软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(11):5296−5333 [doi: 10.13328/j.cnki.jos.007386] [CSTR: 32375.14.jos.007386]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

智能合约法律化原则与转化验证模型*

李任翔[1], 蒋忠元[1], 高 胜[2], 钱 肖[1], 沈秀轩[1], 刘柄呈[1], 陶梅悦[1], 马建峰[1]

[1](西安电子科技大学 网络与信息安全学院, 陕西 西安 710071)

[2](中央财经大学 信息学院, 北京 102206)

通信作者: 蒋忠元, E-mail: zyjiang@xidian.edu.cn; 高胜, E-mail: sgao555@163.com

摘 要: 日常民事纠纷中合同类案件占比高, 数量多. 传统纸质合同存在的查阅困难、管理不便等问题, 已经严重影响了合同执行和纠纷处理的效率. 作为一种执行合同条款的计算机协议, 智能合约凭借其自动化执行, 去中心化和不可篡改的优势, 为法律合同的执行处理提供了新的可能. 但智能合约依赖于严格的编程逻辑, 缺乏法律解释的灵活性, 一旦部署后难以动态调整, 限制了合同参与方的意愿, 在法律适用性和约束力方面仍存在不确定性. 因此, 基于法律合同和智能合约的差异对比提出了文法要求、非赋权原则、有效性审查和安全性准则四大原则, 为具有法律效力的智能合约生成和执行提供了理论框架. 此外, 还设计了满足四大原则的智能合约转化与验证模型, 该模型能够对以过渡系统为表现形式的法律合同进行增强处理, 防止重入攻击, 并将基础和额外增加的规范转换为计算树逻辑验证安全属性, 通过验证的合同可以自动转化为智能合约. 在整体流程中, 模型始终遵循四大原则, 因此转化后的智能合约符合现行法律要求, 可被视为法律合同. 在实验验证部分, 采用一个简化的买卖合同作为案例, 展示其初始、增强过渡系统模型, 部分验证属性结果和最终生成的 Solidity 智能合约代码, 并在前置处理中通过采集 64 616 篇合同数据共构建包含 270 592 条样本的高质量数据集, 同时在一致性判定实验中验证了模型性能, *R*@1、*R*@5、*R*@10 分别达到 90.27%、97.91% 和 99.30%, 证明了模型的准确性和可靠性. 结论表明, 提出的四大原则可行性强, 转化验证模型能够解决纸质合同处理的繁琐问题, 提升法律合同执行和管理的便捷性和灵活性, 同时帮助智能合约获得法律保障, 有效规避潜在风险.

关键词: 法律合同; 智能合约; 法律化; 过渡系统; 属性验证

中图法分类号: TP309

中文引用格式: 李任翔, 蒋忠元, 高胜, 钱肖, 沈秀轩, 刘柄呈, 陶梅悦, 马建峰. 智能合约法律化原则与转化验证模型. 软件学报, 2025, 36(11): 5296–5333. http://www.jos.org.cn/1000-9825/7386.htm

英文引用格式: Li RX, Jiang ZY, Gao S, Qian X, Shen XX, Liu BC, Tao MY, Ma JF. Legalization Principles and Transformation Verification Model of Smart Contracts. Ruan Jian Xue Bao/Journal of Software, 2025, 36(11): 5296–5333 (in Chinese). http://www.jos.org.cn/1000-9825/7386.htm

Legalization Principles and Transformation Verification Model of Smart Contracts

LI Ren-Xiang[1], JIANG Zhong-Yuan[1], GAO Sheng[2], QIAN Xiao[1], SHEN Xiu-Xuan[1], LIU Bing-Cheng[1], TAO Mei-Yue[1], MA Jian-Feng[1]

[1](School of Cyber Engineering, Xidian University, Xi'an 710071, China)

[2](School of Information, Central University of Finance and Economics, Beijing 102206, China)

Abstract: Contract cases account for a substantial proportion of daily civil disputes, reflecting a considerable volume. The limited accessibility and cumbersome management of traditional paper contracts have significantly hindered the efficiency of contract execution and dispute resolution. As a computer protocol designed to execute contract terms, smart contracts offer new possibilities for the execution

* 基金项目: 国家重点研发计划 (2022YFB2701800)
收稿时间: 2024-10-26; 修改时间: 2024-11-23; 采用时间: 2024-12-23; jos 在线出版时间: 2025-07-09
CNKI 网络首发时间: 2025-07-10

and processing of legal contracts, with advantages such as automated execution, decentralization, and immutability. However, their reliance on strict programming logic, lack of interpretative flexibility, and difficulty in dynamic adjustments after deployment constrain the intentions of contract participants and result in uncertainties regarding legal applicability and binding force. Based on the distinctions between legal contracts and smart contracts, this study proposes four key principles, including grammatical requirements, the non-empowerment principle, validity review, and security criteria, providing a theoretical framework for generating and executing legally effective smart contracts. A smart contract transformation and verification model is further designed to adhere to these four principles. The proposed model enhances the processing of legal contracts expressed as transition systems, prevents re-entry attacks, and converts core and additional specifications into computational tree logic for security property verification. Contract passing verification is automatically converted into smart contracts. The entire transformation process complies with the proposed four principles, ensuring that the resulting smart contracts meet current legal standards and can be regarded as legal contracts. Experimental validation includes a simplified sales contract as a case study, demonstrating its initial and enhanced transition system models, partial verification results, and the representative Solidity code generated. The pre-processing operation yields a high-quality dataset constructed from 270 592 samples. Consistency evaluation between contract terms and legal provisions achieves Recall rates of 90.27% at *R*@1, 97.91% at *R*@5, and 99.30% at *R*@10. The feature extraction model, aided by a format conversion tool with nearly 100% fidelity, achieves 91.87% accuracy at the token level, confirming the model's accuracy and reliability. The findings indicate that the proposed principles are highly feasible, while the transformation and verification model effectively addresses the cumbersome nature of paper contract processing, enhances the convenience and flexibility of legal contract execution and management, and enables smart contracts to obtain legal protection while mitigating potential risks.

Key words: legal contract; smart contract; legalization; transition system; property verification

法律合同被定义为两个或多个当事人之间达成的具有法律约束力的协议[1], 自然语言撰写的传统纸质合同是其最常见形式. 2023 年的最高人民法院审判执行工作数据表明, 民商事一审收案 1 753.05 万件, 数量排名前 5 中的第 1 (借款)、第 3 (买卖) 和第 4 (物业服务) 都属于合同纠纷案件, 占比高达近 40% (https://www.court.gov.cn/zixun/xiangqing/427552.html). 因此法律从业人员在审理这些纸质合同案例时, 通常需要花费 2–4 h 来查找和整理相关条款和证据, 这种高昂的时间成本显著降低了处理效率.

作为一种自动执行的计算机程序, 智能合约[2–4]依托区块链技术的去中心化、不可篡改等特性, 已在多个领域中取得了广泛应用. 例如, 利用智能合约保护与共享档案[5], 在物联网环境中实现访问控制和数据管理[6], 以及进行可信数据的存储和管理[7]等. 这些研究展示了智能合约在数据保护、权限控制和信息管理中的独特技术优势. 因此, 智能合约有能力为法律合同的处理提供一种新的解决方案. 法律合同要素, 如当事人信息、合约条款、履行状态等均能以透明可查的数字形式通过智能合约编码上传至区块链平台. 这样不仅可以使合同在自动执行过程中减少第三方干预, 还能方便其在争议发生时即时调取履行记录等相关证据, 确保其不可篡改和可追溯性, 降低时间成本.

然而, 由于智能合约依赖于编程语言执行条款, 其法律适用性和灵活性仍存在不确定性. 为了解决此问题, 本文首先总结了法律合同与智能合约的关系和等效条件, 其次从多方面详细分析传统纸质合同和智能合约的区别, 归纳出智能合约法律化的四大原则: 文法要求、非赋权原则、有效性审查和安全性准则, 然后基于这些原则设计了一个智能合约的转化和验证模型 LTSC-TAV (legal to smart contract transformation and verification model), 该模型能够通过计算树逻辑对法律合同进行形式化验证, 并将其自动转化为具有法律效力的智能合约, 接着论述四大原则在 LTSC-TAV 模型中的体现. 实验方面, 通过对一个简化买卖合同案例的讲解和前置处理中合同集信息的展示, 以及一致性判定、要素处理等操作性能指标的对比, 验证了模型的安全性和可靠性. 最后总结了成果和未来方向. 后文图 1 展示了本文的基本流程.

1 智能合约的法律化探索

具有自动执行特性的智能合约与传统纸质合同在形式和执行方式等方面皆存在差异, 因此智能合约的法律化探索首先需要明确智能合约的定义, 接着再深入探讨两者法律层面之间的关系. 该讨论能够帮助分析法律适应新技术的能力, 维持技术创新和权益保护间的平衡. 最后提出智能合约法律化的原则, 确定智能合约的法律地位和法律责任.

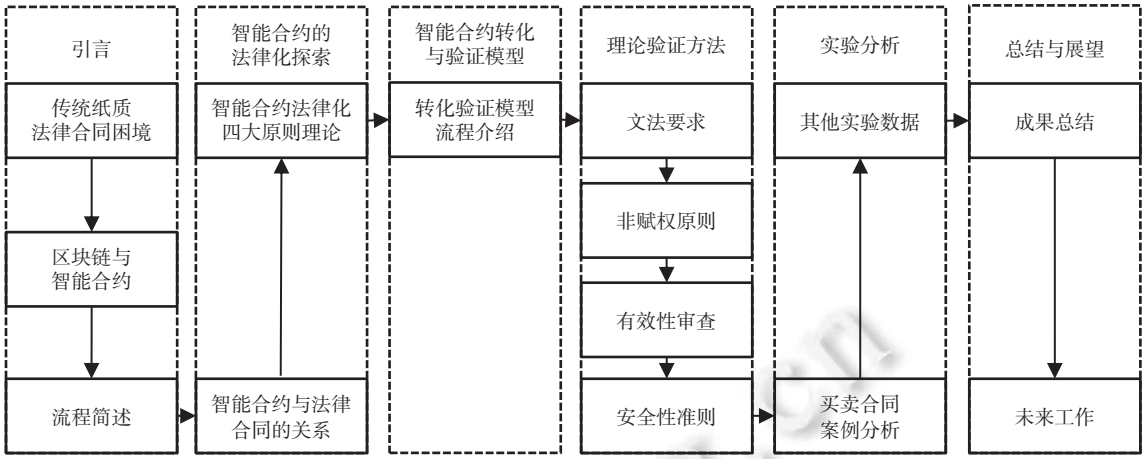

图 1 本文流程图

1.1 智能合约的定义

智能合约广义上是指符合当事人之间约定的任何计算机协议. 它通过程序代码和自动执行手段, 将合同条款转化为机器可读的指令, 并遵循合约的可信性与合法合规性, 由参与者共同协作完成, 满足约定的智能合约才可以正常执行. 因此, 智能合约是一种计算机化的合约.

为了明确智能合约的法律化属性, 在维基百科中, 智能合约的定义是一种旨在根据合约或协议中条款自动执行、控制或记录与法律相关事件和动作的计算机程序或交易协议. 该定义表明, 智能合约以法律合约为对象, 通过计算机程序或交易协议促进、保障、验证、加强合约的协商和履行.

此外, 区块链智能合约定义是部署在区块链上、在满足预定条件时可自动执行并存证的计算机程序.

与广义的智能合约相比, 区块链智能合约是一种自动执行与存证的计算机程序, 需针对区块链结构转化为平台指令代码. 代码在满足触发条件后自动执行, 不需要可信方的参与. 因此, 智能合约是一种自动执行合约条款并自我验证和无需中介的计算机交易协议.

1.2 智能合约与法律合同的关系

关于智能合约是否属于法律意义上合同的问题, 学术界一直存在争议. 部分学者[8,9]支持将智能合约视为法律合同, 他们认为智能合约部署和执行过程中完全基于当事人确认, 即要约方发布合约代码, 向承诺方提出明确的合同条款, 承诺方参与表示接受, 从而形成有效的合约关系, 该流程满足了传统法律合同中的“要约-承诺”结构, 能有效表达当事人之间的合意. 此外智能合约无需第三方中介即可自动执行, 降低了传统合同人为干预和违约的风险, 提高了执行效率和可靠性.

然而, 较多学者[10,11]认为智能合约并不能被直接视为法律合同, 明确智能合约的法律地位需要分析其作为合同的有效性. 首先, 智能合约的信赖基于技术及自治秩序, 而非传统的法律框架. 其次, 智能合约代码的执行无法涵盖所有法律合同, 缺乏灵活性和适应性. 虽然智能合约中根据预设条款自动绑定的执行方式符合法律合同的某些特性, 但代码难以解释, 无法保证法律效果与传统法律合同的要求相符. 此外, 智能合约一旦部署后不可撤回, 无法确保合同双方意愿的动态变化, 会造成强制缔约的现象. 这种情况下, 智能合约虽然在技术上确保了条款的严格执行, 但缺乏了法律合同所需的调整能力, 因此难以被直接认作合同.

《中华人民共和国民法典》(以下简称《民法典》) 第 464 条规定: “合同是民事主体之间设立、变更、终止民事法律关系的协议.” 在智能合约签订过程中, 当事人也约定了各方应该遵循的行为规范, 两者目的、性质一致[11]. 然而, 如果想被视为法律合同的一种新的表现形式, 智能合约在执行过程中就必须保证当事人各方的想法可以充分且随时表达, 在编写和运行过程中, 其代码形式和内容表现也需要满足现有法律和政策框

架的约束.

本文认为, 只有在满足特定属性, 确保技术执行的严格性和安全性的基础上, 兼顾了条款解释的一致性和公平性、灵活应对变化的能力, 智能合约才可以被视为有效的法律合同. 本文将关注于如何将智能合约的技术优势与法律合同的基本原则相结合, 确保其在提供高效合同执行机制的同时, 满足公平性、适应性和解释一致性的要求.

1.3 智能合约的法律化原则

目前在满足合同的法律规定方面, 智能合约仍存在诸多问题. 下文将深入探讨两者间的本质差异, 通过对比分析明确可视为法律合同的智能合约所需符合的四大原则.

1.3.1 文法要求

传统法律合同与智能合约的差异直观体现在文法表达方式和语言特性上, 传统法律合同通常以自然语言表述为基础, 包含各种法律术语和专业词汇, 而智能合约使用计算机代码. 具体区别如下.

(1) 基于自然语言的特点, 传统法律合同具有高度的灵活性和适应性, 大量的法言法语和专业领域术语让传统法律合同的内容抽象, 具有概括性, 适用于多变的环境和未预见的情况.

(2) 相反, 作为严格定义的可执行的计算机指令序列[12], 智能合约代码逻辑性强, 表达方法固定和形式化, 具有非二义性、确定性的特点, 条件界限和范围严格, 执行过程十分具体.

因此, 相比于传统法律合同, 依赖于严密逻辑编程的智能合约减少了由模糊的自然语言导致的解释争议. 但与此同时, 智能合约适用范围窄, 对多变的现实情况无法准确快速地判断和修改, 较为死板.

此外, 智能合约的表达能力相比于自然语言差, 难以表述特定的责任关系和抽象的专业词汇. 面对当事人的权利与义务, 解释说明, 违约责任划分等条款, 现有代码语言很难做到全面映射[13]. 这种表述上的不足直接影响智能合约的法律效力.

《民法典》第 470 条中规定: “合同的内容由当事人约定, 一般包括下列条款: (一) 当事人的姓名或者名称和住所; (二) 标的; (三) 数量; (四) 质量; (五) 价款或者报酬; (六) 履行期限、地点和方式; (七) 违约责任; (八) 解决争议的方法.” 以上条款是法律合同转化的基础要素, 在智能合约法律化过程中必须保留.

综上所述, 文法要求原则可概括为: 智能合约法律化应遵循以自然语言表述为基础、法言法语为标准词汇、计算机形式化表达为语法、法律要素为框架, 生成智能合约.

1.3.2 非赋权原则

除了表达方式和语言, 智能合约和传统合同的执行过程也存在较大差异. 智能合约拥有自动执行的机制, 而法律合同在执行的过程中往往伴随着动态调整.

作为智能合约最显著的特性之一, 自动执行功能展示了智能合约的高度精确性和自动化验证能力. 但这种能力也存在局限性, 一旦激活预设条件, 智能合约将持续执行, 即使有必要的调整或立即停止的需求, 也无法暂停或终止, 这种执行逻辑上的机械运行难以满足多变的实际情况.

在传统法律合同签订时, 各方已约定了法律责任的承担者. 在智能合约法律化时, 计算机不能承担法律责任[14], 智能合约在执行时应保证其行为得到各方的授权. 智能合约所具有的自动执行能力与法律合同中的当事人享有权利之间的矛盾主要体现在:

(1) 智能合约的自动执行功能不应覆盖或取代当事人的自主决策权.

(2) 智能合约不应阻碍或制约人类行为自由, 不应强制当事人做或避免某些行为.

因此, 智能合约不应具备完全的自主权或独立执行权. 在设计智能合约时, 必须限制其自动执行功能, 确保其不替代当事人的自主决策权. 拥有当事人明确授权的前提下, 智能合约才可以在授权范围内自动代行特定行为, 比如处理转账交易, 触发付款和发货操作等基础活动, 以展示其优势.

此外, 《民法典》第 180 条中明确指出: “因不可抗力不能履行民事义务的, 不承担民事责任. 法律另有规定的, 依照其规定.” 不可抗力或特殊情况的应对方法也是智能合约的重要组成部分, 在取得各方的一致认定的结果后, 可以暂停, 终止履行的合约或进行其他处理.

综上, 非赋权原则可表述为: 智能合约法律化过程中应注意约束智能合约不可代替当事人自动执行意志选择, 应针对当事人行为结果进行条款履行的检测和验证, 并存在不可抗力或特殊情况的处理方法.

1.3.3 有效性审查

传统法律合同通常由法律相关人员按照规定制定模板, 由当事方补充后完成, 具有法律效力, 而智能合约通常由相关专业技术领域人员编写, 这些人员普遍缺乏对法律条款的充分理解, 可能将不符合法律规定的条款错误地编码成合约, 或未能识别出违反法律规定的合约. 因此智能合约存在法律有效性问题.

智能合约是一种特殊的计算机程序, 与常规代码结构无异. 智能合约代码通常由两部分构成.

(1) 引用性代码. 该类代码广泛存在于智能合约中, 预先编写, 方便重复使用, 如各种库等.

(2) 针对性代码. 该类代码主要表现合同的专有内容, 通常包含重要信息.

《民法典》第 496 条定义: “格式条款是当事人为了重复使用而预先拟定, 并在订立合同时未与对方协商的条款. 采用格式条款订立合同的, 提供格式条款的一方应当遵循公平原则确定当事人之间的权利和义务, 并采取合理的方式提示对方注意免除或者减轻其责任等与对方有重大利害关系的条款, 按照对方的要求, 对该条款予以说明. 提供格式条款的一方未履行提示或者说明义务, 致使对方没有注意或者理解与其有重大利害关系的条款的, 对方可以主张该条款不成为合同的内容.” 引用性代码和格式条款的定义和作用相似, 因此可以将其归为格式条款的特殊表现形式. 根据规定, 针对引用性代码, 需对当事人进行告知并达成共识.《民法典》第 497 条中也提到: “有下列情形之一的, 该格式条款无效: (一) 具有本法第一编第六章第三节和本法第五百零六条规定的无效情形; (二) 提供格式条款一方不合理地免除或者减轻其责任、加重对方责任、限制对方主要权利; (三) 提供格式条款一方排除对方主要权利.” 引用性代码也应进行无效性检测, 防止以上情形.

针对性代码中包含了合同的主要内容, 体现了法律的效力, 因此主要审查是否存在《民法典》中规定的无效民事法律行为, 如: 欺诈、胁迫、虚假含义的表示等.

综上所述, 有效性审查原则可概括为: 智能合约法律化过程中应对引用性代码进行法律效力审查, 对针对性代码当事人进行告知并达成共识, 并进行无效性检测.

1.3.4 安全性准则

传统法律合同的内容依赖于法律人士的解释, 在签订和执行过程中受到现行法律框架的保护和监督, 通过法院审理来保障合同的合法性和执行公正, 且不涉及复杂的技术实现, 风险低, 不存在由软件或编程错误导致的安全漏洞, 而智能合约则存在安全问题.

作为智能合约的载体, 区块链平台的可信度仅可以保证智能合约正确执行, 不保证代码正确性. 目前大量已部署的合约因软件漏洞而无法正常使用. 这些漏洞通常由合约编写者的语义理解偏差产生, 或恶意编程人员的故意编写. 由于智能合约执行便捷, 缺乏对代码深层理解的当事人通常只是按照规定行使权利和履行义务, 无法发现合约中看似相同但实际差异巨大的表达, 从而无意中落入由恶意编程人员设下的陷阱. 恶意编程人员即可利用不对等信息的优势进行欺诈、胁迫等违法行为. 当执行人意识到权益损害时, 智能合约通常已执行完成. 由于缺乏维权证据, 若智能合约不符合规定, 不被认定为法律合同, 当事人也难以借助法律工具维权.

此外, 区块链平台不允许更新合约代码以修复漏洞或撤销恶意交易, 因此智能合约漏洞风险进一步加剧. 为了应对智能合约的安全问题, 目前有些开发者汇总后端代码到库合约中供前端合约引用, 并在库中部署新实例更新前端引用, 通过分离代码的方式绕过智能合约的不可变性. 然而, 一个可变合约可能会执行其任何条款. 合约条款的不透明和可变性增加了安全和信任问题. 此外, 也有通过执行区块链的硬分叉撤销交易的方案, 但硬分叉需要整个平台的利益相关者间共识, 破坏了平台可信度, 并且也可能引入安全问题, 如原始链和分叉链之间的重放攻击等. 因此, 以上方法认可度均不高. 因此, 在生成智能合约之前进行全面的安全性检测至关重要, 智能合约安全性检测需求迫在眉睫. 通过预先进行的安全性验证, 可以有效发现并修复潜在的逻辑漏洞和安全隐患, 防止恶意利用或无意中的错误编写对合约的合法性和执行带来风险.

综上所述, 安全性准则可表述为: 智能合约法律化过程中应事先对存在的逻辑问题和安全漏洞进行检测和

验证.

1.3.5 四大原则的充分性和必要性

基于前文分析, 可将传统法律合同与智能合约的差异及得出原则归纳为表 1.

表 1 法律合同与智能合约之间的差异对比

对应原则	法律合同	智能合约
文法要求	二义性 专业术语, 抽象	确定性 严谨正式, 具体
非赋权原则	当事人意志选择 特殊情况停止合约	自动执行 无法停止执行
有效性审查	已审查和执行 具有法律效力	难以审查 存在违法和无效行为
安全性准则	不存在安全隐患	隐私保护, 存在漏洞

智能合约法律地位和法律责任的确立, 是保证其在现实法律体系中有效应用的关键. 该过程不仅需要技术方面的可行性, 更需要法律意义上的正当性. 以上提出的四大原则即是实现这一目标的充分且必要的条件, 是理论框架的核心.

文法要求确保智能合约能够以形式化语言严谨地描述传统法律合同信息, 在语法上满足现行法律规范, 并通过自然语言与计算机语言的有效映射, 弥补传统合同解释灵活性与智能合约执行刚性间的差距. 缺乏文法要求, 智能合约将会因法律语言的欠缺而无法完整表达当事人的合意, 导致合同的无效化.

非赋权原则限制了智能合约的自主决策权, 展现了合同各方的真实意愿, 维护了当事人的法律主体性, 与法律对自由意志的保护理念相符. 忽略非赋权原则, 或将令智能合约违背当事人意志, 造成法律责任不明确.

有效性审查保证智能合约在生成与执行阶段符合现行法律法规, 消除可能存在的无效条款或风险冲突. 若无有效性审查, 智能合约有包含非法条款或执行错误的风险, 从而丢失其法律影响力.

安全性准则通过形式化验证和漏洞检测等方法, 为智能合约提供了技术安全保障, 防止因安全漏洞导致的执行失败或法律效力丧失. 安全性准则的缺失则可能导致智能合约因技术漏洞被滥用, 直接威胁合约的履行过程.

此外, 四大原则之间具有内在的逻辑关联. 文法要求提供了智能合约生成的语言基础; 非赋权原则限定了智能合约执行的范围, 保证合同执行的合法性与灵活性, 进一步细化了文法要求的具体应用; 而有效性审查与安全性准则分别从内容和技术角度确保智能合约的法律效力和执行安全. 四大原则相辅相成, 共同构筑起一个多层次、全方位的法律保障体系.

由此可得, 满足文法要求、非赋权原则、有效性审查和安全性准则四大原则的规范生成的智能合约才可以被视为法律合同, 具有法律效力.

2 智能合约转化与验证模型

第 1.2 节分析了传统法律合同与智能合约的差异, 总结了生成可视为法律合同的智能合约所需的四大原则. 满足四大原则, 直接根据法律合同生成智能合约是最佳方案, 可以在文法要求原则论证上节约大量时间. 但目前传统纸质合同转化至计算机智能合约代码不存在完整的标准转化方式, 只存在针对某一过程的思路研究, 且转化的过程不具体, 转化结果也因人而异, 质量参差不齐. 为了满足本文提出的智能合约法律化四大原则, 本文构建了一个完整的具有法律效力的智能合约转化与验证模型 LTSC-TAV.

LTSC-TAV 模型是一个开源的基于 WebGME[15]和 FSolidM[16,17]构建的框架, 充分体现了四大原则, 保证生成智能合约的准确性, 有效性和安全性. 图 2 显示了 LTSC-TAV 模型的设计流程步骤. 其中, 用实线箭头表示必须进行的强制步骤, 用疏虚线箭头表示可选步骤, 用密虚线表示相关步骤的详细介绍, 并使用矩形封闭虚线和左上角内置文字说明一些统一的环节. 下面将按照先后顺序对 LTSC-TAV 模型进行介绍.

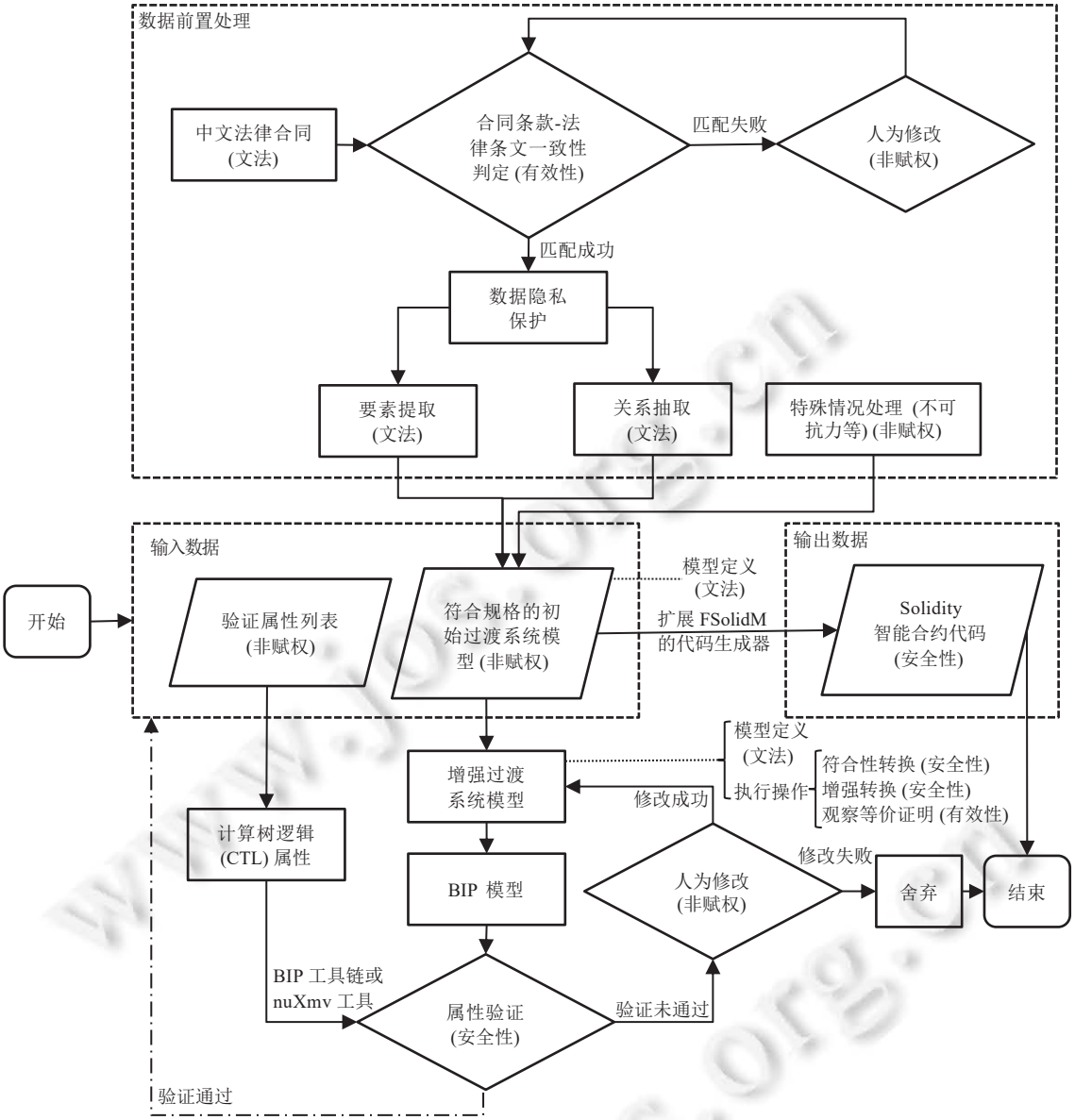

图 2 模型转化与验证工作流程图

模型数据在正式输入前, 需要进行前置处理.

(1) 基于收集的原始中文法律合同, 构建原始数据集.

(2) 匹配法律合同条款与法律条文, 进行一致性判定, 详见第 3.3.1 节的具体描述.

(3) 如果需要对数据隐私进行保护, 可以对敏感要素识别和加密.

(4) 提取合同要素信息, 详见第 3.1.1 节的具体描述.

(5) 抽取合同的逻辑关系, 与 (4) 执行时间并列.

(6) 完善特殊情况的处理方法. 前置处理帮助构建 LTSC-TAV 模型输入数据的基础框架.

模型开始运行后, 需要使用者输入初始数据.

(1) 符合指定 Solidity 合约规格的初始过渡系统模型. 其中合约规格包括变量声明、操作和守卫 (本文中“守卫”

指在过渡系统中用于限制状态转移的条件约束, 确保状态变化符合预设逻辑和安全性要求). 该模型基于前置处理中提取的要素信息, 抽取的逻辑关系和特殊情况的处理方法组合构建, 指定为图形形式, 详见第 3.1 节的具体描述.

(2) 一组需要验证的属性列表.

接下来将开始验证过渡系统的循环. 如果验证指定属性的过渡系统不能满足验证条件, 可以选择生成一个增强的过渡系统合约模型. 接着无论是否增强, 模型将基于过渡系统自动生成一个行为-交互-优先级 (behavior interaction priority, BIP) 模型. 与此同时, 模型将开始时输入初始数据中的指定属性自动转换为计算树逻辑 (computation tree logic, CTL). 然后, BIP 模型和 CTL 属性输入 BIP 工具链[18]或 nuXmv[19]工具中, 进行死锁自由或其他属性的验证. 最后, 满足了所有指定属性后, 属性验证通过, 模型将基于过渡系统自动生成符合四大原则的 Solidity 智能合约代码, 详见第 3.4 节的具体描述.

3 理论验证方法

第 2 节中对 LTSC-TAV 模型的整体流程进行了介绍, 下面将详细描述四大原则的具体体现.

3.1 文法要求

在 LTSC-TAV 模型流程中, 文法要求主要体现在前置条件中原始法律合同数据集的构建, 要素提取和关系抽取, 以及对初始和增强过渡系统的形式操作语义和属性的定义.

3.1.1 基础、框架和语法

文法要求的基础是自然语言表述. 前置条件中以各地市政府采购官方网站为来源收集原始中文法律合同构建. 许多初始合同以照片或图像 PDF 的形式呈现, 无法被直接处理, 因此还需对合同文档预处理, 包括格式转换、文本清洗、分词、标准化和向量化等, 为后续提供干净统一的原始输入数据集. 处理后的数据集保留了自然语言的原始特性, 确保了智能合约的转化基础.

文法要求的框架是法律要素. 为克服从合同文本中识别要素信息的困难, 前置条件的要素提取部分采用自然语言处理领域大型模型技术, 并基于收集到的合同数据特征开发创新的特征提取技术、深度学习架构和训练策略. 这包括了一种基于合同上下文的填充机制 (CDPM)、三头注意力机制[20]以及边缘加权交叉熵损失函数[21,22]. 结合上述方法与主流深度学习框架, 研究实现了一种高效的买卖合同要素提取模型, 该模型基于序列标注 (又称令牌分类) 原则进行操作.

文法要求的语法是法言法语. 合同行为之间的交互逻辑关系由前置处理的关系抽取部分完成, 需要根据要素条款生成归一化法律合约和状态转移图系统[23]. 首先构建归一化法律合约模板, 模板包含了常见合同中的标准化条款, 并为特殊化条款保留了补充位置, 进而形成过渡系统模型, 建立输入数据中初始过渡系统的基本框架, 如后文图 3 所示. 流程中付款置于交付前且为一次性, 违约后协商成功时的合同状态转移未标明, 这些都需要视实际情况而定. 然后将要素条款和模板中的条款匹配. 通过特定法律语言模型深度语义编码, 将合同文本和相关模板条款转换为高维词向量, 并利用条款词向量之间的余弦相似度计算匹配正确率, 进行语义检测. 构建模板时, 综合考虑了法律条款的定义和常见法律合同的描述方式, 按照用法语义规范广泛使用专业词汇, 保证了法言法语语义的展现.

3.1.2 表达方法

遵循计算机法学和语言学规范的形式化方法表达是文法要求的重中之重. 在 LTSC-TAV 模型中, 文法要求的表达方式体现如下.

(1) 基于形式操作语义定义了初始过渡系统和增强过渡系统.

过渡系统 (transition system) 是形式化方法中的基本概念, 用于描述系统所有可能的状态及其状态间的转换关系[24]. 过渡系统为理解和分析整体状态和行为关系提供了一个结构化的框架. 在 LTSC-TAV 模型中, 所定义的初始与增强过渡系统, 为系统状态及其动态行为的描述与分析提供了坚实的理论框架. 为了按照计算机形式化表达

规范将合约正式定义为一个过渡系统, LTSC-TAV 模型引入了 Solidity 语句的一个子集, 具体在附录 A.1 中详细描述. 这些子集包含了所有基本的控制结构: 循环、选择和返回语句, 是一个图灵完备的子集, 可以直接扩展以捕获所有其他的 Solidity 语句. 表 2 中总结了 Solidity 代码的符号表示方法.

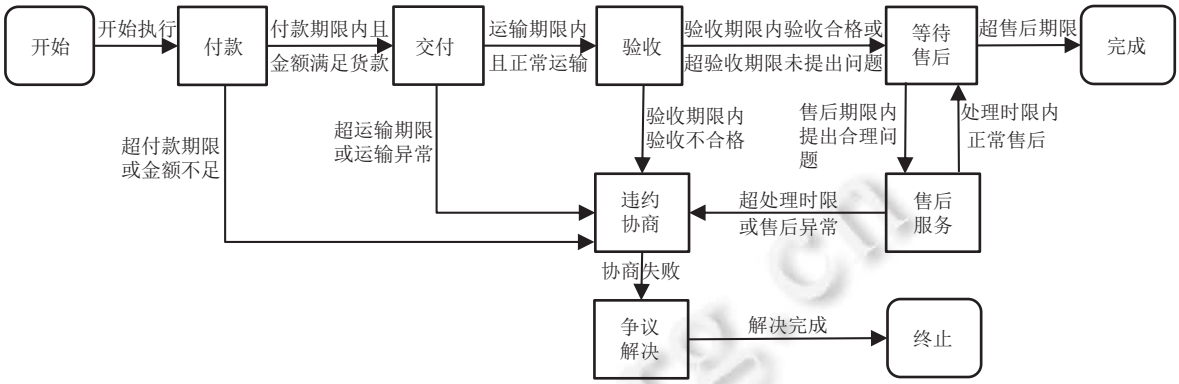

图 3 中文法律买卖合同过渡系统结构模板样例

表 2 Solidity 代码的符号表示方法

符号	含义
T	Solidity类型集合
I	有效的Solidity标识符集合
D	Solidity事件和自定义类型定义集合
E	Solidity表达式集合
D	没有副作用的Solidity表达式集合
S	支持的Solidity语句集合

定义 1. 一个初始过渡系统表示的智能合约是一个元组 $(D,S,S_F,s_0,a_0,a_F,V,T)$.

其中, $D\subset\mathbf{D}$ 是自定义事件和类型定义的集合, 对应合同中可以触发某些操作的条款 (如权利义务、违约条款等) 和不同的条款类型 (如付款条款、交付条款、赔偿条款等).

$S\subset\mathbf{I}$ 是状态的有限集合, 对应合同不同阶段和履行状态, 如“合同生效”“待付款”“待交货”“合同结束”等.

$S_F\subset\mathbf{S}$ 是最终状态的集合, 对应合同的履行完成、解除、违约终止等多种终止状态.

$s_0\in\mathbf{S}$, $a_0\in\mathbf{S}$ 是初始状态和操作, s_0 可以视为合同的“起始状态”, a_0 则对应合同的“生效条件”, 即双方签署后合同生效的过程. 如无生效条件, a_0 可以省略.

$a_F\in\mathbf{S}$ 是回退操作, 对应合同的“备用条款”或“违约处理条款”. 法律合同执行中如果出现了违约或某些无法预见的情况, 会触发合同中的应急处理条款 (如违约金支付、解除合同等), 这类似于智能合约中的回退功能 a_F.

$V\subset\mathbf{I}\times\mathbf{T}$ 是合约变量 (即变量名和类型), 对应法律合同中的合同要素, 如当事人信息、合同金额、交付物品信息等.

$T\subset\mathbf{I}\times\mathbf{S}\times2^{\mathbf{I}\times\mathbf{T}}\times\mathbf{C}\times(\mathbf{T}\cup\varnothing)\times\mathbf{S}\times S$ 是一个过渡关系, 每个属于 T 的过渡包含: 过渡名称 $t^{\text{name}}\in\mathbf{I}$, 源状态 $t^{\text{from}}\in S$, 参数变量 (即参数) $t^{\text{input}}\subseteq\mathbf{I}\times\mathbf{T}$, 过渡守卫 $g_t\in\mathbf{C}$, 返回类型 $t^{\text{output}}\in(\mathbf{T}\cup\varnothing)$, 操作 $a_t\in\mathbf{S}$, 目标状态 $t^{\text{to}}\in S$. T 对应合同中的条款执行逻辑或履行路径. 当法律合同执行时, 一些条件触发后合同通常会进入下一阶段, 例如甲方付款后乙方开始交货, 这些条件和状态转移可以通过 T 来表示.

在模型中, 初始操作 a_0 代表智能合约的构造函数, 用指向初始状态的箭头图形表示. a_0 为空, 即没有构造函数时, 可在状态过渡系统中省略 a_0, 回退操作 a_F 代表合约的回退功能. 每个合约最多存在一个构造函数和一个回退函数. Solidity 代码的回退功能会在附录 B.1 中进行深入讨论.

以初始过渡系统模型为表现形式的 Solidity 智能合约输入后, 可选择将其自动转化为功能等效的增强模型方便输入验证工具中进行属性验证. 与定义 3 相似, 增强过渡系统也可以被定义为规范的形式化表达方式.

定义 2. 生成的增强过渡系统表示的智能合约是一个元组 (D,S,S_F,s_0,V,T).

与初始系统相比, 增强系统不再需要初始操作 a_0 和回退操作 a_F, 其他部分保持一致.

(2) 扩展了 FSolidM 模型的语法, 为模型和支持的 Solidity 语句提供了形式操作语义.

LTSC-TAV 模型以结构化操作语义 (SOS) 规则[25]的形式定义了表现为过渡系统的智能合约的操作语义. 在操作语义规则中, 使用 Ψ 表示账本状态, 包括账户余额、所有合约中状态变量的值、最后一个区块的编号和时间戳等. 执行过渡时, 执行状态 σ 表示为 $\{\Psi, M\}$, 除了账本状态外, 执行状态也包括内存和堆栈状态 M. 为了处理返回语句和异常情况, 规则还引入了一个特殊的执行状态, 当发生异常时该状态为 E, 当返回语句带有值 v 执行时为 $R[v]$, 表达返回值为 v, 不带有返回值时则为 N. 最后表达式为 $\text{Eval}(\sigma,\text{Exp}) \to \left\langle\left(\hat{\sigma},x\right),v\right\rangle$, 含义是在执行状态 σ 中评估 Solidity 表达式 Exp, 结果将产生值 v, 并将执行状态改变为 $\hat{\sigma}$, 执行情况为 x.

LTSC-TAV 模型通过提供过渡 (即函数) 名 $name \in \mathbf{I}$ 和参数值列表 $v_1,v_2,\ldots$ 来触发过渡. 正常执行的不返回任何值的过渡会将账本状态从 Ψ 变为 Ψ', 将合约的状态从 $s \in S$ 变为 $s' \in S$. 该种类型的过渡会由 TRANSITION 规则捕获.

$$t \in T, name = t^{\text{name}}, s = t^{\text{from}}$$

$$M = Params(t,v_1,v_2,\ldots), \sigma = (\Psi, M)$$

$$\text{Eval}(\sigma, g_t) \to \left\langle\left(\hat{\sigma},N\right),\text{true}\right\rangle$$

$$\left\langle\left(\hat{\sigma},N\right),a_t\right\rangle \to \left\langle\left(\hat{\sigma}',N\right),\cdot\right\rangle$$

$$\hat{\sigma}' = (\Psi', M'), s' = t^{\text{to}}$$

$$\text{TRANSITION} \text{ ----------------}$$

$$\langle(\Psi,s), name(v_1,v_2,\ldots)\rangle \to \langle(\Psi',s',\cdot)\rangle$$

TRANSITION 规则解释为: 存在一个过渡 t, 其名 t^{name} 为 $name$ 且原始状态 t^{from} 为 s (第 1 行), 堆栈状态 M 取参数值 $Params(t,v_1,v_2,\ldots)$, 通过堆栈状态 M 和当前账本状态 Ψ 初始化执行状态 σ (第 2 行). 如果守卫条件 g_t 在当前状态 σ 中评估为真 (第 3 行), 那么过渡的操作语句 a_t 将被执行 (第 4 行), 以上过程将导致执行状态更新为 $\hat{\sigma}'$ (见附录 A.3 中的语句规则). 最后, 如果结果执行状态为正常 N, 没有抛出异常, 那么更新的账本状态 Ψ' 和更新的合约状态 s' (第 5 行) 将被确认.

除了 TRANSITION 规则捕获的不返回任何值的过渡, LTSC-TAV 模型还对其余所有错误过渡的执行情况定义了 SOS 规则及返回值, 如: 守卫评估期间引发异常, 过渡被还原等, 详细内容见附录 A.2. 此外, 在附录 A.3 中还为受支持的语句定义了 SOS 规则. 通过对初始和增强过渡系统的定义以及形式操作语义的构建, 以计算机形式化表达方法的语法规范得以体现, LTSC-TAV 模型能够生成严谨完善的智能合约, 充分保证了文法要求原则的实现.

3.2 非赋权原则

LTSC-TAV 模型依照完整的法律合同生成智能合约, 这些合同已经拥有当事人的行为, 因此非赋权原则主要体现为在关键决策节点上明确标注, 以确保使用者的自主决策权, 并融入全流程设计理念, 使得合同在复杂场景中既能灵活适应实际需求, 又保持了法律约束力和自动化优势.

(1) 前置处理中一致性判定失败后的人为处理环节. 在前置处理阶段检测到合同文本与法律条文间匹配不一致判定为失败时, 系统并非自动放弃该合同或将其强制执行, 而是提供了人工干预步骤, 允许合同当事人对合同内容修正, 从而保证合同在正式执行前, 能够更准确地反映各方的真实意图. 在一致性判定失败的情形下, 系统提供了基于自然语言处理技术的法律条款修订工具, 帮助合同当事人直观地查看冲突条款的具体内容及潜在问题. 同

时附加了基于法律条文的智能提示明确调整方向, 例如, 当涉及不可抗力条款的履约冲突时, 系统会自动定位相关条款并提供预定义的法律解决方案供当事人选择. 此外, 系统允许用户在人工干预过程中实时验证新修订条款的法律合规性, 以减少多次修订的复杂度. 该模块大大提高了智能合约的灵活性, 保证了追求自动化执行的同时, 尊重当事人主观判断.

(2) 前置处理中组合生成初始过渡系统时, 额外添加特殊情况处理模块. 特殊情况处理模块以法律条款中的例外条款为依据, 针对可能出现的不可抗力、政策调整或合同履行困难等情形, 提供灵活的状态转移机制. 系统不会一刀切地执行预设的条款, 而是提供了协商机制, 允许各方在面对特殊情况时共同商讨应对策略. 例如, 在合同履行期因市场波动导致履约条件变化时, 系统支持合同各方通过协商动态调整履行条款, 并在智能合约中实时更新对应的执行逻辑. 这种设计不仅增强了合同的法律灵活性, 也显著提高了智能合约在复杂环境中的适应能力, 同时通过状态记录和修改日志的方式, 确保调整过程的透明性和可追溯性. 生成的智能合约中, 特殊情况的处理具体体现为, 参与各方协商后都有权在每个状态下直接跳转到终止状态. 这种方法不仅保护了各方的合法权益, 也能帮助智能合约更好地适应复杂多变的实际情况, 避免出现僵化或不合理结果.

(3) 属性验证未通过后的处理环节. 与一致性判定类似, 验证结果未能通过或与预期不符时, 系统引入了多层次的复核机制. 一方面, 用户可以通过逐条复查属性验证的具体错误信息, 结合系统生成的详细报告判断问题所在; 另一方面, 系统提供了基于规则的修订建议, 例如, 当发现金额条款与履约能力冲突时, 系统会推荐调整具体数额或增加分期履行条款. 此外, 为增强验证的准确性, 系统支持基于案例学习功能, 通过分析历史合同修订案例, 生成优化的条款逻辑. 此设计提高了属性验证的通过率, 避免了因自动化执行带来的潜在法律风险, 同时也使实际操作更加灵活和人性化.

(4) 其他环节. 除了上述 3 个主要环节, 非赋权原则还贯穿于整体执行流程的设计中. 如: 基于要素提取, 关系抽取和特殊情况处理的初始法律过渡系统和需要验证的属性, 在满足文法要求的前提下, 可以自由发挥; 除了预设的基础安全属性, 使用者可以添加任意符合逻辑和规范的属性进行验证, 等等.

综上所述, 非赋权原则通过多层次的干预设计和灵活的执行机制, 不仅有效解决了智能合约与法律灵活性之间的矛盾, 也在法律合规性、执行安全性和用户友好性之间取得了平衡. 通过多模块协同运行, 系统在保证自动化执行的基础上, 为合同当事人提供了充分的自主决策权, 提升了合同在实际应用中的适应性和可操作性. 这种设计为智能合约在法律场景中的推广提供了理论和技术支持.

3.3 有效性审查

在 LTSC-TAV 模型中, 智能合约的生成源自官方认定的法律合同, 这些合同内容规范, 方便学习和参考. 因此告知共识的内容不必再进行分析, 重点应放在法律效力和无效性审查中.

智能合约代码的计算机形式化表达方法与传统法律合同的自然语言之间差异巨大, 难以直接从代码层面进行有效性审查. 因此, LTSC-TAV 模型采用优先判别的思路, 在智能合约验证和生成的过程中检测, 再基于完善的代码生成系统, 即可实现对代码有效性的审查.

3.3.1 合同条款与法律条文的一致性判定

一致性判定是有效性审查思想的直接体现, 不仅需要处理大量的合同条款和法律条文, 还要通过深度语义分析和高效计算来匹配和验证. 以下将按照流程论述.

(1) 数据集构建与语义扩展. 一致性判定的第 1 步是构建和处理涵盖广泛合同类型和法律条文的数据集, 其中涉及以文本对形式配对. 模型使用 Prompt 技术来生成合同条款的不同语义变体对文本语义扩展, 增强模型的语义理解能力, 且要对文本标准化处理, 将其转化为可供模型理解和处理的形式, 减少噪音对匹配结果的影响.

(2) 语义表征的生成与优化. 语义表征生成预训练的 BERT 模型, 模型将自然语言表达转化为向量形式, 生成包含了语义信息上下文嵌入. 法律文本具有高度专业化的语言特点, 采用针对法律文本训练的语言模型 LegalBERT 提升性能. 生成表征后, 模型会通过降维技术优化, 将高维语义向量转化为低维空间, 以便计算和匹配. 此外, 对语义向量的归一化处理也确保了不同合同条款和法律条文之间的向量可比性, 提高相似度计算的准确性.

(3) 相似度计算与匹配策略. 模型使用相似度计算来评估条款与法律条文之间的语义一致性. 余弦值越接近 1, 表示两个向量越相似. 模型还引入了注意力机制, 动态调整不同词语在匹配过程中的权重, 并结合了基于规则和数据驱动的方法进一步提高匹配精度. 对于具有固定格式的条款, 模型使用预定义的匹配规则快速匹配. 基于混合策略, 模型能够在匹配过程中保持高效性和准确性. 计算完成之后, 模型将生成一个相似度矩阵, 并降序排序, 识别出与合同条款最为匹配的法律条文.

(4) 多任务学习与深层语义理解. 除了匹配任务外, 模型还同时执行其他相关任务, 如法律条文的分类、合约条款的风险预测等. 在多任务学习框架中, 模型共享一部分底层表示, 在高层结构上进行特定任务学习、多任务学习能帮助模型更好地捕捉复杂关系, 并在匹配过程中利用辅助任务提供的额外信息优化主任务表现.

3.3.2 增强过渡系统的观察等价性

为了验证符合一致性判定的过渡系统模型在属性验证过程中始终遵守有效性审查原则, 需要探讨增强模型与初始模型在行为上等价的条件, 并对等价性定理加以证明. 为此, 可以通过增加不可观察的β过渡来使观察等价[26]. 下面用S_I表示智能合约初始系统的状态集, S_E表示其增强衍生系统的状态集. 将影响账本状态的过渡视为可观测过渡集A, 而不可观测的过渡视为其余过渡B, $R=\{(q,r)\in S_I\times S_E\}$可被认为是一个弱的双向模拟. 依据此定义, 智能合约系统中那些代表了 Solidity 命名函数或回退执行语义的过渡集都是可观察的, 智能合约的状态变化都是透明的. 另一方面, 增强系统可以将每个 Solidity 函数的执行细节拆解为多条路径或多步执行来模拟实现. 假设每一个路径的最终过渡都是α过渡, 而其余是β过渡, 则对于S_I和S_E的状态, 每个$\alpha\in A$在账本状态上的影响相等, 即如果某个操作 (如 Solidity 函数的执行) 影响了账本状态, 那么不论是在初始系统还是增强系统, 这个影响都是一致的. 因此如果初始状态α满足$\sigma_I=\sigma_E$, 那么结果状态有$\sigma'_I=\sigma'_E$.

在I和E上的弱模拟是关系$R\subseteq S_I\times S_E$, 因此有如下性质和定理.

性质 1. 对于所有的$(q,r)\in R$和每一个$\alpha\in A$, 使得$q\xrightarrow{\alpha}q'$, 则存在r'使得$r\xrightarrow{\beta^*\alpha\beta^*}r'$, 其中$(q',r')\in R$.

对于状态S_I中某个状态的每一个可观察过渡α, 应当证明, 在S_E中有一个包含α和其他不可观测过渡的路径在所有等价状态下都存在, 并且结果状态等价.

性质 2. 对于所有的$(q,r)\in R$和$\alpha\in A$, 使得$r\xrightarrow{\alpha}r'$, 则存在q'使得$q\xrightarrow{\alpha}q'$, 其中$(q',r')\in R$.

对于状态S_E中一个状态的每个可观测量输出过渡α, 应当证明, 在S_I中有一个可观测量的输出过渡在所有等价状态下都存在, 并且结果状态等价.

性质 3. 对于所有的$(q,r)\in R$和$\beta\in B$, 应有$r\xrightarrow{\beta}r'$, $(q,r')\in R$.

对于每一个不可观察过渡, 应当证明, 初始状态通过其产生了结果状态, 该结果状态应当和所有与初始状态等价的状态等价.

定理 1. 对于每一个初始智能合约I及其相应的增强智能合约E, 有$I\sim E$.

下面将对定理 1 进行证明, 即证明对于某些符合特定标准的(q,r)对, 3 个条件都成立.

证明: 从算法来看, 对于S_I中的每个状态q, 在S_E中有且仅有一个对应状态$c(q)$, 在该状态下可以调用与q完全相同的函数. 函数的执行语义表明, 在A中α过渡可能被撤销或正常执行, 因此使用α^{rev}和α^{fin}分别表示这两种情况. 对于每个q中的α过渡, 存在一组从$c(q)$出发的路径P_α, 其中α和P_α表示相同的执行语义, 只是路径中包含了在α中的每个 Solidity 代码语句的不同过渡 (路径中的分支由 if 和 while 构造产生). 每个P_α可以表示为正则表达式$\beta^{\text{call}}\beta^*\alpha$, 其中$\beta^{\text{call}}$代表函数调用这个操作, 每个$\beta$过渡都可以是任意的代码语句, α是α^{rev}和α^{fin}.

图 4 展示了初始系统和增强系统中状态的抽象表示. 其中包含初始系统状态$q\in S_I$ (有两个过渡α^{rev}和α^{fin}), 以及对应的增强系统状态$r=c(q)\in S_E$ (有两个路径P_α^{rev}和P_α^{fin}). 为了证明定理 1 成立, 则需证明由虚线表示的关系R, 即$(q,r_1),(q,r_2),(q,r_3),(q',r')\in R$, 对于每个这样的$\alpha\in A$成立. 换句话说, 如果$r$对应于$q$, 那么它与$q$等价, 并且路径中可到达的所有其他$r_i$也与$q$等价, 输出状态$r'$则与$q'$等价. 如果证明了一个$q,r,\alpha$元组满足要求, 则可适用于所有此类情况.

首先证明$(q,r)\in R$和$(q',r')\in R$. 对于每个$\alpha^{\text{fin}}\in A$, 使得$q\xrightarrow{\alpha^{\text{fin}}}q'$, 存在$P_\alpha^{\text{fin}}$, 使得$r\xrightarrow{P_\alpha^{\text{fin}}}r'$, 且$P_\alpha^{\text{fin}}=\beta^*\alpha^{\text{fin}}$, 其中$\beta\in B$,

$\alpha^{\text{fin}} \in A$, 此外 q' 和 r' 是像 q 和 r 一样的对应状态, 因此如果 $(q,r) \in R$ 得证, $(q',r') \in R$ 也得证; 同样对于每个 $\alpha^{\text{rev}} \in A$, 也存在 P_{α}^{rev}, 使得 $r \xrightarrow{P_{\alpha}^{\text{rev}}} r$, 其中 $\beta \in B$, $\alpha^{\text{rev}} \in A$, 且最终状态 q 和 r 等价, 因此性质 1 对于 (q,r) 成立. 从 r 开始没有 A 的过渡, 因此性质 2 不适用, 不必证明. 性质 3 需证明 $(q,r1) \in R$, 因为 $r1$ 是从 r 出发通过 B 过渡可以到达的唯一状态, 该证明将在后续给出, 在此认为性质 3 已成立. 由于 3 个性质都成立, $(q,r) \in R$ 得证, $(q',r') \in R$ 也得证.

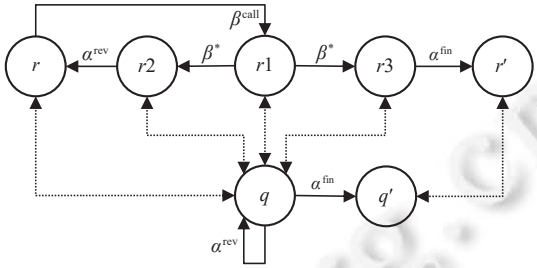

图 4　初始系统和增强系统中状态的抽象表示

接着证明 $(q,r1) \in R$. 对于每个 $\alpha^{\text{fin}} \in A$, 使得 $q \xrightarrow{\alpha^{\text{fin}}} q'$, 存在 P_{α}^{fin}, 使得 $r1 \xrightarrow{P_{\alpha}^{\text{fin}}} r'$, 且 $P_{\alpha}^{\text{fin}} = \beta^* \alpha^{\text{fin}}$, 其中 $\beta \in B$, $\alpha^{\text{fin}} \in A$, 此外, 已证明 $(q',r') \in R$; 同样对于每个 $\alpha^{\text{rev}} \in A$, 也存在 P_{α}^{rev}, 使得 $r1 \xrightarrow{P_{\alpha}^{\text{rev}}} r$, 其中 $\beta \in B$, $\alpha^{\text{rev}} \in A$, 且最终状态 q 和 r 等价, 因此性质 1 对于 $(q,r1)$ 成立. 从 $r1$ 出发没有 A 过渡, 因此性质 2 不适用不必证明. 性质 3 需证明 $(q,r2) \in R$ 和 $(q,r3) \in R$, 因为 $r2$ 和 $r3$ 是从 $r1$ 出发通过 B 过渡可以到达的唯二状态, 该证明将在后续给出, 在此认为性质 3 已成立. 由于 3 个性质都成立, $(q,r1) \in R$ 得证.

最后证明 $(q,r2) \in R$ 和 $(q,r3) \in R$. 对于每个 $\alpha^{\text{fin}} \in A$, 使得 $q \xrightarrow{\alpha^{\text{fin}}} q'$, 存在 P_{α}^{fin}, 使得 $r3 \xrightarrow{P_{\alpha}^{\text{fin}}} r'$, 且 $P_{\alpha}^{\text{fin}} = \beta^* \alpha^{\text{fin}}$, 其中 $\beta \in B$, $\alpha^{\text{fin}} \in A$, 此外, 已证明 $(q',r') \in R$; 同样对于每个 $\alpha^{\text{rev}} \in A$, 也存在 P_{α}^{rev}, 使得 $r2 \xrightarrow{P_{\alpha}^{\text{rev}}} r$, 其中 $\beta \in B$, $\alpha^{\text{rev}} \in A$, 且最终状态 q 和 r 等价, 因此性质 1 对于 $(q,r2)$ 和 $(q,r3)$ 成立. 对于每个 $\alpha^{\text{fin}} \in A$ 和 $\alpha^{\text{rev}} \in A$, 使得 $r3 \xrightarrow{\alpha^{\text{fin}}} r'$ 和 $r2 \xrightarrow{\alpha^{\text{rev}}} r$, 存在 $\alpha^{\text{fin}} \in A$ (或 $\alpha^{\text{rev}} \in A$), 使得 $q \xrightarrow{\alpha^{\text{fin}}} q'$ (或 $q \xrightarrow{\alpha^{\text{rev}}} q$), 此外, 已证明 $(q',r') \in R$ (或 $(q,r) \in R$), 因此性质 2 对于 $(q,r2)$ 和 $(q,r3)$ 成立. 从 $r2$ 或 $r3$ 出发没有 B 过渡, 因此性质 3 不适用不必证明. 由于 3 个性质都成立, $(q,r2) \in R$ 和 $(q,r3) \in R$ 得证. 证毕.

通过比较初始和增强智能合约, 模型建立了观察等价性的属性和定理, 确保增强系统能够准确地捕捉了原始系统的行为, 而不引入或遗漏任何功能. 转化前后, 过渡系统状态始终保持了有效性. 定理 1 进一步表明, 如果两者行为上等价, 则可认为增强系统的属性验证结果也适用于初始系统.

3.4　安全性准则

目前主要存在 3 种主要方法以确保智能合约的安全: 安全编程实践和模式 (如 Checks-Effects-Interactions 模式等), 自动化漏洞检测工具 (如 Oyente 等), 以及正确性形式验证.

遵循安全编程实践和使用常见模式生成智能合约的确可以减少漏洞的发生, 但该方案依赖于编程人员执行, 过程中失误难免, 且仅适用于典型漏洞, 对于非典型或尚未分类的漏洞无效, 此外不能提供形式化安全保证, 实用性十分有限. 自动化漏洞检测工具通常需要安全专家以低级别 (通常是字节码) 指定安全属性和模式, 非专业人员难以使用, 且检测工具几乎不关注合约要求, 只检测通用属性安全性, 因此也难以发现非典型漏洞. 此外, 工具不精确经常误报. 相反, 基于形式操作语义的验证工具提供了强大保证. 该工具可以检测各种违反安全属性规范的典型和非典型漏洞. 目前现有研究在智能合约的形式化验证方面已经取得了一些进展, 例如, 王小兵等人[27]采用 MSVL 和 PPTL 模型检测技术验证合约的逻辑一致性, 韩宁等人[28]使用 Yul 语言的小步语义定义奠定了验证基础, 此外, 王璞巍等人[29]还在 Hyperledger Fabric 上实现了状态转换模型, 以及付利青等人[30]通过模型检测工具验证了以太币投票协议的逻辑正确性. 然而, 这些研究主要集中于技术验证和业务逻辑, 并未解决智能合约的合法性问题. 本

文提出的 LTSC-TAV 模型则填补了这一空白, 实现技术与法律的深度结合. LTSC-TAV 也属于形式验证工具, 通过关键决策的人为干预很好地解决了流程自动化的困难, 帮助开发者在设计早期消除错误, 生成完善的智能合约, 实现安全性验证的效果. 下面将具体介绍安全性准则在 LTSC-TAV 模型中的体现.

3.4.1 预防重入漏洞

重入攻击 (reentrancy attack) 是智能合约领域的一种常见攻击方式, 攻击者可以在外部调用合约函数时, 递归调用相同或其他函数, 从而导致合约状态不一致或资金损失. LTSC-TAV 模型在输入初始过渡系统中允许预防重入漏洞的设计, 在过渡开始和执行过渡操作之间, 将当前状态更改为一个临时状态, 限制调用合约任何函数的能力阻止重入. 虽然重入并非都有害, 但支持重入会大幅增加验证的复杂性, 难以高效验证众多属性, 不利于迭代开发. 重入也显著复杂化了合约行为, 导致漏洞出现. 因此禁止重入是利大于弊的.

即使没有预防重入漏洞的设计, 也可以选择自动生成预防重入的增强过渡系统. 下面将对增强过渡系统自动生成方法进行具体说明.

(1) 符合性转换. 其核心思想是使用功能等效的状态过渡模拟 Solidity 合约中回退函数和构造函数的行为. 首先替换合约中的回退函数操作, 为每个状态都添加一个不改变状态的过渡进行回退, 确保了模拟不同状态下回退函数被调用的情境. 接着添加一个新初始状态, 并创建到原始初始状态的过渡, 该操作即为初始操作. 符合性转换令合约初始化过程纳入模型中, 保持了其原有逻辑结构. 具体见算法 1.

算法 1. 一致性转换 $(D,S,S_F,s_0,a_0,a_F,V,T)$.

输入: 模型 $(D,S,S_F,s_0,a_0,a_F,V,T)$;
输出: 模型 (D,S,S_F,s_0,V,T).

1. **for** state $s \in S$ **do**
2. **add** transition from s to s with action a_F
3. **end for**
4. **add** state s_I
5. **add** transition from s_I to s_0 with action a_I
6. **change** initial state $s_0 := s_I$

(2) 增强转换. 其目标是将包含复杂语句 (如复合、选择、循环等) 的模型简化为仅包含变量声明、表达式和返回语句的模型, 让模型更加简洁和易于验证. 该过渡由语句增强和模型增强两个算法构成, 两个增强算法都基于附录 A.3 中提供的 Solidity 语句的小步操作语义.

算法 2 语句增强的目的是将单个 Solidity 语句翻译为多个状态和过渡. 算法以一个 Solidity 语句、一个起始状态、目标状态和返回状态作为输入, 并创建一组状态和过渡, 这些过渡仅使用变量声明、表达式和返回语句作为操作来实现输入语句, 然后根据语句的类型 (如变量声明、表达式、返回语句、复合语句、选择语句或循环语句) 创建相应的状态和过渡.

算法 2. 语句增强 (a,s_o,s_d,s_r).

输入: 语句 a, 原状态 s_o, 目标状态 s_d, 返回状态 s_r.

1. **if** a is variable declaration statement $\vee$ a is event statement $\vee$ a is expression statement **then**
2. **add** transition from s_o to s_d with action a
3. **else if** a is return statement **then**
4. **add** transition from s_o to s_r with action a
5. **else if** a is compound statement $\{a_1;a_2;\ldots;a_N\}$ **then**

6. **for** $i = 1, 2, \ldots, N-1$ **do**
7. **add** state s_i
8. **end for**
9. *AugmentStatement* (a_1, s_o, s_1, s_r)
10. **for** $i = 2, 3, \ldots, N-1$ **do**
11. *AugmentStatement* (a_i, s_{i-1}, s_i, s_r)
12. **end for**
13. *AugmentStatement* (a_N, s_{N-1}, s_d, s_r)
14. **else if** a is selection statement if (c) a_T else a_F **then**
15. **add** state s_T
16. **add** transition from s_o to s_T with guard c
17. *AugmentStatement* (a_T, s_T, s_d, s_r)
18. **add** state s_F
19. **add** transition from s_o to s_F with guard $!(c)$
20. *AugmentStatement* (a_F, s_F, s_d, s_r)
21. **else if** a is selection statement if (c) a_T **then**
22. **add** state s_T
23. **add** transition from s_o to s_T with guard c
24. *AugmentStatement* (a_T, s_T, s_d, s_r)
25. **add** transition from s_o to s_d with guard $!(c)$
26. **else if** a is loop statement for $(a_I; c; a_A)$ a_B **then**
27. **add** states s_I, s_C, s_B
28. *AugmentStatement* (a_I, s_o, s_I, s_r)
29. **add** transition from s_I to s_d. with guard $!(c)$
30. **add** transition from s_I to s_C. with guard c
31. *AugmentStatement* (a_B, s_C, s_B, s_r)
32. *AugmentStatement* (a_A, s_B, s_I, s_r)
33. **else if** a is loop statement while (c) a_B **then**
34. **add** state s_L
35. **add** transition from s_o to s_d. with guard $!(c)$
36. **add** transition from s_o to s_L with guard c
37. *AugmentStatement* (a_B, s_L, s_o, s_r)
38. **end if**

算法 3 模型增强的目的是通过迭代并使用算法 2, 将每个过渡替换为简化的状态和过渡, 减少模型复杂性. 算法接受一个包含任意支持语句作为操作的模型, 并将其过渡为只包含变量声明、表达式和返回语句的模型. 此外, 算法还为执行过程中可能的异常情况 (如函数调用或转移操作失败) 添加相应的状态和过渡. 该操作确保模型不仅在正常情况下功能等效, 也能在异常时正确模拟 Solidity 合约的行为.

算法 3. 模型增强 (D, S, S_F, s_0, V, T).

输入: 模型 (D, S, S_F, s_0, V, T);
输出: 模型 (D, S, S_F, s_0, V, T).

1. **for** transition $t \in T$ **do**
2. **remove** transition t
3. **add** state s_{grd}
4. **add** transition from t^{from} to s_{grd} with guard g_t
5. **if** action a_t cannot raise exception **then**
6. *AugmentStatement* $(a_t, s_{grd}, t^{to}, t^{to})$
7. **else**
8. **add** transition from s_{grd} to t^{from} with guard "revert"
9. **add** state s_{rvrt}
10. **add** transition from s_{grd} to s_{rvrt} with guard " !revert "
11. *AugmentStatement* $(a_t, s_{rvrt}, t^{to}, t^{to})$
12. **end if**
13. **end for**

3.4.2 属性验证过程

为了检测不属于常见类型的漏洞, 开发者必须指定合约的预期行为. LTSC-TAV 模型帮助开发者能够以活性、死锁自由和安全属性的形式指定预期行为, 这些属性已包含了重要的安全关切和漏洞. 属性验证方法的关键优势在于允许开发者指定与高级模型相关的期望属性而非字节. LTSC-TAV 模型可以自动验证合约的行为是否满足验证属性, 如果不满足, LTSC-TAV 模型会解释导致属性违规的执行序列, 帮助人为识别和纠正错误行为的设计. 由于合约代码在部署后不能更新, 该验证方法只需在部署前生成一次代码即可满足安全性准则要求, 非常适合智能合约. 属性验证的过程如下.

(1) 活性、死锁自由和安全属性

LTSC-TAV 模型使用了计算树逻辑规范属性 (CTL). CTL 具有规范的公式, 用于指定由过渡系统生成执行树属性[31]. CTL 公式由表示过渡和过渡系统语句的原子谓词构成, 并使用多种操作符, 如 EX、AX、EF、AF、EG、AG(单目) 和 E[·U·]、A[·U·]、E[·W·]、A[·W·](双目). 每个操作符由树分支上的量词和时间模态组成, 两者共同定义了子公式在执行过程中必须成立的时机. 各字母的直观意义如下: 分支量词为 A (表示"所有") 和 E (表示"存在"); 时间模态为 X (表示"下一次")、F (表示"将来的某个时间")、G (表示"全局地")、U (表示"直到") 和 W (表示"弱直到"). 如果过渡系统初始状态中属性成立, 则认为该属性得到满足.

表 3 展示了所有类型的自然语言模板及其对应的 CTL 公式. 其中, a、b、c 为占位符, 表示过渡和语句集, 即 $\langle Transitions \cup Statements \rangle$.

表 3 不同类型属性的模板及对应公式

类型	模板	CTL公式
安全	a不能发生在 b 之后	AG(b → AG(¬a))
安全	a只能发生在 b 之后	A[¬a W b]
安全	如果 a 发生, b 只能发生在 c 之后	AG(a → AX A[¬b W c])
安全	a不能发生	AG(¬a)
安全	a不能在 b 之前发生	A[¬a | AG(¬b) W b]
活性	b发生后, a 终将发生	AG(b → AF(a))
活性	a终将发生	AF(a)

LTSC-TAV 模型会自动验证死锁自由, 还可在开始时输入额外需要验证的安全性和活性属性. 为方便属性指定, LTSC-TAV 模型提供了一组预定义的类自然语言模板以对应 CTL 中的属性, 以下是常用预定义模板.

$\langle Transitions \cup Statements \rangle$ cannot happen after $\langle Transitions \cup Statements \rangle$.

该模板属于安全属性类型. 其中 *Transitions* 是模型过渡的一个子集 (即 $Transitions \subseteq T$). *Statements* 中的内容来自特定过渡操作的内部语句 (即 $Statements \subseteq T \times S$).

If ⟨*Transitions* ∪ *Statements*⟩ happens, ⟨*Transitions* ∪ *Statements*⟩ can happen only after ⟨*Transitions* ∪ *Statements*⟩ happens.

该模板也属于安全属性类型. 适用于验证货币提取功能中的典型漏洞 (例如 transfer), 该漏洞允许攻击者在更新余额之前再次提取货币. 实际使用时, *Transition.Statement* 形式指代特定过渡的一个语句. 如果同一过渡中有多个相同语句, 那么所有语句都会对同一属性检查. 为验证带有语句的属性, 需要将输入模型转化为增强模型.

⟨*Transitions* ∪ *Statements*⟩ will eventually happen after ⟨*Transitions* ∪ *Statements*⟩.

该模板属于活性属性类型, 可检查拒绝服务漏洞 (附录 B.2).

(2) 数据抽象

LTSC-TAV 模型使用属性验证的方法检查合约行为是否存在明显的逻辑漏洞, 以及是否满足开发者要求的特殊属性. 在进行属性验证时, 必须考虑数据和时间的影响. 智能合约使用环境输入作为控制数据, 例如守卫等. 输入数据无限时, 合约状态也会存在无限多可能. 在此情况下, 无法研究每一个状态, 并且效率非常低[32], 因此必须对适当的数据和时间抽象处理.

数据抽象可以忽略依赖环境输入的变量, 如由环境输入更新的变量等. 不评估涉及此类变量的过渡守卫会导致合约行为的过度近似. 因此, 应该假设它们的值都是可能的, 并且状态空间探索应该包括具有和不具有每个守卫过渡的执行轨迹. 本质上, 我们分析的是一个更抽象的合约模型, 其可达状态集和轨迹集是实际合约状态集 (个别情况是轨迹集) 的超集. 用图 5 中的函数为例, 函数执行的过度近似应当包括同时访问两行 (1) 和 (2) 的轨迹, 即使这两个条件不能同时被一个 x 值满足. 需要特别注意的是, 不依赖环境输入的变量不需要抽象, 如已知范围的迭代计数器等. 在模型中, 这些变量是根据合约语句计算更新的.

```
void fn(int x){
    if(x<0){
        ...                (1)
    }
    if(x>0){
        ...                (2)
    }
}
```

图 5 过度近似的代码示例

对时间变量也需要抽象处理, 不过采用的方式略有不同. 虽然要知道哪些过渡会随着时间的增加而失效, 但这个时间可以是任意大小, 所以不必表示每个状态中所花费的时间. 因此, 对于模型中的一个来自状态 s_x 的时间守卫过渡, 以下之一适用.

如果是 $t \leqslant t_{\max}$ 类型的守卫, 检查时间变量是否不高于阈值, 添加一个带有操作 $t \leqslant t_{\max}+1$ 的循环过渡到 s_x, 令守卫失效. 如果 s_x 中没有其他过渡可用, 在执行守卫失效的循环时可能会发现死锁.

如果守卫是 $t > t_{\min}$ 类型的, 检查时间变量是否已超过了阈值, 添加一个操作 $t > t_{\min}+1$ 到守卫的过渡上. 该操作将时间设置为达到下一个状态的最早点, 可用于有界活性属性的检查.

这种过度近似有多重含义.

安全属性: 在抽象模型中满足的安全属性在实际系统中也得到了保证. 安全属性用于检查错误状态的不可达性, 如果这些状态在抽象模型中不可达, 作为抽象模型状态的一个子集, 它们在具体模型中也将不可达.

活性属性: 在抽象模型中被违反的活性属性在实际系统中也被违反. 活性属性用于检查状态的可达性. 如果这些状态在抽象模型中不可达 (即违反活性), 它们在具体模型中也将不可达. 此属性类型有利于检查拒绝服务漏洞

(附录 B.2).

死锁自由: 没有启用外出过渡的状态被识别为死锁状态. 如果在抽象模型中没有死锁状态可达, 那么在实际系统中也不会可达.

(3) BIP 映射和属性验证

行为-交互-优先级 (behavior-interaction-priority, BIP) 组件框架具有广泛的应用价值, 已被用于构建许多设计系统. 在 BIP 中, 系统通过叠加行为、交互和优先级共 3 层建模. 行为层由一组过渡系统组件构成. 行为层使用一个端口标记每个组件的过渡, 该端口指定了过渡的唯一名称. 端口构成了组件的接口, 用于与其他组件交互. 此外, 每个过渡还可以关联一组守卫条件和一组操作. 守卫条件是变量上的谓词, 必须为真才能允许执行相关过渡. 操作是执行相关过渡时所触发的计算. 组件的交互通过交互层和优先级层来描述. LTSC-TAV 模型中没有使用这两层, 因此省略详细解释.

为了检查设计中系统行为的正确性, 形式化属性验证至关重要. 尽管目前仍广泛采用模拟和测试等方法, 通过选择合适的测试输入来充分覆盖程序控制流, 但形式化属性验证能够覆盖所有可能输入的执行路径. 因此, 形式化属性验证为确认系统模型是否满足指定属性提供了严格的检测方法.

在 LTSC-TAV 模型中, BIP 同样将合约行为建模为过渡系统. 因此, 由初始或增强过渡系统到 BIP 模型的转化只是过渡、状态、守卫和操作之间的简单映射, 这种一对一的映射不必证明. 模型的翻译算法对输入语法进行导向解析, 并收集追加到模板属性列表的值. 具体值为: 变量 $v \in V$, 其中 $type(v)$ 是 v 的数据类型, $name(v)$ 是变量名 (即标识符); 状态 $s \in S$; 过渡 $t \in T$, 其中 t^{name} 是过渡和相应端口名, t^{from} 和 t^{to} 是外出和进入状态, g_t 和 a_t 是实现相关操作和守卫的函数调用.

图 6 展示了 BIP 代码的生成模板. 其中, 加粗字体表示生成的输出, 斜体字表示输入替换的元素.

	atomtypeContract()
$\forall v \in V$:	**data** *type*(*v*) *name*(*v*)
$\forall v \in T$:	**export port synPort** t^{name} ()
	places $s_0, \ldots, s_{\lvert S \rvert - 1}$
	initial to s_0
$\forall t \in T$:	**on** t^{name} **from** t^{from} to t^{to}
	provided (g_t) **do** {a_t}
	end

图 6 BIP 代码生成模板

LTSC-TAV 模型使用 BIP 提供的状态空间探索分析方法默认验证系统是否无死锁. 死锁是一项基本的正确性属性. 对于安全性和活性属性的验证, 模型使用 BIP-to-NuSMV 工具将 BIP 模型转化为 NuSMV, 即符合 nuXmv 符号模型检查器输入语言的形式. 验证属性需要以时序逻辑公式的形式输入, 或者使用预定义的自然语言模板, 以生成 CTL 规范. 属性由 nuXmv 工具进行检查. 基于双向模拟的 BIP-to-NuSMV 转化的正确性已由 Noureddine 等人证明. 如果属性被违反, 将得到一个反例转换序列, 该序列示例化地说明了违反情况, 帮助用户定位输入模型的错误并识别其原因.

(4) 代码生成

属性验证通过后, LTSC-TAV 模型将基于 FSolidM 代码生成器的扩展自动生成 Solidity 智能合约代码, 以规范的模型化初始过渡系统作为代码生成过程的输入, 遵循定义的过渡系统的操作语义来生成 Solidity 代码. 详细的 FSolidM 代码生成器介绍参见文献. 与 FSolidM 相比, LTSC-TAV 模型生成器主要区别有: 每个过渡的开始时, 如果有非空操作, 则状态变量 state 设置为 InTransition; 根据初始操作 a_0 生成构造函数; 根据回退操作 a_F 生成回退

函数; 为了保持模型与生成代码的功能等价, 生成过程中不使用 FSolidM 代码生成器插件.

VeriSolid 代码生成器的输入是一个定义为过渡系统 (见定义 3) 的智能合约 $(D,S,S_F,s_0,a_0,a_F,V,T)$. 输入时需要指定合约的名称 *name*. 对于每个过渡 $t\in T$, 还需指定 t^{playable}, 若实现过渡 t 的函数与付款有关, 则此值为真, 否则为假. 对于每个合约变量或输入变量 (即函数参数) $v\in \mathbf{I}\times\mathbf{T}$, 定义 $name(v)\in\mathbf{I}$ 和 $type(\mathbf{v})\in\mathbf{T}$ 分别表示变量的名称和类型. 使用粗体表示生成的代码, 斜体表示需要用输入替换或稍后指定的元素, 合约结构可表示为:

Contract:: = **contract** *name*{
StatesDefinition
VariablesDefinition
Constructor
Fallback
Transition(t_1)
…
Transition($t_{|T|}$)
}

其中, $\{t_1,\ldots,t_T\}$ 是过渡 T 的集合. 状态定义为:

StatesDefinition := **enum States**{
InTransition, s_0, …, $s_{|S|-1}$
}
States private state;

其中, $\{s_0,\ldots,s_{|S|-1}\}$ 是状态 S 的集合. 变量定义为:

VariablesDefinition ::= *D*
type(v_1) *name*(v_1);
…
type($v_{|V|}$) *name*($v_{|V|}$);
uint private
creation Tim = **now**;

其中, D 是自定义事件和类型定义的集合, $\{v_1,\ldots,v_{|V|}\}$ 是合约变量 V 的集合. 构造函数为:

Constructor ::= **constructor**()**public**{
Action(a_0, **States**.s_0)
state = **States**.s_0;
}

其中, s_0 和 a_0 分别为初始状态和初始操作. 回退函数为:

Fallback ::= **function**()**payable public**{
State memory currentState = **state**;
Action(a_F, **currentState**)
state = **currentState**;
}

其中, a_F 是回退操作. 过渡函数为:

$$
\begin{aligned}
Transition(t) ::= \textbf{function } & t^{\text{name}}(type(i_1)name(i_1),\ldots,type(i_{|\text{input}|})name(i_{|\text{input}|}))\\
& \textbf{public } Payable(t)Returns(t)\{\\
& \textbf{require}\left(\textbf{state} == \textbf{States}.t^{\text{from}}\right);\\
& \textbf{require}(g_t);\\
& Action(a_t,\textbf{States}.t^{\text{to}})\\
& \textbf{state} = \textbf{States}.t^{\text{to}};\\
&\}
\end{aligned}
$$

其中, t^{name} 是过渡 t 的名称, $\{i_1,\ldots,i_{|\text{input}|}\}$ 是参数变量集 (即函数参数), g_t 和 a_t 分别是条件和操作. t^{from} 和 t^{to} 分别是起始状态和目标状态.

若 t^{playable} 值为真, 则 $Payable(t)$ ::= **payable**; 否则 $Payable(t)$ 为空. 若返回类型 $t^{\text{output}} = \varnothing$, 则 $Returns(t)$ 为空; 否则为 $Returns(t)$::= **returns**(t^{output}). 若 $a = \varnothing$, 即空操作语句, 则 $Action(a,s)$ 为空. 否则:

$$Action(a,s) ::= \textbf{state} = \textbf{States}.\textbf{InTransition};$$

$$SafeAction(a,s)$$

其中, $SafeAction(a,s)$ 仅表示了语句 a, 但将语句 a 中的 **return** *expression*; 或 {**state** = s; **return** *expression*;} 语句替换为: **return**; 或 {**state** = s; **return**;}. 这适用于所有在 a 中的嵌套语句, 包括选择语句和循环语句中的操作.

4 实验分析

为了验证原则和模型的有效性、可靠性, 本节将结合特定案例展示模型在实际场景中的应用表现. 此外, 本节还展示其他数据进一步评估模型的多方面能力, 以及在处理各种合同形式的适应能力.

4.1 实验数据买卖合同案例分析

为了直观说明如何将智能合约表示为过渡系统并验证和生成, 本文选择以简化版的买卖合同为实验案例, 展示模型在转化验证合同时的灵活性和可靠性.

4.1.1 案例说明及其初始过渡系统

简化的买卖合同案例中存在买方和卖方两名当事人, 开始执行后, 首先乙方等待甲方付款, 然后乙方进行发货甲方进行货物验收, 验收完成后将展示验收结果. 买卖合同有 4 个主要状态.

(1) 等待付款 (AwaitingPayment, AP): 合同开始状态, 甲方需在规定时间内付款.

(2) 货物验收 (GoodsInspection, GI): 乙方已完成运货, 甲方需进行检查.

(3) 终止 (Termination, T): 合同终止, 双方解除协议.

(4) 结束 (Finished, F): 合同完成.

从合同案例中可发现, 智能合约包含多个状态, 合约提供的函数允许当事人调用操作并改变合约状态. 因此, 可以将智能合约自然地表示为一个包括一组状态和状态间过渡的过渡系统, 守卫条件满足的前提下可以调用过渡迫使合约执行过渡操作, 为推理其行为提供了足够的抽象级别.

图 7 所示为买卖合同案例过渡系统 (省略了过渡采取的具体操作). 合同主要过渡如下.

(1) 付款 (pay): 仅甲方可执行, 允许多次付款, 款项累计在存款处.

(2) 运输 (transport): 仅乙方可执行, 付款小于规定时间且累计金额不小于货款时, 允许运货, 并将状态过渡为 GI.

(3) 验收 (inspect): 仅甲方可执行, 在小于规定时间完成运输时, 允许验收, 完成后出示结果.

(4) 取消等待付款 (cancelAP): 在规定时间内累计金额小于货款时乙方可执行, 或双方协商一致后执行, 将状态过渡为 T.

(5) 取消货物验收 (cancelGI): 在未按时完成运输或验收结果不合格时甲方可执行, 或双方协商一致后执行, 将状态过渡为 T.

(6) 结束合同 (finish): 在验收结果合格时执行, 将状态过渡为 F.

(7) 退还 (return): 退还付款, 退回货物.

(8) 取款 (withdraw): 支付和货物等价的款项, 退还多余款项.

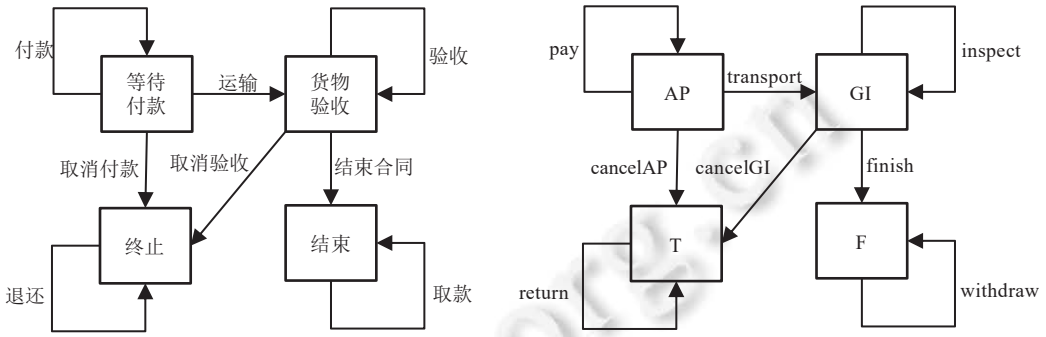

图 7 买卖合同案例的中文和英文版初始过渡系统

该初始过渡系统构建了买卖合同的核心执行流程, 从合同签订到终止或完成的状态变化, 涵盖了付款、运输、验收和退货等主要条款, 明确了状态转移的基本规则 (如付款完成后才可运输). 这种规则的嵌入提高了模型的条款匹配精度, 使得合同执行流畅且规范. 通过对买卖合同案例的初始过渡系统建模, 实验充分展示了合同条款和法律逻辑在形式化表达上的紧密结合. 合同中 4 个主要状态及其过渡关系的设计, 不仅保证了逻辑上的完整性, 也通过付款、运输、验收等明确的操作映射, 实现了自然语言条款与智能合约语法的高效连接. 这种连接的关键在于前置处理阶段对合同条款信息的准确提取及过渡关系的合理抽象. 例如, 取消付款与取消验收操作分别针对不同阶段的取消条件, 清晰地反映了买卖双方在合同履行过程中动态调整的灵活性.

此外, 实验中的状态划分设计充分体现了文法要求和非赋权原则. 例如, 终止状态通过直观地标注终止及条件, 避免了可能存在的无限循环或不确定过渡, 增强了合约执行的确定性和安全性. 相比传统法律合同的模糊性表达, 智能合约中的形式化表达不仅减少了解释争议, 还在执行层面大大提高了效率.

4.1.2 案例的增强过渡系统

基于买卖合同案例的增强后过渡系统如图 8 所示, 其中使用方括号是关于时间的基础守卫, 以区分过渡名称, 并使用 AS 和数字组合的形式表示扩展的中间状态. 增强过渡系统的设计通过将复杂操作进一步细化和标准化, 显著提高了属性验证的准确性和系统执行的可靠性. 以 withdraw 过渡为例, 过渡系统的增强过程会将复杂的操作分解为一系列更简单的步骤, 每执行一个操作或一个条件, 都会过渡到一个新的状态. 确保了在每一步操作中都能严格满足守卫条件. 这种细化的关键在于为复杂操作建立了中间状态, 既避免了因过度简化带来的潜在逻辑漏洞, 也确保了操作过程的透明性.

图 8 中的增强系统还体现了对安全性准则的全面落实. 例如, 通过在 cancelAP 和 cancelGI 操作中添加守卫条件, 系统能够动态检查操作的合法性, 从而有效防止误操作或恶意操作对合约的破坏. 这种设计不仅增强了合约的容错能力, 也提升了其在多方参与场景下的适应性. 此外, 如果遇到错误或不符合要求的条件, 守卫条件 revert 可以将操作回退, 确保合约执行的正确性和安全性.

增强系统的设计还反映了对非赋权原则的高度重视. 通过在每个过渡中添加对操作者身份和权限的严格校验, 系统确保了当事人的自主决策权. 例如, transport 操作明确限制为卖方执行, 避免了因权限混淆导致的潜在合规性问题.

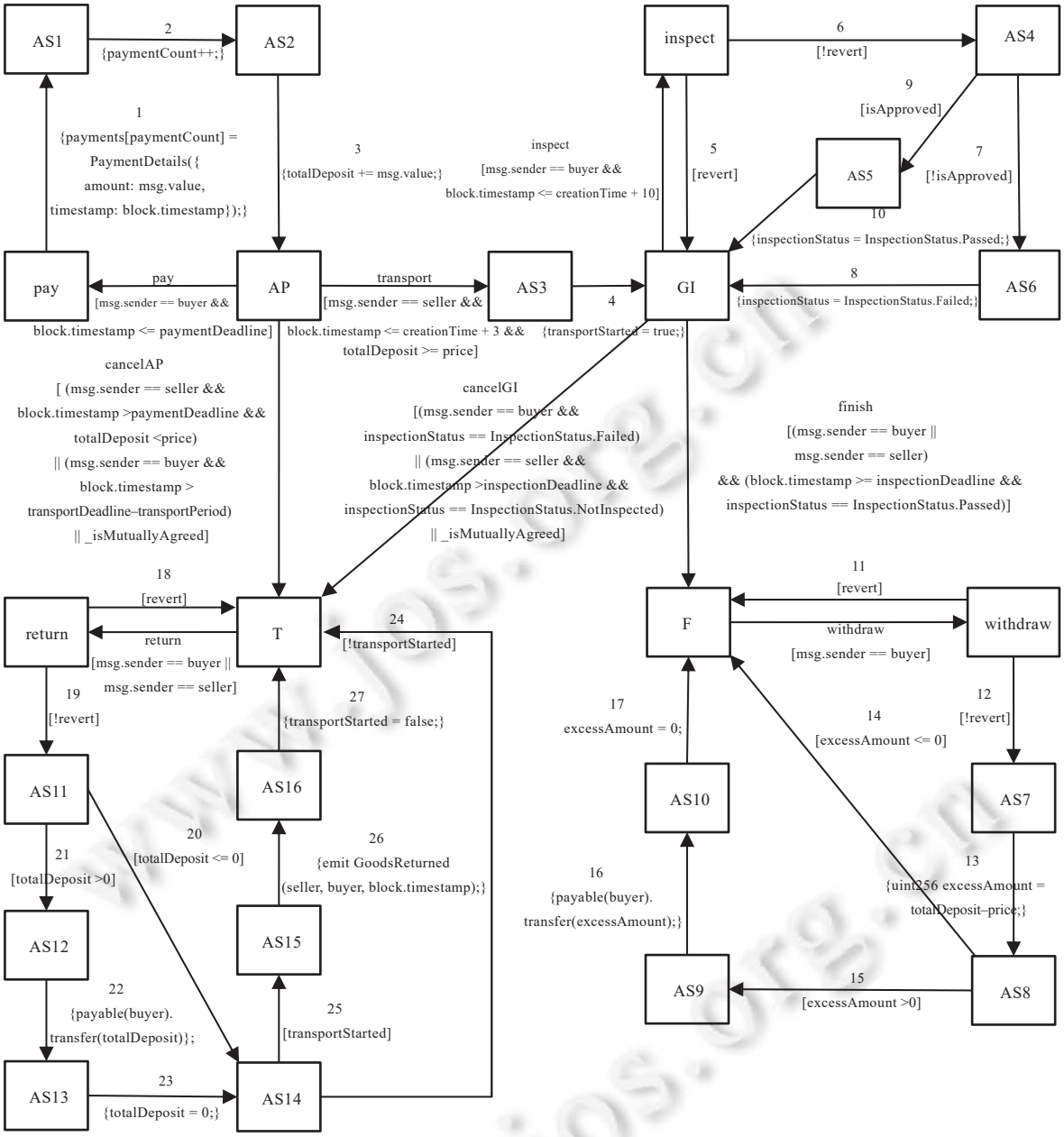

图 8 买卖合同案例的增强模型

4.1.3 属性验证

表 4 总结了买卖合同案例的属性和验证结果. 为了便于展示, 属性语句使用添加的增强过渡编号来代替语句. 首先展示的是买卖合同的初始和增强模型结果. 在初始模型上, 本实验检查了 4 个安全属性, 在允许更细粒度分析的增强模型上, 本实验检查了两个额外的安全属性. 所有属性都被验证为成立. 这些模型被发现是没有死锁的.

实验中对属性验证的结果分析表明, 模型能够全面覆盖典型的安全性和活性属性, 同时通过反例验证进一步加强了对潜在风险的识别能力. 例如, 在初始买卖合同中, 安全性属性如“pay 不能发生在 transport 之后”的验证结果完全符合预期, 表明模型在处理标准合同逻辑时具有较高的可靠性.

表 4 买卖合同案例的属性分析和验证结果

案例研究	属性	类型	结果
初始买卖合同	**pay** cannot happen after **transport:** AG(**transport** → AG¬**pay**)	安全性	已验证
	cancelAP or **cancelGI** cannot happen after **finish**: AG(**finish** → AG¬(**cancelGI**∨**cancelAP**))	安全性	已验证
	withdraw can happen only after **finish**: A[¬**withdraw** W **finish**]	安全性	已验证
	finish can happen only after **transport**: A[¬**finish** W **transport**]	安全性	已验证
增强买卖合同	**16** cannot happen after **14**: AG(**14** → AG¬**16**)	安全性	已验证
	if **16** happens, **17** can happen only after **16**: AG(**16** → AX A[¬**17** W (**16**)])	安全性	已验证
DAO攻击	if **call** happens, **call** can happen only after **subtract**: AG(**call** → AX A[¬**call** W **subtract**])	安全性	已验证
以太坊之王1	**10** will eventually happen after **6**: AG(**6** → AF **10**)	活性	违反
以太坊之王2	**16** will eventually happen after **fallback**: AG(**fallback** → AF **16**)	活性	违反

针对增强系统的验证, 实验进一步展示了增强设计在提升复杂属性验证能力方面的优势. 例如, 属性“16 可以发生在 14 之后”的验证过程表明, 系统在引入更多中间状态后, 能够有效捕捉复杂逻辑关系, 并通过守卫条件确保所有过渡的合法性和一致性.

值得一提的是, DAO 攻击模拟了 DAO 合约的简化版本, 验证安全属性的结果中排除了两种攻击的可能性. 设置以太坊国王王座案例的目的是检查拒绝服务漏洞, 充分展示模型在捕捉异常行为和边界问题上的能力. 本实验中创建了两个版本模型. 模型 1 实验检查了一个活性属性, 该属性指出在补偿计算 (过渡 6) 之后的某个时间点会发生加冕 (过渡 10). 结果显示该属性被以下反例所违反: 6→7→8. 第 2 个活性属性指出, 如果回退失败则加冕 (过渡 16) 会在某个时间点之后发生, 属性违反的反例如下: 12→13→14. 对于相同违规行为, 可能存在多个反例, 在此仅举例其中的一个. “以太坊之王”案例实验清晰地揭示了潜在的拒绝服务漏洞. 这种反例验证的意义在于帮助用户发现可能存在的问题, 从而为后续的合约优化提供明确的方向.

此外, 验证过程还展示了增强模型在处理时间约束和动态状态转移中的灵活性. 例如, 通过在每个状态中引入时间守卫 (如 block.timestamp), 增强系统能够更精确地模拟实际合同的时间依赖性, 避免了因超时导致的潜在漏洞.

4.1.4 Solidity 智能合约代码

增强过渡系统通过验证后自动生成的 Solidity 智能合约代码, 不仅完全符合实验目标, 还为智能合约的实际部署提供了可靠的技术支撑. 部分代码展示如图 9 所示. 生成的代码充分体现了四大原则.

(1) 文法要求. 合约中使用了如 buyer、seller 等自然语言中明确变量名称, 与 AP、T 等法律合同中常见表达方式, 清晰反映出合约功能和操作逻辑, 符合法律用语的标准, 方便理解和对应. 通过对状态变量和操作函数的严格定义, 确保了执行过程中的安全性. 例如, withdraw 操作通过对 totalDeposit 和 price 的校验, 不仅避免了过量支付或资金损失的风险, 也为多次支付的合法性提供了保障.

此外, 合约利用状态机、事件、函数等编程元素来定义和约束各方的行为, 构建了完整的法律框架, 涵盖了合同的执行、检查、终止等关键环节, 确保每一步操作都有明确的条件和后果.

(2) 非赋权原则. 合约进行了条件验证确保操作合法且符合合同约定, 也提供了多种取消操作, 允许特定条件改变状态.

(3) 有效性审查. 合约每个状态过渡都与对应的合同条款一致, 在涉及双方操作中, 合约要求达成一致意见才

能执行. 条件检查机制也防止无效操作的发生, 避免了无效合约的产生.

(4) 安全性准则. 本合约使用了条件验证与事件记录确保合约执行的合法性和可靠性. 此外合约也通过 payments 映射记录每次付款的详细信息, 确保资金流动的透明性和可追踪性. 此外, 生成的代码通过严格的权限验证和状态过渡限制, 有效防止了重入攻击等常见漏洞. 例如, 合约在每次状态改变时都会进入临时状态 States.InTransition, 确保当前操作未完成时无法调用其他函数, 从而实现了高效且安全的执行逻辑.

```
pragma solidity ^0.8.0;
contract PurchaseContract {
    enum States {InTransition, AP, GI, T, F} // 状态：过渡中，等待付款、验收货物、终止、结束
    States private state = States.AP;
    address payable public buyer; // 买方
    address payable public seller; // 卖方
    uint256 public price; // 货物价格
    uint256 public totalDeposit; // 总支付存款
    struct PaymentDetails {uint256 amount; uint256 timestamp;}
    mapping(uint256 => PaymentDetails) public payments; // 映射记录每次支付的详细信息
    uint256 public paymentCount;
    // ...

    constructor( address payable _buyer, address payable _seller, uint256 _price // ...
    ) {
        // 构造函数验证地址和价格有效性
    }

    // ...

    // 过渡withdraw（取款）
    function withdraw() public {
        require(state == States.F, "Invalid state");
        require(msg.sender == buyer, "Only buyer can withdraw");
        // 状态变化
        state = States.InTransition;
        // 操作
        uint256 excessAmount = totalDeposit - price;
        if (excessAmount > 0) {
            payable(buyer).transfer(excessAmount);
            excessAmount = 0; // 清零退还余额
        }
        // 状态变化
        state = States.F;
    }

    // ...

}
```

图 9 Solidity 智能合约部分代码展示

4.2 其他实验数据

R@K (*Recall* at *K*) 是信息检索和推荐系统中常用的评估指标, 用于衡量模型在返回的前 *K* 个结果中寻找相关项的能力. *R@K* 表示在给定查询的所有相关项中, 有多少比例出现在模型返回的前 *K* 个结果中. *R@K* 为在前 *K* 个结果中的相关项与所有相关项总数的比值, 取值范围在 0–1 之间, 值越高表示模型越能够在靠前的位置检索到相关结果. 在某些任务中, 尤其是涉及合同条款与法律条文匹配等高精度需求时, 较高的 *R@K* 值表明模型具备出色的文本表征能力和匹配效果.

前置处理中合同条款和法律条文的一致性判定方面, 将所提算法框架在 CLMD 上测试, 验证结果表明, 最优的文本表征模型为 Transformer, *R*@1 值为 90.27%, *R*@5 值为 97.91%, *R*@10 值为 99.30%. 详细数据见表 5.

表 5　模型性能表现 (%)

Text-Encoder	FLSE模块	*R*@1	*R*@5	*R*@10
RNN		82.63	93.75	97.91
	√	81.64	96.53	97.95
Transformer		88.19	97.22	99.30
	√	90.27	97.91	99.30
Mamba		86.80	99.30	99.30
	√	95.13	99.30	99.30

前置处理中的原始中文法律合同数据集是以政府采购网站为来源合法合规采集, 总计 64 616 篇真实公开的买卖、租赁和保险合约. 在原始数据集中, 许多合同以照片图像或 PDF 格式呈现, 难以被直接处理. 因此选取相对清晰的合同, 并使用 DocTransPro 将其转换为可进一步处理的 TXT 格式文本. DocTransPro 是自主设计的基于 CnOcr 文档格式转换和表格识别设计的工具, 该工具基于 PDF 解析器和 pdf2word 库的多层次处理, 采用 Otsu 阈值法和形态学操作提高图像质量, 结合 CnOcr 库的深度学习模型进行高精度文本识别, 以及通过计算机视觉方法进行表格结构提取, 实现高效的文档转换和数据保存. DocTransPro 的流程如图 10 所示. 定义文档转换保真度为正确转换的有效字符数与原文档有效字符总数的比值, 通过测试, 使用本工具后图片 PDF 保真度可达 99.452%, 文字 PDF 保真度可达 99.988%, 可以满足整体项目的基本需求.

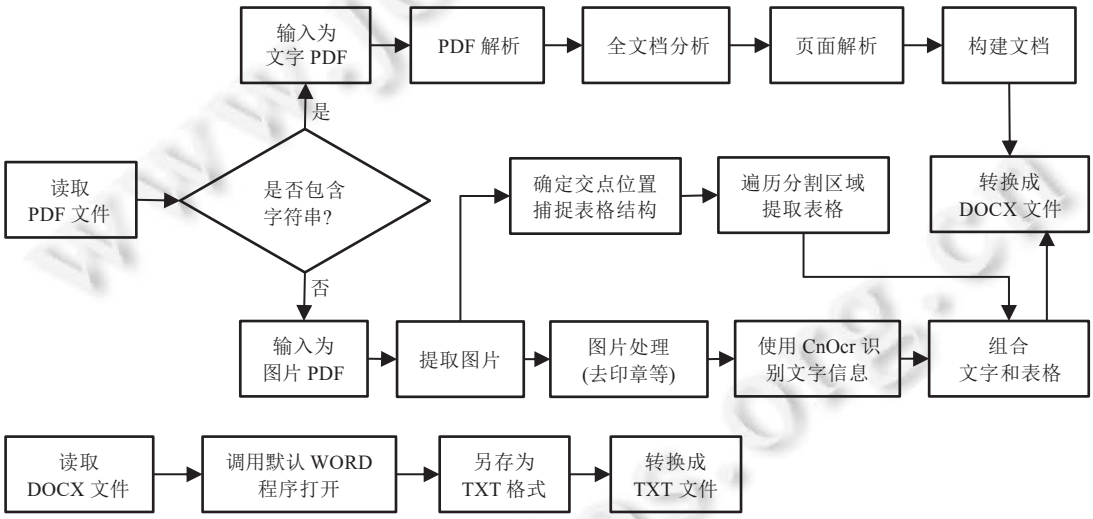

图 10　DocTransPro 转换流程图

为了实现要素提取工作, 从所收集的合同中选取了 1 000 篇买卖合同、303 篇租赁合同、302 篇保险合同作为目标, 使用自动要素标注工具进行标注. 基于智能合约转化需求以及 3 大类型合约的特点, 针对不同类型的合同, 定义了总计 11 大类 30 小类的要素, 粒度细且覆盖范围广泛, 文本的跨度范围+类别标签的形式记录在数据集中. 最后, 为了更好地模拟真实场景中的 CEE, 模型基于句末标点符号和换行符将合同分解为句子. 经过处理, 构建了包含 270 592 条数据的数据集. 根据合同数量, 以 70:15:15 的比例将数据集分为训练集、测试集和开发集进行构建要素提取模型, 确保每个集都有所有类别. 合同数据的统计结果如表 6 所示, 其中的 NL (Null) 代表无效类别样本.

Micro *F*1 和 Macro *F*1 是用于评估多类别分类模型性能的指标, Micro *F*1 将所有类别的预测汇总在一起计算

整体的精确率和召回率, 重视每个样本的贡献, 适合处理类别不平衡的数据; 而 Macro $F1$ 则对每个类别分别计算 $F1$ 分数, 然后取平均值, 适合分析模型在所有类别上的一致性, 但容易受到少量样本类别的影响. 两个指标在多类别任务中常结合使用以全面衡量模型的整体表现与类别间的均衡性. 目前, 在所构建的大规模中文买卖合约数据集上, 所提出的模型可在令牌级实现 91.87% 的准确率, 在要素级实现 80.05% 的 Micro $F1$ 指标和 75.77% 的 Macro $F1$ 指标.

表 6　合同数据集数目统计

大类	小类	训练集	测试集	验证集	总和
NL	NL	190312	41872	38408	270592
甲方	甲方	1089	239	238	1566
乙方	乙方	1175	256	257	1688
签订时间	签订时间	612	128	127	867
合同期限	期限	412	94	74	580
	生效期	31	4	5	40
	等待期	13	5	3	21
保险额度	责任限额	296	45	66	407
	免赔额	129	42	34	205
	保险费	24	5	2	31
支付	租金	220	39	43	302
	押金	7	5	1	13
	支付频率	102	20	20	142
	条件	1607	360	327	2294
	后置条件	196	47	34	277
	行为	1536	331	298	2165
发货	条件	409	87	104	600
	后置条件	225	45	68	338
	行为	432	94	106	632
验收	条件	436	96	72	604
	后置条件	246	64	57	367
	行为	474	97	88	659
终止违约条款	条件	2136	459	436	3031
	合同终止	1582	360	313	2255
	结果	2163	472	476	3111
非终止违约条款	条件	3382	739	666	4787
	后置条件	1067	240	238	1545
	结果	4025	861	831	5717
标的	名称	3247	1029	665	4941
	数量	2248	677	515	3440
	单价	1824	714	379	2917
	总价	2194	706	475	3375

注: 小类为对应大类的细分项

4.3 相关工作的介绍和对比

近年来智能合约形式化验证的研究不断深入, 目前验证技术主要包括 3 种形式化方法: 定理证明、符号执行和模型检测[33]. 这些方法各有特点, 分别广泛应用于不同场景下.

定理证明是一种基于数学逻辑的验证方法, 通过严格的推理规则验证智能合约的逻辑正确性. 其基本原理是将智能合约的操作形式化为逻辑命题, 通过预定义的公理体系进行演绎推导, 证明其满足特定的性质. 例如, F*框

架利用定理证明技术形式化合约语义, 并验证安全性属性, 确保合约满足指定的逻辑规则[34]. 定理证明的主要优点是验证结果严谨且适用范围广泛, 适合复杂系统的逻辑验证, 但其劣势在于依赖大量的人工参与和规则定义, 难以实现自动化.

符号执行是一种静态分析技术, 通过将程序变量替换为符号并模拟合约的运行路径, 从而验证合约的功能和漏洞. 符号执行的核心在于能够探索程序的多种执行路径, 发现潜在的问题. 例如, Mythril 工具对以太坊智能合约的 EVM 字节码执行符号化分析, 检测常见的漏洞, 如重入攻击和整数溢出等[35]. 符号执行的优势在于其能够覆盖智能合约中多种复杂的执行路径. 符号执行适合漏洞检测和动态分析, 但也存在路径爆炸等问题局限, 致使复杂合约验证效率较低.

模型检测是一种基于状态空间的自动化验证方法, 通过对系统的状态转移进行穷尽搜索来验证合约是否满足指定的属性. 该方法将合约抽象为一个有限状态机, 利用逻辑公式描述其属性, 并通过工具对其进行验证. 例如, VeriSmart 模型检测工具面向以太坊智能合约的安全验证, 通过结合符号化状态搜索和逻辑属性检查, 精确检测智能合约中的逻辑漏洞, 如重入攻击、整数溢出等常见问题[36]. 模型检测的优点是自动化程度高、验证全面, 但由于状态数量随变量和逻辑复杂性指数增长, 其主要问题是容易出现状态空间爆炸, 在处理大型合约时效率有限.

本文提出的转化验证模型体现了形式化建模与属性验证的思想, 通过对法律合同的形式化描述, 将其转化为满足智能合约执行要求的模型. 本文模型可归为模型检测技术范畴, 其核心在于利用计算树逻辑 (CTL) 对合同的属性进行验证, 并将验证通过的合同转化为可执行的智能合约代码. 本文模型在逻辑属性验证方法上与 VeriSmart 等模型具有相似性, 如均采用逻辑描述合约属性并通过工具进行自动化验证. 然而, 本模型的独特之处在于结合了法律框架, 提出了文法要求、非赋权原则、有效性审查和安全性准则四大原则, 将形式化验证与法律合规性紧密结合. 这使得本文模型特别适用于需要法律效力的智能合约场景. 相比之下, VeriSmart 更偏重于技术层面的逻辑一致性和漏洞检测, 而本文模型则在实现逻辑一致性的基础上, 更注重合约的法律适应性与约束力, 能够有效解决传统智能合约在法律适用性和动态调整能力上的不足.

此外, 本文模型在转化过程中的严密性设计确保了合约的逻辑一致性和法律合规性, 尤其在法律化要求的场景下表现出显著优势. 这种双重保障使得本文模型在实际应用中具备更高的实用价值. 相比于现有的技术模型, 本文提出的解决方案不仅提供了强有力的形式化验证, 还扩展了智能合约的法律合规性和可操作性.

4.4 模型的复杂性与局限性

作为一种结合法律合同与智能合约生成的综合性框架, LTSC-TAV 模型设计的目标是实现从合同文本到可执行智能合约的高效转化. 尽管模型在中小规模合同验证中表现出较高的适用性和灵活性, 但在复杂合同场景下的扩展能力和计算效率仍面临挑战, 需要更全面的评估与优化.

首先, 在大型复杂合同的处理方面, 模型需要应对条款嵌套深度增加、语义多样性扩展以及特殊场景适配性不足的难题. 复杂合同往往涉及多层嵌套逻辑, 例如嵌套条件、依赖条款和循环义务等, 这对现有的语义解析与逻辑验证模块提出了更高要求. 针对特定行业需求 (如金融合同中的风险评估、物流合同中的实时追踪义务), 目前的通用框架缺乏更个性化的弹性扩展能力. 因此, 模型在应对非标准化合同时, 可能难以完全适配高度特定条款的执行.

其次, 模型的计算限制主要集中在语义解析和属性验证两个关键环节. 语义解析模块依赖于生成高维语义向量以实现合同条款与法律条文的匹配. 这一过程需要大量计算资源, 当合同条款数量增多且语义复杂性提升时, 计算成本将呈指数级增长. 此外, 属性验证阶段的时间复杂度也较高. 模型通过 CTL 公式进行属性推导和验证, 而属性规则数量的增加将导致验证任务的非线性增长. 针对动态调整规则时, 这种计算负担将进一步加剧, 从而降低整体效率.

再者, 输入质量也是模型的潜在瓶颈, 影响整体性能. LTSC-TAV 模型高度依赖于前置处理阶段的合同要素提取与逻辑关系抽取, 其精度直接决定了后续验证的准确性与效率. 然而, 在应对多语言合同或上下文关联性较强的条款时, 现有技术可能难以完全捕获细节, 影响智能合约的最终生成质量. 此外, 模型对异常情况 (如非结构化数据、

模糊条款等) 的处理能力仍存在改进空间. 这些都可能在实际应用中限制模型的鲁棒性和普适性.

最后, 在通用性方面, LTSC-TAV 模型的验证逻辑具有一定的普适性, 理论上可以适用于其他语言或国家的法律合同. 但由于中文语义特性和法律体系的差异, 模型的直接迁移存在挑战. 不同语言在语法结构、语义表达上有所不同, 需要调整要素提取、一致性判定等前置处理规则. 而各国法律对合同条款的要求不同, 也需要重新设计模型的验证属性和生成逻辑, 以适配目标法律体系. 因此, 模型的跨语言、跨国家应用需结合目标语言特点和法律规范进行调整, 以保证其准确性和适用性.

因此, LTSC-TAV 模型具有较大的优化潜力, 可以从以下几个方面进行改进.

(1) 引入分层处理机制: 根据条款的重要性和优先级动态调整处理流程, 优先解析和验证核心条款. 通过集中资源处理关键条款的方法, 显著降低整体计算压力, 同时保障核心条款验证的高效性和准确性.

(2) 采用分布式计算框架: 通过分布式部署将语义解析与属性验证任务分散至多个计算节点, 以提升并行处理能力. 结合图神经网络进行状态压缩, 进一步缓解状态空间爆炸问题.

(3) 开发行业定制化模块: 针对特定行业 (如金融、物流等) 的合同需求, 设计专用模块, 以应对条款高度定制化的挑战. 通过模型微调和领域知识注入, 提升模型对行业合同的适应能力.

(4) 优化语义解析技术: 利用更先进的大规模预训练语言模型加强语义解析能力, 同时结合注意力机制优化语义匹配的精度和效率. 对语义向量进行动态归一化处理, 减少噪声对匹配结果的干扰.

(5) 集成动态验证优化: 通过属性分组和逻辑简化技术, 动态调整验证流程, 减少冗余验证任务. 对于实时场景, 可引入增量验证方法, 仅针对新增条款或规则进行部分验证, 从而提高响应效率.

总之, LTSC-TAV 模型在智能合约法律化转化中的理论框架基础上具有创新性和强大潜力, 但在复杂场景下的计算能力和适应性仍有待完善. 通过优化分层处理、分布式计算和语义解析等关键技术, 可进一步提升模型的效率和泛化能力, 满足多样化场景的合同需求. 为一种结合法律合同与智能合约生成的综合性框架, LTSC-TAV 模型设计目标是实现从合同文本到可执行智能合约的高效转化. 模型在现有的中小规模合同验证中表现出较强的适用性和灵活性, 但其在复杂合同场景下的扩展能力和计算效率仍需进一步评估.

5 总结与展望

5.1 总　结

本研究针对传统纸质合同在执行和管理中的效率问题, 首先讨论了法律合同与智能合约的关系, 有助于法律专业人士和技术开发者充分理解这种新形式合同. 其次基于两者的本质差异提出了文法要求、非赋权原则、有效性审查和安全性准则四大原则, 为智能合约的生成与执行提供了理论基础, 推动了智能合约技术法律化进程. 此外, 构建了转化与验证模型 LTSC-TAV, 根据过渡系统的操作语义生成功能等效的 Solidity 代码 (附录 A.2). 且通过形式化方法验证其四大原则属性, 有效解决了传统纸质合同在管理与执行中的效率问题, 提升了法律合同处理的便捷性与灵活性. 最后展示了具体的买卖合同转化验证案例和要素提取、一致性判定等其他实验数据, 通过实验验证了理论和模型的真实效果, 展示了智能合约在实际应用中的潜力和优势, 提高了模型的说服力和实用价值. 本研究对智能合约的规范化生产和应用具有重要意义.

5.2 未来工作

展望未来, LTSC-TAV 模型研究将聚焦于智能合约法律化的核心议题, 在多层次进一步拓展其应用价值.

在深度上, 模型需要面向更加复杂和多样化的合同场景, 开发动态调整机制以适应大型合同中复杂的逻辑嵌套和利益相关者的交互需求. 模型应增强对动态合同条款和实时变更的支持, 确保其高效应对复杂场景中的条款调整与状态更新. 此外, 还需扩展对特定行业的支持, 构建行业定制化模块, 满足金融、物流等垂直领域的复杂需求, 从而提升其适用性和精准度.

在广度上, 模型需进一步提升其通用性和计算效率. 当前的计算负担主要集中在一致性判定与属性验证环节, 未来可通过底层算法优化与分布式架构部署降低计算复杂度. 可以引入剪枝算法和语义压缩技术, 有效减少无效

计算, 提高验证效率. 或应用分布式计算框架实现任务的高效分工与并行处理, 从而支持大规模合同的快速处理和实时验证. 此外, 模型在区块链平台上的适配性仍需进一步完善, 探索适应不同区块链协议的智能合约生成机制将成为重要方向.

安全性方面, 智能合约领域的潜在风险与日俱增, LTSC-TAV 模型需要紧跟技术前沿整合新型加密技术、形式化验证工具和安全防护机制, 更有效地应对多样化的漏洞威胁, 进一步提高其对抗性和韧性.

为了实现这些目标, 未来可以从以下几个方面重点突破: 一是通过引入分层处理机制, 按条款的重要性动态分配资源, 优化模型的核心功能; 二是结合大规模预训练语言模型显著提升语义解析的精度与效率; 三是通过动态验证技术与增量验证策略的结合, 在提高响应速度的同时, 确保复杂合同的全面验证.

总之, 随着性能的不断提升, LTSC-TAV 模型将在多行业、多场景的复杂合同管理中发挥更加广泛且深远的影响.

References:

[1] Schwartz A, Scott RE. Contract theory and the limits of contract law. Yale Law Journal, 2003, 113(3): 541–619. [doi: 10.2307/3657531]

[2] Szabo N. The idea of smart contracts. Satoshi Nakamoto Institute. 1997. https://nakamotoinstitute.org/library/the-idea-of-smart-contracts/

[3] Szabo N. Smart contracts. Satoshi Nakamoto Institute. 1994. https://nakamotoinstitute.org/smart-contracts/

[4] Buterin V. A next-generation smart contract and decentralized application platform. White paper. 2014. https://ethereum.org/en/whitepaper/

[5] Tan HB, Zhou T, Zhao H, Zhao Z, Wang WD, Zhang ZX, Sheng NZ, Li XF. Archival data protection and sharing method based on blockchain. Ruan Jian Xue Bao/Journal of Software, 2019, 30(9): 2620–2635 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5770.htm [doi: 10.13328/j.cnki.jos.005770]

[6] Shi JS, Li R. Survey of blockchain access control in Internet of things. Ruan Jian Xue Bao/Journal of Software, 2019, 30(6): 1632–1648 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5740.htm [doi: 10.13328/j.cnki.jos.005740]

[7] Qian WN, Shao QF, Zhu YC, Jin CQ, Zhou AY. Research problems and methods in blockchain and trusted data management. Ruan Jian Xue Bao/Journal of Software, 2018, 29(1): 150–159 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5434.htm [doi: 10.13328/j.cnki.jos.005434]

[8] Lang F. New interpretation of smart contracts based on blockchain technology from the contractual perspective. Journal of Chongqing University (Social Science Edition), 2021, 27(5): 169–182 (in Chinese with English abstract). [doi: 10.11835/j.issn.1008-5831.fx.2020.04.001]

[9] Stazi A. Smart Contracts and Comparative Law: A Western Perspective. Cham: Springer, 2021. 84–87. [doi: 10.1007/978-3-030-83240-7]

[10] Chen JD. The legal structure of smart contract. Orient Law, 2019, (3): 18–29 (in Chinese with English abstract). [doi: 10.19404/j.cnki.dffx.20190307.010]

[11] Cappiello B, Carullo G. Blockchain, Law and Governance. Cham: Springer, 2021. 184–185. [doi: 10.1007/978-3-030-52722-8]

[12] He HW, Yan A, Chen ZH. Survey of smart contract technology and application based on blockchain. Journal of Computer Research and Development, 2018, 55(11): 2452–2466 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2018.20170658]

[13] Liu Q, Wang DJ, Wang XX, Zheng XR, Meng B. Review on conformance between legal contract and smart contract. Application Research of Computers, 2021, 38(1): 1–8 (in Chinese with English abstract). [doi: 10.19734/j.issn.1001-3695.2019.12.0652]

[14] Si X, Cao JF. On the civil liability of artificial intelligence: Focus on autonomous vehicles and intelligent robots. Science of Law (Journal of Northwest University of Political Science and Law), 2017, 35(5): 166–173 (in Chinese with English abstract). [doi: 10.16290/j.cnki.1674-5205.2017.05.016]

[15] Maróti M, Kecskés T, Kereskényi R, Broll B, Völgyesi P, Jurácz L, Levendoszky T, Lédeczi Á. Next generation (meta) modeling: Web- and cloud-based collaborative tool infrastructure. In: Proc. of the 2014 MPM. 2014. 41–60.

[16] Mavridou A, Laszka A. Designing secure ethereum smart contracts: A finite state machine based approach. In: Proc. of the 22nd Int'l Conf. on Financial Cryptography and Data Security. Nieuwpoort: Springer, 2018. 523–540. [doi: 10.1007/978-3-662-58387-6_28]

[17] Mavridou A, Laszka A. Tool demonstration: FSolidM for designing secure ethereum smart contracts. In: Proc. of the 7th Int'l Conf. on Principles of Security and Trust. Thessaloniki: Springer, 2018. 270–277. [doi: 10.1007/978-3-319-89722-6_11]

[18] Basu A, Bensalem B, Bozga M, Combaz J, Jaber M, Nguyen TH, Sifakis J. Rigorous component-based system design using the BIP framework. IEEE Software, 2011, 28(3): 41–48. [doi: 10.1109/MS.2011.27]

[19] Bliudze S, Cimatti A, Jaber M, Mover S, Roveri M, Saab W, Wang Q. Formal verification of infinite-state BIP models. In: Proc. of the

13th Int'l Symp. on Automated Technology for Verification and Analysis. Shanghai: Springer, 2015. 326–343. [doi: 10.1007/978-3-319-24953-7_25]

[20] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I. Attention is all you need. In: Proc. of the 31st Int'l Conf. on Neural Information Processing Systems. Long Beach: Curran Associates Inc., 2017. 6000–6010.

[21] Zhang X, Zhao JB, LeCun Y. Character-level convolutional networks for text classification. In: Proc. of the 29th Int'l Conf. on Neural Information Processing Systems. Montreal: MIT Press, 2015. 649–657.

[22] Lin TY, Goyal P, Girshick R, He KM, Dollar P. Focal loss for dense object detection. In: Proc. of the 2017 IEEE Int'l Conf. on Computer Vision (ICCV). Venice: IEEE, 2017. 2999–3007. [doi: 10.1109/ICCV.2017.324]

[23] Alur R, Dill DL. A theory of timed automata. Theoretical Computer Science, 1994, 126(2): 183–235. [doi: 10.1016/0304-3975(94)90010-8]

[24] Solidity Documentation. Common patterns. 2024. http://solidity.readthedocs.io/en/develop/common-patterns.html#state-machine

[25] Plotkin GD. A structural approach to operational semantics. The Journal of Logic and Algebraic Programming, 2004, 60–61: 17–139. [doi: 10.1016/j.jlap.2004.05.001]

[26] Milner R. Communication and Concurrency. New York: Prentice Hall, 1989.

[27] Wang XB, Yang XY, Shu XF, Zhao L. Formal verification of smart contract based on MSVL. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1849–1866 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6253.htm [doi: 10.13328/j.cnki.jos.006253]

[28] Han N, Li XM, Zhang QY, Wang GH, Shi ZP, Guan Y. Executable semantics of Ethereum intermediate language. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1717–1732 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6246.htm [doi: 10.13328/j.cnki.jos.006246]

[29] Wang PW, Yang HT, Meng J, Chen JC, Du XY. Formal definition for classical smart contracts and reference implementation. Ruan Jian Xue Bao/Journal of Software, 2019, 30(9): 2608–2619 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5773.htm [doi: 10.13328/j.cnki.jos.005773]

[30] Fu LQ, Tian HB. Ethereum coin voting protocol based on smart contract. Ruan Jian Xue Bao/Journal of Software, 2019, 30(11): 3486–3502 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5575.htm [doi: 10.13328/j.cnki.jos.005575]

[31] Baier C, Katoen JP. Principles of Model Checking. Representation and Mind Series. Cambridge: The MIT Press, 2008.

[32] Clarke EM, Grumberg O, Long DE. Model checking and abstraction. ACM Trans. on Programming Languages and Systems, 1994, 16(5): 1512–1542. [doi: 10.1145/186025.186051]

[33] Zhang WB, Chen SM, Wei LF, Song W, Huang DM. State-of-the-art survey of smart contract verification based on formal methods. Chinese Journal of Network and Information Security, 2022, 8(4): 12–28 (in Chinese with English abstract). [doi: 10.11959/j.issn.2096-109x.2022041]

[34] Bhargavan K, Delignat-Lavaud A, Fournet C, Gollamudi A, Gonthier G, Kobeissi N, Kulatova N, Rastogi A, Sibut-Pinote T, Swamy N, Zanella-Béguelin S. Formal verification of smart contracts: Short Paper. In: Proc. of the 2016 ACM Workshop on Programming Languages and Analysis for Security. Vienna: ACM, 2016. 91–96. [doi: 10.1145/2993600.2993611]

[35] ConsenSys. Mythril: A Security Analysis Tool for EVM Bytecode. 2024. https://github.com/ConsenSys/mythril

[36] So S, Lee M, Park J, Lee H, Oh H. VERISMART: A highly precise safety verifier for Ethereum smart contracts. In: Proc. of the 2020 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE, 2020. 1678–1694. [doi: 10.1109/SP40000.2020.00032]

[37] Jiao J, Kan SL, Lin SW, Sanan D, Liu Y, Sun J. Executable operational semantics of solidity. arXiv:1804.01295, 2018.

[38] Solidity Documentation. Security considerations. 2024. http://solidity.readthedocs.io/en/develop/security-considerations.html

[39] Finley K. A $50 million hack just showed that the DAO was all too human. 2016. https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/

[40] Newman LH. Security news this week: $280m worth of Ethereum is trapped thanks to a dumb bug. 2017. https://www.wired.com/story/280m-worth-of-ethereum-is-trapped-for-a-pretty-dumb-reason/

附中文参考文献:

[5] 谭海波, 周桐, 赵赫, 赵哲, 王卫东, 张中贤, 盛念祖, 李晓风. 基于区块链的档案数据保护与共享方法. 软件学报, 2019, 30(9): 2620–2635. http://www.jos.org.cn/1000-9825/5770.htm [doi: 10.13328/j.cnki.jos.005770]

[6] 史锦山, 李茹. 物联网下的区块链访问控制综述. 软件学报, 2019, 30(6): 1632–1648. http://www.jos.org.cn/1000-9825/5740.htm [doi: 10.13328/j.cnki.jos.005740]

[7] 钱卫宁, 邵奇峰, 朱燕超, 金澈清, 周傲英. 区块链与可信数据管理: 问题与方法. 软件学报, 2018, 29(1): 150–159. http://www.jos.org.cn/1000-9825/5434.htm [doi: 10.13328/j.cnki.jos.005434]

[8] 郎芳. 区块链技术下智能合约之于合同的新诠释. 重庆大学学报 (社会科学版), 2021, 27(5): 169–182. [doi: 10.11835/j.issn.1008-5831.fx.2020.04.001]
[10] 陈吉栋. 智能合约的法律构造. 东方法学, 2019, (3): 18–29. [doi: 10.19404/j.cnki.dffx.20190307.010]
[12] 贺海武, 延安, 陈泽华. 基于区块链的智能合约技术与应用综述. 计算机研究与发展, 2018, 55(11): 2452–2466. [doi: 10.7544/issn1000-1239.2018.20170658]
[13] 刘琴, 王德军, 王潇潇, 郑绪睿, 孟博. 法律合约与智能合约一致性综述. 计算机应用研究, 2021, 38(1): 1–8. [doi: 10.19734/j.issn.1001-3695.2019.12.0652]
[14] 司晓, 曹建峰. 论人工智能的民事责任: 以自动驾驶汽车和智能机器人为切入点. 法律科学 (西北政法大学学报), 2017, 35(5): 166–173. [doi: 10.16290/j.cnki.1674-5205.2017.05.016]
[27] 王小兵, 杨潇钰, 舒新峰, 赵亮. 面向 MSVL 的智能合约形式化验证. 软件学报, 2021, 32(6): 1849–1866. http://www.jos.org.cn/1000-9825/6253.htm [doi: 10.13328/j.cnki.jos.006253]
[28] 韩宁, 李希萌, 张倩颖, 王国辉, 施智平, 关永. 以太坊中间语言的可执行语义. 软件学报, 2021, 32(6): 1717–1732. http://www.jos.org.cn/1000-9825/6246.htm [doi: 10.13328/j.cnki.jos.006246]
[29] 王璞巍, 杨航天, 孟佶, 陈晋川, 杜小勇. 面向合同的智能合约的形式化定义及参考实现. 软件学报, 2019, 30(9): 2608–2619. http://www.jos.org.cn/1000-9825/5773.htm [doi: 10.13328/j.cnki.jos.005773]
[30] 付利青, 田海博. 基于智能合约的以太币投票协议. 软件学报, 2019, 30(11): 3486–3502. http://www.jos.org.cn/1000-9825/5575.htm [doi: 10.13328/j.cnki.jos.005575]
[33] 张文博, 陈思敏, 魏立斐, 宋巍, 黄冬梅. 基于形式化方法的智能合约验证研究综述. 网络与信息安全学报, 2022, 8(4): 12–28. [doi: 10.11959/j.issn.2096-109x.2022041]

附录 A. 形式化定义

A.1 支持的 Solidity 子集

本节以形式化的方式定义 LTSC-TAV 支持的 Solidity 子集. 首先, 使用以下符号代替特定集合.

T: Solidity 类型集合.

I: 有效 Solidity 标识符集合.

D: Solidity 事件和自定义类型定义集合.

其中, 支持的事件 (⟨*event*⟩) 和自定义类型 (⟨*struct*⟩) 如下.

$$\langle event\rangle ::= \textbf{event}\ @identifier((@type\ @identifier\ \ (,\ @type\ @identifier)*)?);$$

$$\langle struct\rangle ::= \textbf{struct}\ @identifier\ \{(@type\ @identifier;)*\}$$

E: Solidity 表达式集合.

C: 没有副作用的 Solidity 表达式集合 (即评估时不会更改存储、内存、余额等的表达式).

包含内容如下.

$$\begin{aligned}\langle pure\rangle ::= &|\langle variable\rangle\\ &|@constant\\ &|(\langle pure\rangle)\\ &|\langle unary\rangle\langle pure\rangle\\ &|\langle pure\rangle\langle operator\rangle\langle pure\rangle\\ \langle variable\rangle ::= &|@identifier\\ &|\langle variable\rangle.@identifier\\ &|\langle identifier\rangle[\langle pure\rangle]\end{aligned}$$

$$\langle operator\rangle ::= == \mid != \mid < \mid > \mid >= \mid <= \mid + \mid * \mid - \mid / \mid \% \mid \&\& \mid \| \mid$$

$$\langle unary\rangle ::= ! \mid + \mid -$$

S: 支持的 Solidity 语句集合.

支持语句的类型如下.

变量声明 (如: int256 value = 0).

表达式 (如: msg.sender.transfer(amount);).

事件语句 (如: emit Deposit(amount, msg.sender);).

返回语句 (: 如: return; 和 return amount;).

选择语句 (包括 if … else if … 等).

循环语句 (包括 for 和 while).

复合语句 (即 { *statement*1 *statement*2 … }).

定义 **S** 子集的正式语法如下.

$$\begin{aligned}\langle statement\rangle ::= &|\langle declaration\rangle;\\ &|@expression;\\ &|\textbf{emit } @identifier((@expression\ \ (,\ @expression)*)?);\\ &|\textbf{return } (@pure)?;\\ &|\textbf{if } (@expression)\langle statement\rangle\ (\textbf{else } \langle statement\rangle)?\\ &|\textbf{for } (\langle declaration\rangle;\ @expression;\ @expression)\ \langle statement\rangle\\ &|\textbf{while } (@expression)\ \langle statement\rangle\\ &|\{(\langle statement\rangle)*\}\end{aligned}$$

$$\langle declaration\rangle ::= @type\ @identifier\ (= @expression)?$$

其中, $@expression \in \mathbf{E}$ 表示 Solidity 主表达式, 可以包括函数调用、转账等, 而 $@pure \in \mathbf{C}$ 是没有副作用的 Solidity 表达式.

A.2 过渡系统的操作语义

令 Ψ 表示账本的状态, 其中包括所有合约中账户余额、状态变量、最后一个区块的编号和时间戳等. 在执行过渡期间, 执行状态 $\sigma = \{\Psi, M\}$ 还包括内存和堆栈状态 M. 此外, 引入执行状态 v 处理返回语句和异常. 当发生异常时, $R[v]$ 表示返回语句以数值 v 执行时的执行状态 (即 return v), 而 N 表示没有引发异常. 最后, 定义 $\text{Eval}(\sigma, \text{Exp}) \to \left\langle\left(\hat{\sigma}, x\right), v\right\rangle$ 表示在执行状态 σ 下评估 Solidity 表达式 Exp 以生成值 v, 并且作为副作用更改执行状态 $\hat{\sigma}$ 和执行状态 x.

过渡通常由一个过渡名 (即函数名) $name \in \mathbf{I}$ 和一组参数值 $v_1, v_2, \ldots$ 触发. 没有返回值的正常执行将账本从状态 Ψ 变为 Ψ', 将合约从状态 $s \in S$ 变为 $s' \in S$. 该过渡通过 TRANSITION 规则捕获.

$$\frac{\begin{array}{c}t \in T, name = t^{\text{name}}, s = t^{\text{from}}\\ M = Params(t, v_1, v_2, \ldots), \sigma = (\Psi, M)\\ \text{Eval}(\sigma, g_t) \to \left\langle\left(\hat{\sigma}, N\right), \text{true}\right\rangle\\ \left\langle\left(\hat{\sigma}, N\right), a_t\right\rangle \to \left\langle\left(\hat{\sigma}', N\right), \cdot\right\rangle\\ \hat{\sigma}' = (\Psi', M'), s' = t^{\text{to}}\end{array}}{\text{TRANSITION} \quad \langle(\Psi, s), name(v_1, v_2, \ldots)\rangle \to \langle(\Psi', s', \cdot)\rangle}$$

该规则适用情况为: 存在一个过渡 t, 其名字 t^{name} 为 $name$, 其源状态 t^{from} 是当前合约状态 s (第 1 行). 执行状态 σ 通过取参数值 $Params(t, v_1, v_2, \ldots)$ 和当前账本状态 Ψ(第 2 行) 初始化. 如果当前状态 σ 下的守卫条件 g_t 评估为 true 且引发没有异常, 那么过渡操作 a_t 将被执行 (第 4 行), 这将导致执行状态更新为 σ'(见附录 A.3 中的语句规

则). 最后, 如果操作结果中的执行状态是正常的 N (即没有引发异常), 则更新后的账本状态 Ψ' 和更新后的合约状态 s' 将被永久保存 (第 5 行).

正常执行的返回值由 TRANSITION-RET 规则捕获.

$$t \in T, name = t^{\text{name}}, s = t^{\text{from}}$$

$$M = Params(t, v_1, v_2, \ldots), \sigma = (\Psi, M)$$

$$\text{Eval}(\sigma, g_t) \to \left\langle \left(\hat{\sigma}, N\right), \text{true} \right\rangle$$

$$\left\langle \left(\hat{\sigma}, N\right), a_t \right\rangle \to \left\langle \left(\hat{\sigma}', R[v]\right), \cdot \right\rangle$$

$$\hat{\sigma}' = (\Psi', M'), s' = t^{\text{to}}$$

$$\text{TRANSITION-RET} \quad \overline{\langle (\Psi, s), name(v_1, v_2, \ldots) \rangle \to \langle (\Psi', s', v) \rangle}$$

当过渡操作 a_t 以返回 v 状态结束时, 该规则适用, 执行状态最终为 $R[v]$.

如果名字 $name = t^{\text{name}}$ 的过渡 t 存在, 但其源状态 t^{from} 不是 s, 则由过渡 TRANSITION-WRO 规则捕获.

$$t \in T, name = t^{\text{name}}, s \neq t^{\text{from}}$$

$$\text{TRANSITION-WRO} \quad \overline{\langle (\Psi, s), name(v_1, v_2, \ldots) \rangle \to \langle (\Psi, s, \cdot) \rangle}$$

同样, 如果当前状态下过渡的守卫条件 g_t 评估为 false, 则该过渡将被回退, 这由过渡 TRANSITION-GRD 规则捕获.

$$t \in T, name = t^{\text{name}}, s = t^{\text{from}}$$

$$M = Params(t, v_1, v_2, \ldots), \sigma = (\Psi, M)$$

$$\text{Eval}(\sigma, g_t) \to \left\langle \left(\hat{\sigma}, N\right), \text{false} \right\rangle$$

$$\text{TRANSITION-GRD} \quad \overline{\langle (\Psi, s), name(v_1, v_2, \ldots) \rangle \to \langle (\Psi, s, \cdot) \rangle}$$

如果在评估守卫条件 g_t 时引发异常 (即执行状态变为 E), 则该过渡将被回退, 这由过渡 TRANSITION-EXC1 规则捕获.

$$t \in T, name = t^{\text{name}}, s = t^{\text{from}}$$

$$M = Params(t, v_1, v_2, \ldots), \sigma = (\Psi, M)$$

$$\text{Eval}(\sigma, g_t) \to \left\langle \left(\hat{\sigma}, E\right), x \right\rangle$$

$$\text{TRANSITION-EXC1} \quad \overline{\langle (\Psi, s), name(v_1, v_2, \ldots) \rangle \to \langle (\Psi, s, \cdot) \rangle}$$

同样, 如果在执行过渡操作 a_t 时引发异常, 则该过渡将被回退, 这由过渡 TRANSITION-EXC2 规则捕获.

$$t \in T, name = t^{\text{name}}, s = t^{\text{from}}$$

$$M = Params(t, v_1, v_2, \ldots), \sigma = (\Psi, M)$$

$$\text{Eval}(\sigma, g_t) \to \left\langle \left(\hat{\sigma}, N\right), \text{true} \right\rangle$$

$$\left\langle \left(\hat{\sigma}, N\right), a_t \right\rangle \to \left\langle \left(\hat{\sigma}', E\right), \cdot \right\rangle$$

$$\text{TRANSITION-EXC2}\ ----------$$

$$\langle(\Psi,s),name(v_1,v_2,\ldots)\rangle\rightarrow\langle(\Psi,s,\cdot)\rangle$$

另一方面, 如果不存在名字为 *name* 的过渡, 则执行回退操作 a_F, 这由过渡 TRANSITION-FAL 规则捕获.

$$\forall t\in T,name\neq t^{\text{name}}$$

$$\sigma=\{\Psi,\varnothing\}$$

$$\left\langle\left(\hat{\sigma},N\right),a_F\right\rangle\rightarrow\left\langle\left(\hat{\sigma}',x\right),\cdot\right\rangle,x\neq E$$

$$\hat{\sigma}'=(\Psi',y)$$

$$\text{TRANSITION-FAL}\ ----------$$

$$\langle(\Psi,s),name(v_1,v_2,\ldots)\rangle\rightarrow\langle(\Psi',s,\cdot)\rangle$$

最后, 如果在执行回退操作 a_F 时引发异常, 则该过渡将被回退, 这由过渡 TRANSITION-EXC3 规则捕获.

$$\forall t\in T,name\neq t^{\text{name}}$$

$$\sigma=\{\Psi,\varnothing\}$$

$$\left\langle\left(\hat{\sigma},N\right),a_F\right\rangle\rightarrow\left\langle\left(\hat{\sigma}',E\right),\cdot\right\rangle$$

$$\text{TRANSITION-EXC3}\ ----------$$

$$\langle(\Psi,s),name(v_1,v_2,\ldots)\rangle\rightarrow\langle(\Psi,s,\cdot)\rangle$$

A.3 支持的 Solidity 语句的操作语义

参考文献 [37] 中已经定义了 Solidity 的小步操作语义, 能够帮助一步步推理出一个计算步骤. 本文扩展了其中的语义以支持异常和返回值, 以一个或多个规则呈现每个支持的 Solidity 语句的语义. 每条规则接收一个执行状态 σ、一个执行状态 $x\in\{N,E,R[v]\}$, 以及一个语句 $\text{Stmt}\in\mathbf{S}$, 并将它们映射到一个新的执行状态和一个剩余的待执行语句 (或如果 · 没有留下待执行语句).

首先, 从适用于每个语句的基本规则开始定义. 如果引发了异常或者执行了返回语句, 则不应再执行任何剩余的语句, 这通过 SKIP-EXC 和 SKIP-RET 规则来捕获.

$$\text{SKIP-EXC}\ ----------$$

$$\langle(\sigma,E),\text{Stmt}\rangle\rightarrow\langle(\sigma,E),\cdot\rangle$$

$$\text{SKIP-RET}\ ----------$$

$$\langle(\sigma,R[v]),\text{Stmt}\rangle\rightarrow\langle(\sigma,R[v]),\cdot\rangle$$

返回语句会更改执行状态为 $R[\cdot]$, 跳过所有剩余的语句, 这通过 RETURN 规则来捕获.

$$\text{RETURN}\ ----------$$

$$\langle(\sigma,N),\text{return;}\rangle\rightarrow\langle(\sigma,R[\cdot]),\cdot\rangle$$

为了返回数值 v, return Exp 语句会将执行状态更改为 $R[v]$, 这通过 RETURN-VAL 规则捕获.

$$\text{Eval}(\sigma,\text{Exp})\rightarrow\langle(\sigma',N),v\rangle$$

$$\text{RETURN-VAL}\ ----------$$

$$\langle(\sigma,N),\text{return Exp;}\rangle\rightarrow\langle(\sigma',R[v]),\cdot\rangle$$

如果在 $\text{Eval}(\sigma,\text{Exp})$ 评估期间引发了异常, 那么执行状态将更改为 E, 这通过 RETURN-EXC 规则来捕获.

$$\text{Eval}(\sigma,\text{Exp})\rightarrow\langle(\sigma',E),v\rangle$$

$$\text{RETURN-EXC}\ ----------$$

$$\langle(\sigma,N),\text{return Exp;}\rangle\rightarrow\langle(\sigma',E),\cdot\rangle$$

复合语句 (即用{}中的语句列表) 通过按顺序执行内部语句来执行, 这通过 COMPOUND 规则捕获.

$$\langle(\sigma,N),\text{Stmt}_1\rangle\rightarrow\langle(\sigma_1,x_1),\cdot\rangle$$

$$\langle(\sigma_1,x_1),\text{Stmt}_2\rangle\rightarrow\langle(\sigma_2,x_2),\cdot\rangle$$

$$\cdots$$

$$\langle(\sigma_{n-1},x_{n-1}),\text{Stmt}_n\rangle\rightarrow\langle(\sigma',x),\cdot\rangle$$

$$\text{COMPOUND}\ ----------$$

$$\langle(\sigma,N),\{\text{Stmt}_1\ \text{Stmt}_2\ \ldots\ \text{Stmt}_n\}\rangle\rightarrow\langle(\sigma',x),\cdot\rangle$$

循环语句. while 语句首先对条件表达式 Exp 进行求值, 如果结果为false, 则跳过循环体 Stmt 的执行, 这由 WHILE1 规则捕获.

$$\text{Eval}(\sigma,\text{Exp})\rightarrow\langle(\sigma',N),\text{false}\rangle$$

$$\text{WHILE1}\ ----------$$

$$\langle(\sigma,N),\text{while(Exp) Stmt}\rangle\rightarrow\langle(\sigma',N),\cdot\rangle$$

类似地, 如果 Exp 的求值结果为异常, 那么循环体 Stmt 也会被跳过, 这由 WHILE-EXC 规则捕获.

$$\text{Eval}(\sigma,\text{Exp})\rightarrow\langle(\sigma',E),x\rangle$$

$$\text{WHILE-EXC}\ ----------$$

$$\langle(\sigma,N),\text{while(Exp) Stmt}\rangle\rightarrow\langle(\sigma',E),\cdot\rangle$$

另一方面, 如果 Exp 求值为 true, 则执行循环体 Stmt, 这由 WHILE2 规则捕获.

$$\text{Eval}(\sigma,\text{Exp})\rightarrow\left\langle\left(\hat{\sigma},N\right),\text{true}\right\rangle$$

$$\left\langle\left(\hat{\sigma},N\right),\text{Stmt}\right\rangle\rightarrow\left\langle\left(\hat{\sigma}',x\right),\cdot\right\rangle$$

$$\text{WHILE2}\ ----------$$

$$\langle(\sigma,N),\text{while(Exp) Stmt}\rangle\rightarrow\left\langle\left(\hat{\sigma}',x\right),\text{while(Exp) Stmt}\right\rangle$$

for 循环语句可以被简化为 while 循环, 这由 FOR 规则捕获.

$$\langle(\sigma,N),\text{Stmt}_I\rangle\rightarrow\langle(\sigma',x),\cdot\rangle$$

$$\text{FOR}\ ----------$$

$$\langle(\sigma,N),\text{for}(\text{Stmt}_I;\text{Stmt}_C;\text{Stmt}_A)\ \text{Stmt}_B\rangle\rightarrow\langle(\sigma',x),\text{while}(\text{Exp}_C)\ \{\text{Stmt}_B\ \text{Stmt}_A\}\rangle$$

条件语句. if 语句通过 IF1、IF2 和 IF-EXC 规则捕获.

$$\text{Eval}(\sigma,\text{Exp})\rightarrow\left\langle\left(\hat{\sigma},N\right),\text{true}\right\rangle$$

$$\left\langle\left(\hat{\sigma},N\right),\text{Stmt}\right\rangle\rightarrow\left\langle\left(\hat{\sigma}',x\right),\cdot\right\rangle$$

$$\text{IF1}\ ----------$$

$$\langle(\sigma,N),\text{if(Exp) Stmt}\rangle\rightarrow\left\langle\left(\hat{\sigma}',x\right),\cdot\right\rangle$$

$$\text{Eval}(\sigma,\text{Exp})\rightarrow\left\langle\left(\hat{\sigma},N\right),\text{false}\right\rangle$$

$$\text{IF2}\ ----------$$

$$\langle(\sigma,N),\text{if(Exp) Stmt}\rangle\rightarrow\left\langle\left(\hat{\sigma},N\right),\cdot\right\rangle$$

$$\mathrm{Eval}(\sigma,\mathrm{Exp})\to\left\langle\left(\hat{\sigma},E\right),x\right\rangle$$

$$\text{IF-EXC}\ ----------$$

$$\langle(\sigma,N),\mathrm{if(Exp)\ Stmt}\rangle\to\langle(\sigma,E),\cdot\rangle$$

类似地, if … else 语句通过 IFELSE1、IFELSE2 和 IFELSE-EXC 规则捕获.

$$\mathrm{Eval}(\sigma,\mathrm{Exp})\to\left\langle\left(\hat{\sigma},N\right),\mathrm{true}\right\rangle$$

$$\left\langle\left(\hat{\sigma},N\right),\mathrm{Stmt}_1\right\rangle\to\left\langle\left(\hat{\sigma}',x\right),\cdot\right\rangle$$

$$\text{IFELSE1}\ ----------$$

$$\langle(\sigma,N),\mathrm{if(Exp)\ Stmt}_1\ \mathrm{else\ Stmt}_2\rangle\to\left\langle\left(\hat{\sigma}',x\right),\cdot\right\rangle$$

$$\mathrm{Eval}(\sigma,\mathrm{Exp})\to\left\langle\left(\hat{\sigma},N\right),\mathrm{false}\right\rangle$$

$$\left\langle\left(\hat{\sigma},N\right),\mathrm{Stmt}_2\right\rangle\to\left\langle\left(\hat{\sigma}',x\right),\cdot\right\rangle$$

$$\text{IFELSE2}\ ----------$$

$$\langle(\sigma,N),\mathrm{if(Exp)\ Stmt}_1\ \mathrm{else\ Stmt}_2\rangle\to\left\langle\left(\hat{\sigma}',x\right),\cdot\right\rangle$$

$$\mathrm{Eval}(\sigma,\mathrm{Exp})\to\left\langle\left(\hat{\sigma},E\right),x\right\rangle$$

$$\text{IFELSE-EXC}\ ----------$$

$$\langle(\sigma,N),\mathrm{if(Exp)\ Stmt}_1\ \mathrm{else\ Stmt}_2\rangle\to\langle(\sigma,E),\cdot\rangle$$

杂项语句. 表达式语句通过 EXPRESSION 规则捕获.

$$\mathrm{Eval}(\sigma,\mathrm{Exp})\to\langle(\sigma',x),v\rangle$$

$$\text{EXPRESSION}\ ----------$$

$$\langle(\sigma,N),\mathrm{Exp;}\rangle\to\langle(\sigma',x),\cdot\rangle$$

变量声明语句通过 VARIABLE 或 VARIABLE-ASG 规则捕获.

$$\mathrm{Decl}(\sigma,\mathrm{Type},\mathrm{Name})\to\langle(\sigma',x)\rangle$$

$$\text{VARIABLE}\ ----------$$

$$\langle(\sigma,N),\mathrm{Type\ Name;}\rangle\to\langle(\sigma',x),\cdot\rangle$$

$$\mathrm{Eval}(\sigma,\mathrm{Exp})\to\langle(\sigma',x),v\rangle$$

$$\text{VARIABLE-ASG}\ ----------$$

$$\langle(\sigma,N),\mathrm{Type\ Name = Exp;}\rangle\to\langle(\sigma',x),\{\mathrm{Type\ Name;Name} = v;\}\rangle$$

其中, $\mathrm{Decl}(\sigma,\mathrm{Type},\mathrm{Name})$ 引入了一个变量到命名空间中 (并在需要时为内存类型的变量扩展内存).

事件语句通过 EVENT 和 EVENT-EXC 规则捕获.

$$\mathrm{Eval}(\sigma,\mathrm{Exp}_1)\to\langle(\sigma_1,N),v_1\rangle$$

$$\cdots$$

$$\mathrm{Eval}(\sigma_{n-1},\mathrm{Exp}_n)\to\langle(\sigma_n,N),v_n\rangle$$

$$\mathrm{Log}(\sigma_n,(name,v_1,\ldots,v_n))\to(\sigma',N)$$

$$\text{EVENT}\ ----------$$

$$\langle(\sigma,N),\mathrm{emit}\ name(\mathrm{Exp}_1,\ldots,\mathrm{Exp}_n);\rangle\to\langle(\sigma',x),\cdot\rangle$$

$$\mathrm{Eval}(\sigma,\mathrm{Exp}_1)\to\langle(\sigma_1,x_1),v_1\rangle$$
$$\dots$$
$$\mathrm{Eval}(\sigma_{n-1},\mathrm{Exp}_n)\to\langle(\sigma_n,x_n),v_n\rangle$$
$$\mathrm{Log}(\sigma_n,(name,v_1,\dots,v_n))\to(\sigma',y)$$
$$x_1=E\vee\dots\vee x_n=E\vee y=E$$
$$\text{EVENT-EXC}\ ----------$$
$$\langle(\sigma,N),\mathrm{emit}\ name(\mathrm{Exp}_1,\dots,\mathrm{Exp}_n);\rangle\to\langle(\sigma',x),\cdot\rangle$$

其中, Log 记录了区块链上指定的值.

附录 B. 背景

B.1 Solidity 函数调用

Solidity 代码中嵌套函数的调用是导致一些已识别漏洞的主要原因, 在此简要描述一个智能合约如何调用另一个合约中的函数或进行代理执行, 更多信息可参考 Solidity 智能合约文档[38]. 一个合约可以有多种方式调用另一个合约中定义的函数.

(1) *addressOfContract*.call(*data*): 低级调用. 函数的名称和参数需要根据以太坊 ABI 在 *data* 中指定. 如果执行成功, call 方法返回布尔值 true(如果在指定地址没有合约也返回 true); 如果执行失败 (例如被调用函数抛出异常), 则返回 false.

(2) *contract*.*function*(*arg*1,*arg*2,…): 高级调用. 可能在成功时返回一个值作为输出. 如果调用的方法失败 (或不存在), 调用方会引发异常, 调用方做出的所有更改都会被回退, 且异常会自动在调用层次结构中传播.

如果为 call 指定的函数不存在, 则调用被调用方的回退函数. 回退函数没有名称和参数, 且不能返回任何内容. 一个合约最多可以有一个回退函数, 如果没有, 则不会执行任何函数, 这也并不会构成失败. 如果以太币被发送到合约, 则回退函数也会被调用, 方法如下.

(1) *addressOfContract*.send(*amount*): 如果回退函数存在, 发送指定金额的货币到合约调用. 如果 send 失败 (例如回退函数抛出异常), 则返回布尔值 false; 否则返回 true.

(2) *addressOfContract*.transfer(*amount*): 类似于 send, 但在失败时引发异常, 处理方式与高级函数调用失败类似.

最后, 一个合约还可以使用 *addressOfContract*.delegatecall(*data*) 函数委托执行给另一个合约. 委托类似于低级调用, 区别在于委托时由 data 指定的函数是在调用方的上下文中执行的 (例如函数将查看调用方的状态变量而非被调用方). 即合约可以使用 delegatecall 从其他合约借用代码, 这是创建库的基础.

B.2 常见 Solidity 漏洞的举例

本文主要讨论 3 种常见的 Solidity 智能合约漏洞类型.

(1) 重入攻击. 当一个合约在调用另一个合约中的函数时, 调用方会被阻止从而进入临时的中间状态并等待调用返回. 如果被调用的合约包含恶意代码, 它可以在返回之前再次调用原合约的关键函数, 这使得调用方处于未完成的状态时被多次调用, 从而造成不一致的状态或资金的意外转移. 重入攻击是一种常见的智能合约漏洞, 曾在著名的 The DAO 攻击事件[39]中造成了严重的资金损失.

(2) 拒绝服务. 如果一个函数涉及使用 transfer 发送以太币或进行高级函数调用到另一个合约, 那么接收方合约可以通过始终抛出异常来阻止该函数执行. 文中模型通过活性属性来检测此类漏洞, 在第 5.1.3 节属性验证的“以太坊之王 2”类型中有所展示.

(3) 死锁. 一个合约可能会因意外或通过对抗性行为陷入死锁状态, 此时再也无法从合约中提取或转移货币. 这意味着合约中的货币实际上已经丢失, 类似于发生在 Parity 多签钱包合约的情况[40]. 本模型可以自动验证合同模型是否无死锁, 不需要指定任何属性.

李任翔(2001－), 男, 硕士生, 主要研究领域为自然语言处理, 智能合约.

蒋忠元(1988－), 男, 博士, 教授, 博士生导师, 主要研究领域为复杂 (无人) 网络系统安全, 大数据与 AI 安全, 空天地一体化网络技术, 法律自然语言处理.

高胜(1987－), 男, 博士, 教授, 博士生导师, 主要研究领域为区块链, 智能合约, 联邦计算, 大模型安全.

钱肖(2000－), 男, 硕士, 主要研究领域为自然语言处理, 深度学习.

沈秀轩(1998－), 女, 博士生, 主要研究领域为文本匹配, 多任务学习.

刘柄呈(2001－), 男, 硕士生, 主要研究领域为信息安全, 隐私保护.

陶梅悦(2001－), 女, 硕士生, 主要研究领域为信息抽取, 深度学习.

马建峰(1963－), 男, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为应用密码学, 无线网络安全, 数据安全, 移动智能系统安全.