     res.add(new MutationTestResult());
20.   }
21.   return res;
22. }
```

图 5.14: 测试报告生成模块关键代码

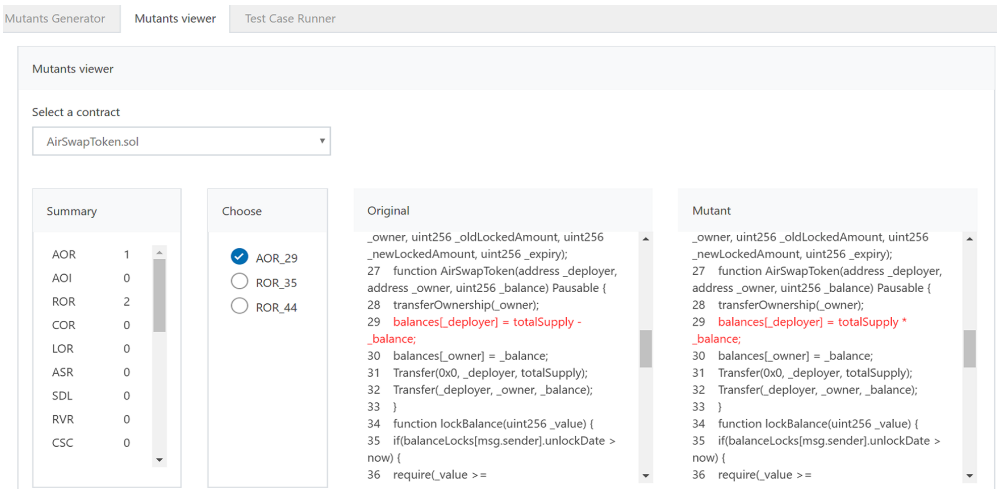
图5.3.4展示了测试报告生成模块关键代码。第 5-7 行根据系统中保存的 Mutant 信息，即变异行号和变异后的行内容，对原合约进行替换，从而生成变异体。第 8 行代码的功能是检查 MutationTestController 的状态，判断用户是否点击了停止按钮，如果没有，则开始运行测试用例。每一个变异体测试完毕后，通过第 13 行的代码将变异后的合约用原始合约替换，以便生成下一个变异体。当所有变异体测试完毕后，在 16 行调用 TestResultAnalysis.getTestresult() 函数来获取变异测试的结果，并将结果返回给前端页面。

5.4　示例展示

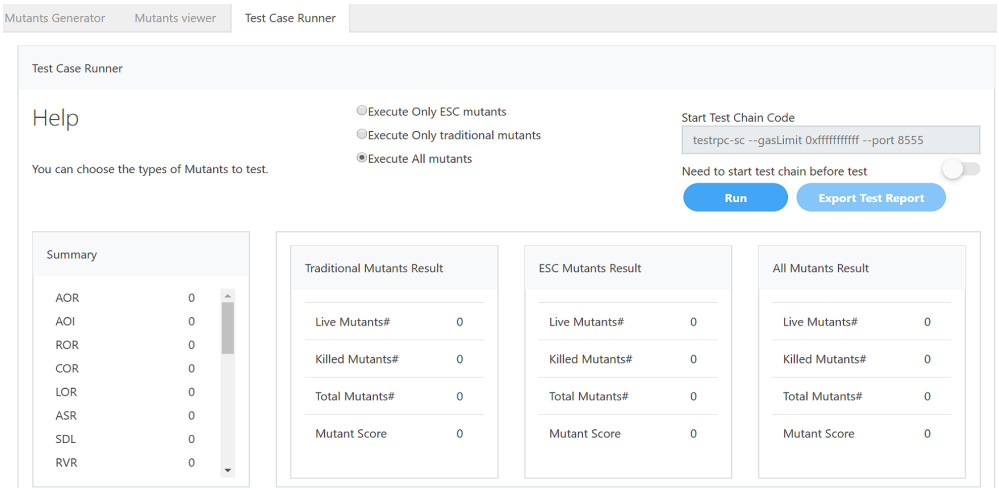
本文实现的面向 Solidity 的变异测试系统包含三个页面，分别是变异体生成页面、变异体展示页面和测试用例运行页面。本节以截图的形式分别展示系统的三个页面以及输出的测试报告，并作简要说明。



(a) 变异体生成页面



(b) 变异体展示页面



(c) 测试用例运行页面

图 5.15: MuSC 页面展示图

变异体生成（Mutants Generator）页面如图5.15(a)所示。该页面主要负责的功能为测试前提检测、测试任务配置和变异体生成。使用时首先在页面右上的文本框中输入待测项目的路径，然后点击 Test Project 按钮即可进行测试前提检测。若输入项目符合 Truffle 框架要求，且自带测试用例全部通过，会在右下部展示出项目中的所有合约文件。用户可以勾选想要测试的合约文件，并在左侧选择使用的变异算子，最后点击 Generate 按钮即可生成变异体。

变异体展示（Mutants viewer）页面如图5.15(b)所示。该页面主要负责的功能为变异体展示。用户首先点击左上选择框选中一个待测合约，页面最左侧会显示该合约生成的所有变异体，变异体名由变异算子名和变异行号组成。用户选择一个变异算子，即可在页面右部展示具体内容。系统同时展示原合约内容以及变异体内容，并将变异行标红以便定位。

测试用例运行（Test Case Runner）页面如图5.15(c)所示。该页面主要负责的功能为测试链配置、变异测试执行以及测试报告导出。右上文本框用于用户输入自定义测试链命令，如果不修改则使用默认测试链。用户也可以点击中部的选择框来选择仅执行一般变异体或特殊变异体。点击右侧 Run 按钮即可开始执行变异测试，这一过程通常需要花费较长时间，系统会在页面上展示目前正在检测的变异体名以及执行进度。当变异测试完成后，页面下部会展示出简单的结果，如果需要查看详细测试结果，点击 Export Test Report 即可导出测试报告。

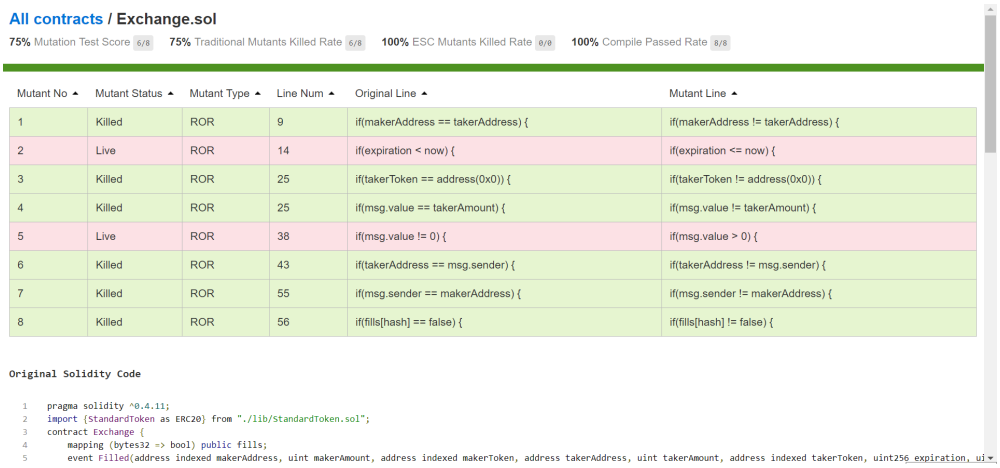


图 5.16: 测试报告示例图

测试报告示例如图5.16所示。生成的测试报告以 html 的格式保存在测试项目文件夹下，可以使用浏览器打开。测试报告上半部分以表格的形式详细展示了每个变异体的杀死情况，下半部分为经过 AST 变换后的原合约代码。测试人员可以基于该报告针对存活的变异体设计新的测试用例，以提高测试的充分性。

5.5　本章小结

本章对面向 Solidity 的变异测试系统进行详细设计并介绍实现细节。本章将系统分为测试任务配置、变异集合生成和测试报告生成三个模块。针对每个模块，首先介绍该模块的详细设计，然后使用核心类图和顺序图来展示该模块的实现，最后给出该模块的关键代码。介绍完三个模块之后，通过实际截图的方式来展示系统的三个页面以及生成的测试报告。

第六章　实验与评估

本章节主要评估本文所提出的变异测试系统及变异算子的有效性。首先介绍研究问题与实验对象，然后针对两个研究问题设计了两个实验，分别给出了详细的实验步骤并对结果进行分析，最后讨论有效性威胁。

6.1　研究问题

本节旨在回答以下研究问题：

- **RQ1：与逻辑测试相比，变异测试是否可以更有效地评估 ESC 测试用例集的测试充分性？**对于一个测试套件，其测试充分性可以通过其缺陷检测能力来衡量。本实验通过对比基于变异测试的测试用例集与基于逻辑覆盖的测试用例集，来验证变异测试方法是否能够提高 ESC 测试的充分性。
- **RQ2：本文提出的 Solidity 特殊变异算子在 ESC 变异测试中是否有效？**对于每一个特定的变异算子，需要通过实验获得（1）非等价变异体生成率；（2）在实际应用中是否会导致真正的错误。这些数据将用来分析变异体与实际缺陷之间的相关性。

6.2　实验对象

表 6.1: 实验对象

DApp	代码行数	代码分支数	测试用例数
AirSwap	422	76	46
CryptoFin	492	58	64
SkinCoin	401	66	51
SmartIdentity	262	34	87

本文以四个不同的真实以太坊分布式应用程序（DApp）中的 26 个智能合约作为实验对象。其中，SkinCoin [1]是一种通用的加密货币，用于在游戏中即时

[1]SkinCoin: http://www.britchicks.com/

交易皮肤和在电子竞技赛事上下注；SmartIdentity [2]依赖以太坊区块链来表示使用智能合约的身份；Cryptofin [3]是一个 Solidity 库的集合，其主要关注点是数组；AirSwap [4]是一个建立在以太坊上的点对点的交易网络平台。选择以上四个项目作为实验对象的原因在于，它们都在 GitHub 上开源，且附带了一组设计良好的测试套件，可以减少后续扩充测试用例时的工作量。表 6.1展示了实验对象的基本信息，包括其名称、代码行数、代码分支数和测试用例数。

6.3　评价指标

本节介绍实验中使用到的评价指标。针对 RQ1 变异测试的有效性，本文使用分支覆盖率和变异得分作为评价指标。其中，分支覆盖率由程序中所有判定语句的取真分支或取假分支被执行到的比率（判定结果被评价的次数 / 判定结果的总数）计算而来，变异得分由被杀死变异体数和非等价变异体总数的比率（被杀死变异体数 / (变异体总数 - 等价变异体数)）计算而来。这两种指标都可以用来衡量测试套件的充分性，分支覆盖率和变异得分越高，均表明测试套件的充分性越高。本文分别基于最大分支覆盖率和最大变异得分构建两个测试套件，再获取两个测试套件在验证集上的两种指标，从而对比测试套件之间的区别。针对 RQ2 变异算子的有效性，本文使用等价变异体率、变异得分作为评价指标。一个变异算子的等价变异体率越低，说明它的应用效果越好，变异得分越低，说明该类型变异体更难被杀死，更有实际价值 [65]。

6.4　实验一：变异测试有效性

6.4.1　实验设计

本实验通过对比基于变异测试与基于覆盖的方法来验证变异测试的有效性，实验设计如下：

(1) **变异体集生成**。首先对四个实验对象应用全部变异算子（10 个通用变异算子和 16 个 Solidity 特殊变异算子）来生成变异体，再通过字节码对比以及人工检查的手段去除重复变异体和等价变异体，得到变异体集 MS_1。

[2]SmartIdentity: http://www.deloitte.co.uk/smartid/

[3]CrytoFin: https://cryptofinlabs.github.io/cryptofin-solidity/

[4]AirSwap: https://www.airswap.io/

(2) **测试用例池扩充**。该步骤需要构建一个足够大的测试用例池 T，以满足尽可能高的语句覆盖率、分支覆盖率以及变异体杀死能力。该测试用例池主要有三个来源，一是项目自带的测试用例，二是使用 fuzzing 方法生成的随机测试用例，三是人工扩充的测试用例。

(3) **测试用例集构建**。首先在 MS_1 上运行 T 中的所有测试用例，再根据测试结果随机选择部分测试用例构建两个测试用例集 T_{cov} 和 T_{mut}。其中，T_{cov} 满足语句覆盖率和分支覆盖率与 T 相同，T_{mut} 满足变异体杀死能力与 T 相同。最后，随机补充测试用例使 T_{cov}、T_{mut} 的测试用例规模保持一致。

(4) **人工缺陷集构建**。该步骤请两位具有智能合约开发经验且对本文提出的变异算子不知情的硕士生进行人工注入缺陷，每人为每个实验对象注入 20 个缺陷，得到包含 $2*4*20=160$ 个变异体的人工缺陷集 MS_2。

(5) **测试用例集对比**。在 $MS2$ 上分别运行两个测试用例集 T_{cov} 和 T_{mut}，获取它们的变异得分，并进行对比。为了减少误差，本文随机生成了三组测试用例集，将三次实验得到的结果取平均作为最终结果。

6.4.2　结果分析

表 6.2: 每个 DApp 生成的变异体数量

DApp	**General**		**Specific**		**Total**	
	ALL	**NEQ**	**ALL**	**NEQ**	**ALL**	**NEQ**
SkinCoin	314	274	106	106	420	380
SmartIdentity	267	225	86	86	353	311
AirSwap	406	363	175	175	581	538
Cryptofin	464	406	214	183	678	589
Total	1451	1268	581	550	2032	1818

表6.2中总结了根据上述实验步骤 1 中生成的可编译变异体。对于每个DApp，列 2、4、6 和列 3、5、7 分别展示由通用算子、特殊算子和总算子产生的变异体数量与非等价变异体数量。为了排除重复和等效突变体，本文对生成的全部变异体进行了平凡等效变异体检测 [66]。首先编译每个新变异体，再将新变异体的字节码与之前所有变异体的字节码进行比较，如果匹配，则丢弃新变异体。随后，我们又人工检查了一遍剩下的变异体，将语法不同但语义与原程序相同的等价变异体分离出来。从表中数据可以得出通用变异算子总非等价变异体率为 $1268/1451 \approx 87.4\%$，特殊变异算子的总非等价变异体率为 $550/581 \approx 94.7\%$，

表 6.3: T_{cov} 和 T_{mut} 在 MS_1 和 MS_2 上的覆盖率及变异得分

DApp	Line Cov		Branch Cov		MS of MS_1		MS of MS_2	
	$T_{cov}(T)$	T_{mut}	$T_{cov}(T)$	T_{mut}	T_{cov}	$T_{mut}(T)$	T_{cov}	T_{mut}
SkinCoin	93.91	89.61	79.55	77.33	62.5	76.0	65.8	74.2
SmartIdentity	99.11	93.24	94.12	85.19	59.3	68.3	72.5	78.8
AirSwap	100.0	95.25	83.33	72.22	62.4	75.6	44.2	51.7
Cryptofin	100.0	94.12	92.17	76.67	72.4	84.7	33.3	47.5
Average	98.26	93.06	89.80	77.85	64.2	76.2	53.6	63.1

表明本文提出的特殊变异算子生成有效变异体的能力优于通用变异算子。最终，本文从 2032 个可编译变异体中筛选出 1818 个非等价变异体作为 $MS1$。

表6.3是 T_{cov} 和 T_{mut} 在两个变异体集 MS_1 和 MS_2 上的测试结果。表第 2-5 列展示了测试用例集在 MS_1 上的语句覆盖率和分支覆盖率，可以看出 T_{mut} 的语句和分支覆盖率均低于 T_{Cov}，因此仅从覆盖率来看 T_{cov} 优于 T_{mut}。表第 6-7 列展示了测试用例集在 MS_1 上的变异得分，可以看出 T_{mut} 的变异得分均高于 TS_{cov}，因此从缺陷检测能力来看 T_{mut} 更优于 T_{cov}。随后，我们在人工缺陷集 MS_2 上进行了验证，结果如表最后两列所示，T_{mut} 的平均变异得分（63.1）高于 T_{cov} 的平均变异得分（53.6），表明 T_{mut} 的缺陷检测能力更优于 T_{cov}。综上，可以得出变异测试在评估 ESC 测试集充分性方面比逻辑测试更有效的结论。

6.5　实验二：变异算子有效性

6.5.1　实验设计

本实验使用变异体数量和等价变异体率来评估变异算子的有效性。此外，本文还进行了一项调查，以分析变异算子和真实缺陷之间的关系。实验设计如下：

(1) **变异体统计与分类**。为了验证变异算子的有效性，本文使用项目自带的测试套件对它们进行了一次变异测试，并将生成的变异体按照变异算子分类，以统计每个变异算子生成的变异体数以及等价变异体率。

(2) **缺陷报告收集与分类**。本文通过检索 GitHub [67–70]、DASP [71] 和 Peck-Shield [72] 中的缺陷报告，最终收集到 729 份已关闭的以太坊智能合约的报告。对于每份报告，人工判断其是否与某一类型的变异算子相关。

6.5.2　结果分析

表 6.4: 变异算子实验结果统计表

Operator	Number of Mutants				MS
	All	Equ.	Killed	Live	
General Mutants					
AOR	257	4	117	136	46.2
AOI	416	108	128	159	44.6
ROR	410	41	148	221	40.1
COR	29	0	5	24	17.2
LOR	0	0	0	0	0.0
ASR	34	0	18	16	52.9
SDL	229	30	78	121	39.2
RVR	36	0	11	25	30.6
CSC	40	0	26	14	65.0
Subtotal	1451	183	531	716	42.6
ESC Specific Mutants					
FSC	4	0	0	4	0.0
FVC	136	0	67	69	49.3
DLR	6	0	1	5	16.7
VTR	101	31	16	54	22.9
PKD	1	0	0	1	0.0
DKD	6	0	2	4	33.3
GVC	65	0	34	31	52.3
MFR	55	0	19	36	34.5
AVR	127	0	35	103	25.4
EUR	20	0	4	16	20.0
TUR	6	0	0	6	0.0
RSD	13	0	3	10	23.1
RSC	13	0	5	8	38.5
ASD	19	0	2	17	10.5
ASC	19	0	13	6	68.4
Subtotal	581	31	201	370	35.2
Total	**2032**	**214**	**732**	**1086**	**40.3**

表 6.4给出了每个变异算子的统计数据，其中第 2-5 列分别描述了所有、等价、死亡和存活变异体的数量，最后一列给出了其变异得分。为了便于对比通用变异算子和特殊变异算子，本文将两类变异算子分成两组来展示。表中第一组数据是由通用变异算子产生的变异体。除 LOR 外，其他通用变异算子产生了至少一个变异体。其中，COR 的变异得分最低，为 17.2 分，但这只说明测试用例在这方面不够充分，并不意味着 COR 算子是无效的。AOI 产生的变异体最多。然而，由 AOI 产生的 416 个变异体中有 108 个是等价的，这是因为某些插入的算术运算符不处理对执行结果有影响的变量。10 个通用变异算子共产生 1451 个变异体，其中 1247 个为非等价变异体，非等价变异体生成率为 85.94%。

第二组数据对应于 ESC 特有的变异算子。FSC、PKD 和 TUR 的变异得分最低，均为 0，说明其变异体均未被测试套件杀死。这三个算子只产生了 11 个变异体，所以这个结果可能没有意义。FVC 和 AVR 分别产生 136 和 127 个变异体，占所有变异体的近一半。此外，VTR 产生了 101 个变异体，但其中 31 个是等价的。例如，VTR 可以将 $(\text{uint i} = 0; \text{i} < 20; \text{i++})$ 中的 uint 用 int 或 uint8 替换，从而产生两个变异体。然而，这两个变异体不会影响程序执行的结果，因此它们在逻辑上是等价的。ASC 产生 19 个变异体，变异得分最高为 68.4 分，ASD 产生 19 个变异体，变异得分最低为 10.5 分。这可能表明许多测试用例只触发 assert(true)，而不考虑 assert(false)，这可能会导致隐藏的缺陷。在项目自带测试套件上，特殊变异算子的平均变异得分为 35.2，低于通用变异算子的平均变异得分 42.6。这表明，测试人员在编写测试用例时更少注意到由 Solidity 特征引起的问题，这也反应了特殊变异算子的重要性。

此外，本文还进行了一项调研，从开源社区查找与以太坊智能合约相关的各种问题和错误报告。在 729 个报告中，117 个与新提出的变异算子相关，包括 41 个关键字算子错误、35 个全局变量和函数算子错误、9 个变量单位算子错误和 32 个错误处理算子错误。下面给出部分代表性的例子：

- **SWC-100 (FVC)** 默认情况下，没有指定功能可见性类型的函数是 public 的。如果开发人员忘记设置可见性，并且恶意用户能够进行未经授权或意外的状态更改，则可能会导致漏洞。

- **SWC-109 (DLR)** 未初始化的本地存储变量可能指向合约中的意外存储位置，这可能导致意外的漏洞。

- **DASP#item-3 (VTR/MFR)** 溢出条件会产生不正确的结果，特别是在未预料到可能发生的情况下，可能会危及程序的可靠性和安全性。

- **SWC-120 (GVC)** 使用区块时间戳 block.timestamp 生成随机数是不安全的，因为矿工可以将其设定为几秒内的任意值。同理，区块哈希 blockhash 和区块难度 block.difficulty 也是不安全的，它们都可以被矿工操控。

- **DASP#item-2 (AVR/RSD/RSC)** 当合约使用 tx.origin 而不是 msg.sender 来验证调用方、处理长时间要求的大型授权逻辑以及在代理库或代理合约中鲁莽地使用 delegatecall 时，可能会出现访问控制漏洞。

- **SWC-123 (ASD/ASC)** Solidity 中的 assert() 函数用于检查不变量。正确的函数化代码不应该到达失败的断言状态。可到达断言意味着合约中存在允许其进入无效状态的错误，或者断言语句使用不正确（例如验证输入）。

上述结果表明，本文提出的 Solidity 变异算子与现实存在的错误有关，因此能够注入真实的缺陷，从而帮助开发人员避免一些常见的错误。

6.6　效度分析

内部效度分析。内部效度指因变量和自变量之间关系的确定性程度，代表实验结论的真实性。本实验存在两种内部效度威胁。首先，MuSC 是否正确地对智能合约进行了变异。MuSC 的变异操作是在 AST 级别实现的，其中 AST 是由 solidity-parser-antlr 生成的，故障注入和 AST 还原由作者实现。本文人工检查了每一个变异题，确保它们按照预期进行了变异。其次，由于 Solidity 版本在不断更新，解析器和变异工具也需要相应更新。同时，变异算子的设计还应考虑版本更新的影响，以满足最新的测试要求。

构建效度分析。构建效度指的是一个测验能测量理论的概念或特质的程度。本实验存在两种构建效度威胁。首先，本文测试用例集包含了人工编写的测试用例，具有一定主观性，无法确保其质量。而测试用例的质量将对本文的实验产生影响，例如低质量的测试用例会使绝大多数变异体存活，从而导致变异得分极低。其次，等价变异体的鉴定是人工进行的，不能保证所有的等价变异体都被排除在外，本文对所有等价变异体进行了二次检验来减少人工识别的误差。

外部效度分析。外部效度是指研究结果、变量条件、时间和背景的代表性和普遍适用性。本文的实验是在来自四个 DApp 的 26 个智能合约上进行的，无法保证所选实验对象的代表性。为了尽可能使实验代表性最大化，本文四个实验对象是从四个不同的应用领域来选择的，包括交易平台、智能身份证、游戏和 Solidity 库。考虑到这些合约功能上的不同，它们涵盖的 Solidity 特性也较广泛。

6.7　本章小结

本章主要介绍以太坊智能合约变异测试的实验部分。首先提出两个研究问题：（1）变异测试在以太坊智能合约测试领域是否有效，（2）本文提出的变异算子是否有效；然后介绍本文的实验对象，并针对两个研究问题，分别设计两个实验并详细给出了实验步骤和结果；最后对从内部效度、结构效度和外部效度三方面对实验进行效度分析。通过实验一可以得出结论：变异测试在评估以太坊智能合约测试套件充分性方面的有效性超过基于逻辑覆盖率的评估方法。通过实验二可以得出结论：新提出的 Solidity 变异算子产生的等价变异体率较低，且能有效揭示真实存在的缺陷。

第七章　总结与展望

7.1　总结

随着区块链以及智能合约技术的发展，智能合约的质量保障愈发重要。为了有效评估以太坊智能合约测试的充分性，本文引入了变异测试的思想，设计并实现了一个面向 Solidity 语言的变异测试系统。本文针目前对 Solidity 语言的变异测试研究的不足，从关键字、全局变量及函数、异常检测、合约漏洞等方面入手，设计了 16 个新的变异算子。这些变异算子使变异测试能够检测出将 Solidity 语言特有的缺陷，从而提高在 Solidity 上应用变异测试的效果。

本文研究了面向 Solidity 语言的变异测试流程，在此基础上对 Solidity 变异测试系统进行需求分析和概要设计，随后实现了一个变异测试工具 MuSC。MuSC 基于 Spring boot 框架构建，使用了 Nodejs 服务、Truffle 框架以及 mysql 等技术。它可以在 AST（抽象语法树）级别上高效且精确地执行变异操作，并提供变异体展示、自定义测试链创建、合约自动化部署、测试报告生成等功能。此外，MuSC 还提供了简洁易用的图形用户界面，增强了系统的可用性。MuSC 已经在 GitHub 上开源，任何人都可以从https://github.com/belikout/MuSC-Tool-Demo-repo获取其源码并使用。

本文在 4 个真实以太坊分布式应用中的 26 个智能合约上进行了实证研究，以评估我们提出的变异测试方法的有效性。实验结果表明（1）基于变异测试的方法在缺陷检测率上优于基于覆盖的方法（变异得分 63.1 比 53.6）；（2）针对 Solidity 的特殊变异算子能够有效地揭示真实的缺陷。这些结果体现了变异测试在以太坊智能合约质量保障中的巨大潜力。

7.2　展望

本文对面向 Solidity 的变异测试系统的研究还有许多改进和提高空间。第一，本文的研究是在 Solidity 0.5.12 版本上进行的，截至 2020 年 4 月，Solidity 已经更新到 0.6.3，可能某些语言特性已经发生了改变。变异算子需要随之更新，以适应最新版本的测试需求。因此，针对 Solidity 的特殊变异算子的研究是一个长期工作。第二，由于时间原因，本文中实验对象仅为 4 个 DApp 中的 26 个智能合约，代表性有限。在后续的实验中，实验对象被扩充到 10 个 DApp，对应 98 个智能合约，从而提高了实验的说服力。第三，本文未对面向 Solidity 的变异

测试进行优化，因此每次测试需要花费的时间较长。后续考虑从变异算子约减、高阶变异体等角度入手来提高效率，增加变异测试的可用性。第四，由于区块链的特性，智能合约的执行结果与区块链状态有关。因此，如果链状态发生改变，同一个测试用例的两次执行结果可能会不同。而目前 MuSC 工具在判断变异体是否被杀死时仅根据 Truffle 框架执行测试用例时的输出，没有考虑测试链状态对变异体杀死条件的影响。后续可以对此进行深入研究，通过构建多样化的测试链来提高变异测试效果。

参考文献

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system.

[2] 薛腾飞, 傅群超, 王枞, 王新宴, 基于区块链的医疗数据共享模型研究, 自动化学报 43 (9) (2017) 1555–1562.

[3] 张宁, 王毅, 康重庆, 程将南, 贺大玮, 能源互联网中的区块链技术: 研究框架与典型应用初探, 中国电机工程学报 36 (15) (2016) 4011–4022.

[4] Y. Zhang, J. Wen, An iot electric business model based on the protocol of bitcoin, in: Proceedings of the 18th international conference on intelligence in next generation networks, 2015, pp. 184–191.

[5] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, H. Wang, Blockchain challenges and opportunities: A survey, International Journal of Web and Grid Services 14 (4) (2018) 352–375.

[6] B. Vitalik, et al., Ethereum: A next-generation smart contract and decentralized application platform, URL https://github. com/ethereum/wiki/wiki/% 5BEnglish% 5D-White-Paper.

[7] Thoughts on the dao hack, https://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/ (2016).

[8] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, A. Hobor, Finding the greedy, prodigal, and suicidal contracts at scale, in: Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC), 2018, pp. 653–663.

[9] P. Delgado-Pérez, S. Segura, I. Medina-Bulo, Assessment of C++ object-oriented mutation operators: A selective mutation approach, Software Testing, Verification and Reliability 27 (4-5) (2017) e1630.

[10] J. Yue, M. Harman, An analysis and survey of the development of mutation testing, IEEE Transactions on Software Engineering 37 (5) 649–678.

[11] Y.-S. Ma, Y.-R. Kwon, J. Offutt, Inter-class mutation operators for Java, in: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE), 2002, pp. 352–363.

[12] D. Le, M. A. Alipour, R. Gopinath, A. Groce, MuCheck: An extensible tool for mutation testing of haskell programs, in: Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA), 2014, pp. 429–432.

[13] E. Omar, S. Ghosh, An exploratory study of higher order mutation testing in aspect-oriented programming, in: Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering (ISSRE), 2012, pp. 1–10.

[14] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman, Mutation testing advances: An analysis and survey, in: Advances in Computers, Vol. 112, 2019, pp. 275–378.

[15] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on test data selection: Help for the practicing programmer, Computer 11 (4) (1978) 34–41.

[16] A. J. Offutt, Investigations of the software testing coupling effect, ACM Transactions on Software Engineering and Methodology 1 (1) (1992) 5–20.

[17] T. Budd, R. Lipton, R. Demillo, F. Sayward, The design of a prototype mutation system for program testing, Managing Requirements Knowledge, International Workshop on 0 (1978) 623.

[18] K. N. King, A. J. Offutt, A fortran language system for mutation-based software testing, Software: Practice and Experience 21 (7) (1991) 685–718.

[19] H. Agrawal, R. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, E. Spafford, Design of mutant operators for the c programming language, Tech. rep. (1989).

[20] S. Kim, J. Clark, J. McDermid, The regorous generation of java mutation using hazop, in: In Proceedings of the 12 the International Conference on Software and Systems Engineering and Their Applications (ICSSEA'99), 1999.

[21] Y.-S. Ma, J. Offutt, Y. R. Kwon, MuJava: An automated class mutation system, Software Testing, Verification and Reliability 15 (2) (2005) 97–133.

[22] S. Mirshokraie, A. Mesbah, K. Pattabiraman, Efficient javascript mutation testing, in: Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation, 2013, pp. 74–83.

[23] L. Deng, J. Offutt, P. Ammann, N. Mirzaei, Mutation operators for testing android apps, Information and Software Technology 81 (2017) 154–168.

[24] W. Chan, S. Cheung, T. Tse, Fault-based testing of database application programs with conceptual data model, in: Proceedings of the 5th International Conference on Quality Software, 2005, pp. 187–196.

[25] J. Tuya, M. J. Suárez-Cabal, C. De La Riva, Mutating database queries, Information and Software Technology 49 (4) (2007) 398–417.

[26] S. C. Lee, J. Offutt, Generating test cases for xml-based web component interactions using mutation analysis, in: Proceedings of the 12th International Symposium on Software Reliability Engineering, 2001, pp. 200–209.

[27] W. Xu, J. Offutt, J. Luo, Testing web services by xml perturbation, in: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering, 2005, pp. 10–pp.

[28] J. J. Honig, M. H. Everts, M. Huisman, Practical mutation testing for smart contracts, in: Data Privacy Management, Cryptocurrencies and Blockchain Technology, 2019, pp. 289–303.

[29] P. Hartel, R. Schumi, Gas limit aware mutation testing of smart contracts at scale, arXiv preprint arXiv:1909.12563.

[30] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, A. Hobor, Making smart contracts smarter, in: Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS), 2016, pp. 254–269.

[31] E. Albert, P. Gordillo, B. Livshits, A. Rubio, I. Sergey, Ethir: A framework for high-level analysis of ethereum bytecode, in: International Symposium on Automated Technology for Verification and Analysis, 2018, pp. 513–520.

[32] E. Albert, J. Correas, P. Gordillo, G. Román Díez, A. Rubio, Safevm: a safety verifier for ethereum smart contracts, 2019, pp. 386–389.

[33] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, B. Scholz, Vandal: A scalable security analysis framework for smart contracts, arXiv preprint arXiv:1809.03981.

[34] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, B. Roscoe, Reguard: finding reentrancy bugs in smart contracts, in: Proceedings of the 40th International Conference on Software Engineering: Companion, 2018, pp. 65–68.

[35] X. Wang, H. Wu, W. Sun, Y. Zhao, Towards generating cost-effective test-suite for Ethereum smart contract, in: Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 2019, pp. 549–553.

[36] I. Sergey, A. Hobor, A concurrent perspective on smart contracts, 2017, pp. 478–493.

[37] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, Z. Chen, Easyflow: Keep ethereum away from overflow, in: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), 2019, pp. 23–26.

[38] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, Y. Smaragdakis, Madmax: Surviving out-of-gas conditions in ethereum smart contracts, Proceedings of the ACM on Programming Languages 2 (2018) 1–27.

[39] N. Atzei, M. Bartoletti, T. Cimoli, A survey of attacks on ethereum smart contracts (sok), in: International conference on principles of security and trust, 2017, pp. 164–186.

[40] B. Jiang, Y. Liu, W. Chan, Contractfuzzer: Fuzzing smart contracts for vulnerability detection, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE), 2018, pp. 259–269.

[41] 付梦琳, 吴礼发, 洪征, 冯文博, 智能合约安全漏洞挖掘技术研究, 计算机应用 39 (7) (2019) 1959–1966.

[42] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, M. Vechev, Securify: Practical security analysis of smart contracts, in: Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS), 2018, pp. 67–82.

[43] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, P. Saxena, Exploiting the laws of order in smart contracts, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2019, pp. 363–373.

[44] S. Kalra, S. Goel, M. Dhawan, S. Sharma, Zeus: Analyzing safety of smart contracts., in: NDSS, 2018, pp. 1–12.

[45] N. Grech, L. Brent, B. Scholz, Y. Smaragdakis, Gigahorse: thorough, declarative decompilation of smart contracts, in: Proceedings of the 41st International Conference on Software Engineering (ICSE), 2019, pp. 1176–1186.

[46] T. Chen, X. Li, X. Luo, X. Zhang, Under-optimized smart contracts devour your money, in: Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, pp. 442–446.

[47] M. Marescotti, M. Blicha, A. E. Hyvärinen, S. Asadi, N. Sharygina, Computing exact worst-case gas consumption for smart contracts, in: International Symposium on Leveraging Applications of Formal Methods, 2018, pp. 450–465.

[48] K. Struga, O. Qirici, Bitcoin price prediction with neural networks., in: RTA-CSIT, 2018, pp. 41–49.

[49] 欧阳丽炜, 王帅, 袁勇, 倪晓春, 王飞跃, 智能合约: 架构及进展, 自动化学报 45 (3) (2019) 445–457.

[50] N. Szabo, Smart contracts: Building blocks for digital markets.

[51] 袁勇, 王飞跃, 区块链技术发展现状与展望, 自动化学报 42 (4) (2016) 481–494.

[52] 沈鑫, 裴庆祺, 刘雪峰, 区块链技术综述, 网络与信息安全学报 2 (11) (2016) 11–20.

[53] 邵奇峰, 金澈清, 张召, 钱卫宁, 周傲英, et al., 区块链技术: 架构及进展, 计算机学报 41 (5) (2018) 969–988.

[54] P. G. Frankl, S. N. Weiss, C. Hu, All-uses vs mutation testing: An experimental comparison of effectiveness, Journal of Systems and Software 38 (3) (1997) 235–253.

[55] J. H. Andrews, L. C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments?, in: Proceedings of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA, 2005, pp. 402–411.

[56] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing?, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 2014, pp. 654–665.

[57] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, Ethereum project yellow paper (2014) 1–32.

[58] Introduce a real constant keyword and rename the current behaviour, https://github.com/ethereum/solidity/issues/992 (2016).

[59] Solidity data location, https://solidity.readthedocs.io/en/v0.5.8/types.html#data-location (2019).

[60] Special variables and functions, https://solidity.readthedocs.io/en/v0.5.8/units-and-global-variables.html#special-variables-and-functions (2019).

[61] Solidity ether units, https://solidity.readthedocs.io/en/v0.5.8/units-and-global-variables.html#ether-units (2019).

[62] Solidity error handling: Assert, require, revert and exceptions, https://solidity.readthedocs.io/en/v0.5.8/control-structures.html#error-handling-assert-require-revert-and-exceptions (2019).

[63] Alert: New batchoverflow bug in multiple erc20 smart contracts, https://blog.peckshield.com/2018/04/22/batchOverflow/ (2018).

[64] P. B. Kruchten, The 4+ 1 view model of architecture, IEEE software 12 (6) (1995) 42–50.

[65] U. Praphamontripong, J. Offutt, L. Deng, J. Gu, An experimental evaluation of web mutation operators, in: 2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2016, pp. 102–111.

[66] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, M. Harman, Detecting trivial mutant equivalences via compiler optimisations, IEEE Transactions on Software Engineering 44 (4) (2017) 308–333.

[67] Not so smart contracts: Examples of solidity security issues, https://github.com/crytic/not-so-smart-contracts (2018).

[68] Smart contract weakness classification and test cases, https://smartcontractsecurity.github.io/SWC-registry/ (2020).

[69] Uscc-submissions-2017, https://github.com/Arachnid/uscc/tree/master/submissions-2017 (2017).

[70] Ethereum-goethereum-issues, https://github.com/ethereum/go-ethereum/issues (2020).

[71] Decentralized application security project, http://www.dasp.co/#item-1 (2020).

[72] Peckshield, https://blog.peckshield.com/ (2020).

简历与科研成果

基本情况 巫浩然，男，汉族，1996 年 12 月出生，江苏省仪征市人。

教育背景

20014.9 ~ 2018.6 西安电子科技大学软件学院 本科

2018.9 ~ 2020.6 南京大学软件学院 硕士

这里是读研期间的成果

1. Weisong Sun, Xingya Wang, **Haoran Wu**, Ding Duan, ”MAF: Method-Anchored Test Fragmentation for Test Code Plagiarism Detection”, ICSE-SEET 2019
2. **Haoran Wu**, Jiehui Xu, Yubin Gao, Xingya Wang, Guangyao Xu, Weisong Sun, Ding duan, ”Testing Adequacy of Smart Contracts on Blockchain”, IRC-SEMS 2019
3. Xingya Wang, **Haoran Wu**, Weisong Sun, Yuan Zhao, ”Towards generating cost-effective Test-Suite for ethereum smart contract”, SANER 2019
4. Zixin Li, **Haoran Wu**, Jiehui Xu, Xingya Wang, Lingming Zhang, Zhenyu Chen, ”MuSC: A tool for mutation testing of ethereum smart contract”, ASE-DEMO 2019
5. “一种基于测试代码片段相似性的测试程序抄袭检测方法”, 申请号:201810561-223.3, 已受理。
6. “一种基于多目标优化的智能合约测试方法”, 申请号:201910498422.9, 已受理。
7. “一种基于 NSGA-II 的智能合约测试方法”, 申请号：201910499496.4, 已受理。
8. “安卓单元测试自动生成工具系统”, 申请号:2018SR726260, 已受理。

致　　谢

时光如梭，两年的硕士生涯即将结束。在这个特殊的毕业季，我要对所有曾经帮助和支持我的人表示由衷的感谢。首先，我要感谢我的导师刘嘉老师和何铁科老师，他们为我的研究指明了方向。接着，我要感谢实验室里教导过我，关心过我的老师，他们是陈振宇老师、房春荣老师、刘嘉老师和王兴亚老师。其中，我要特别感谢王兴亚老师，我在硕士期间的每一篇成果背后，都有王老师的一份功劳。在硕士论文的每一个环节，他都极其认真严谨地给予我指导和意见，这样的责任心和工作态度使我深受感动和鼓舞。

其次，我要感谢实验室的各位师兄弟姐妹，他们是孙伟松、王新宇、乔力、刘子寒、李紫欣、刘芳潇、段定以及更多我无法逐一列出名字的同学。他们在我需要的时候鼎力相助，没有他们，我的科研之路不会走的如此顺利。

再次，我要感谢三位舍友王栋、王加琪和张伦鸿。感谢他们两年来的包容理解以及在生活中的帮助，让我的硕士生活充满了欢乐和温暖。

最后，我要感谢我的父母，感谢他们的养育之恩。他们一直都在背后默默地支持我、鼓励我，是我最坚强的后盾。不论我遭遇多大的挫折，只要有他们在，我就永远不会倒下。

附件二

《学位论文出版授权书》

本人完全同意《中国优秀博硕士学位论文全文数据库出版章程》（以下简称“章程”），愿意将本人的学位论文提交“中国学术期刊（光盘版）电子杂志社”在《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》中全文发表。《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》可以以电子、网络及其他数字媒体形式公开出版，并同意编入《中国知识资源总库》，在《中国博硕士学位论文评价数据库》中使用和在互联网上传播，同意按“章程”规定享受相关权益。

作者签名：巫浩然

2020 年 5 月 23 日

论文题名	面向 Solidity 语言的变异测试系统设计与实现				
研究生学号	MF1832175	所在院系	软件学院	学位年度	2020
论文级别	☐硕士 ☑硕士专业学位 ☐博士 ☐博士专业学位 （请在方框内画钩）				
作者 Email	hrw.nju@foxmail.com				
导师姓名	刘嘉 副教授				

论文涉密情况：

☑ 不保密

☐ 保密，保密期（_____年_____月_____日 至 _____年_____月_____日）

注：请将该授权书填写后装订在学位论文最后一页（南大封面）。