软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(8):2875−2895 [doi: 10.13328/j.cnki.jos.006603]

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

智能合约的时间约束模式及其形式化验证*

赵颖琪[1,2], 朱雪阳[1,2], 李广元[1,2], 包玉龙[1]

[1](计算机科学国家重点实验室(中国科学院 软件研究所), 北京 100190)

[2](中国科学院大学, 北京 100049)

通信作者: 朱雪阳, E-mail: zxy@ios.ac.cn

摘 要: 智能合约是一套以数字形式定义的承诺. 通过智能合约, 可以大大减少协议制定的中间环节, 提高协议制定的效率. 区块链技术为智能合约的执行提供了可信平台. 随着区块链应用的拓广与深入, 智能合约的作用必然越来越突出, 智能合约的可靠性问题也将更加突显. 以提高智能合约可靠性为目的的研究日益得到重视, 但尚未有人对智能合约的时间性质可能引起的可靠性问题进行过系统的研究. 通过分析典型智能合约, 对智能合约时间约束的不同表现形式进行梳理, 提炼出相应的时间约束模式并对其进行形式化; 定义 Solidity 智能合约到时间自动机的转换规则, 并实现其到实时模型检测工具UPPAAL入口模型的自动转换; 再利用UPPAAL验证合约的时间相关性质. 最后对两个实际合约进行实例研究, 结果表明了所提炼模式的普遍性以及所提出形式化验证方案的可行性和有效性.

关键词: 智能合约; 时间约束模式; 模型检测; Solidity; 形式化方法

中图法分类号: TP311

中文引用格式: 赵颖琪, 朱雪阳, 李广元, 包玉龙. 智能合约的时间约束模式及其形式化验证. 软件学报, 2022, 33(8): 2875–2895. http://www.jos.org.cn/1000-9825/6603.htm

英文引用格式: Zhao YQ, Zhu XY, Li GY, Bao YL. Time Constraint Patterns of Smart Contracts and Their Formal Verification. Ruan Jian Xue Bao/Journal of Software, 2022, 33(8): 2875–2895 (in Chinese). http://www.jos.org.cn/1000-9825/6603.htm

Time Constraint Patterns of Smart Contracts and Their Formal Verification

ZHAO Ying-Qi[1,2], ZHU Xue-Yang[1,2], LI Guang-Yuan[1,2], BAO Yu-Long[1]

[1](State Key Laboratory of Computer Science (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Smart contract consists of a set of commitments defined in digital form. Smart contracts can greatly reduce the intermediate links in agreement formulation and improve the efficiency of agreement formulation. Blockchain technology provides a trusted platform for the execution of smart contracts. As the application of blockchain technology expands and deepens, the role of smart contracts will become more and more important, and the potential reliability problems, however, may cause huge lose to participants. The reliability of smart contracts has received more and more attention, but there is no systematic research on problems caused by the time properties of smart contracts. This study analyzes typical smart contracts, sorts out the different manifestations of time constraints, summarizes several time constraint patterns of smart contracts and formalizes them; defines transform rules from Solidity smart contracts to the timed automata. The translation from smart contracts to the model of model checker UPPAAL is then implemented and UPPAAL is used to verify the time properties of smart contracts. Case study is carried out on two practical smart contracts. The results show that the patterns extracted are general and the formal verification solution proposed is feasible and efficient.

Key words: smart contract; time constrained mode; model checking; Solidity; formal method

* 基金项目: 国家自然科学基金(62072443)

本文由“形式化方法与应用”专题特约编辑陈立前副教授、孙猛教授推荐.

收稿时间: 2021-09-05; 修改时间: 2021-10-14; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

近些年来, 比特币的出现, 引起人们对区块链技术的关注[1]; 而智能合约的引入, 又使得区块链技术更加生机勃勃[2]. 区块链技术具有去中心化、去信任、透明性、集体维护、不可篡改等特性, 为智能合约提供了安全可靠的记录载体和执行环境. 有了智能合约, 开发人员能够在区块链上建立和发布各种分布式应用, 为区块链技术的应用提供了无限的可能.

智能合约是一套以数字形式定义的承诺[3]. 承诺是指合约参与方同意的权利和义务, 是参与各方需共同遵守的协议. 早在 1994 年, 美国计算机科学家尼克·萨博(Nick Szabo)就提出了“智能合约”的概念, 但当时的经济和通信基础设施不足以支持它的实际应用. 自以太坊[4]将智能合约引入区块链后, 智能合约的应用已渗透到诸多领域中. 在金融方面, 在没有中间商的情况下, 利用智能合约进行公寓、停车场、自行车等的出租及销售[5]. 在法律层面, 用智能合约实现法律合同[1], 将书面化的法律语言转化为能够自动化执行的合约. 在医疗数据共享领域中, 智能合约可以规范数据的产生者(病人)、数据的保管者(医院)、数据的使用者(数据分析公司)三方的权责[6], 从而促进医疗大数据的共享.

由于智能合约在复杂的网络环境中运行并承载着巨大的经济利益, 它的可靠性至关重要[7]. 已有多次由于合约代码漏洞引发了安全事故. 例如, 在著名的“The DAO”事件中, 攻击者利用智能合约的可重入漏洞, 窃取了当时价值约为五千万美元的 360 万个以太币[8]. 由于区块链的匿名性, 攻击者仍逍遥法外. 这些问题的暴露, 使人们意识到在部署之前对合约的潜在安全问题进行检测的必要性. 相关研究正如火如荼地展开[2,9−11]. 但我们尚未发现有人对智能合约的时间性质可能引起的可靠性问题进行过系统的研究.

承诺往往具有时效性, 即时间约束. 例如, 在航班延误保险合约[12]中, 系统向被保险人承诺: 当被保险人购买了所搭乘航班的保险后, 若搭乘航班延误且延误时间达到保险单所载明的时间, 则乘客获得相应的保险金. 按照合约规定, 乘客需在规定时间内购买保险方可理赔. 如果在合约中关于这方面的时间约束书写有误, 例如合约未检测购买保险时间, 那么攻击者可在得知飞机延误后再去购买保险, 造成保险公司的损失. 在竞拍合约[13]中, 竞拍发起者规定了竞拍期限, 承诺在期限内处理竞拍人的投标, 期限外不予处理. 如果没有合适的时间限定, 如果合约受益人取出收益和竞拍者投标两个交易同时进行, 可能导致受益人不能获得实际最高的出价额(具体讨论见第 4.1 节). 由此可见: 如果不事先对合约的时间相关性质进行检测, 由于区块链的不可篡改特性, 部署后一旦引发问题, 可能产生不可逆转的损失. 因此, 对智能合约的时间约束行为进行验证, 是保障合约可靠性的重要问题之一.

本文基于实时模型检测工具 UPPAAL[14], 研究 Solidity[4]智能合约是否满足时间约束的验证问题. 通过对典型案例的研究, 总结出 5 种智能合约的时间约束模式, 并实现对这几种模式的半自动化验证. 将带有时间约束的合约文件转换为时间自动机模型(UPPAAL 的入口模型), 将待验证的性质用分支时序逻辑公式(UPPAAL 的性质描述语言)描述, 再利用 UPPAAL 进行验证合约是否满足给定性质, 以便及时发现合约可能存在的与时间相关的问题, 进而为合约的修改提供支持. 验证框架如图 1 所示. 据我们所知, 本文是首个对智能合约的时间约束行为进行系统化研究的工作.

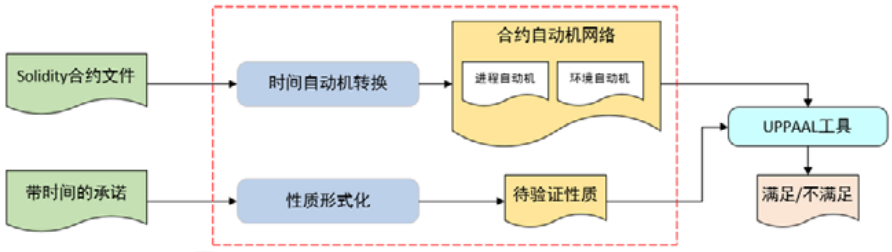


图 1 验证框架图

本文的主要贡献包括: 通过分析典型智能合约, 对智能合约时间约束的不同表现形式进行梳理, 提炼出相应的时间约束模式并对其进行形式化; 定义 Solidity 智能合约到时间自动机的转换规则, 并实现其到实时模型检测工具 UPPAAL 入口模型的自动转换; 再利用 UPPAAL 验证合约的时间相关性质; 最后, 对两个实际合

约进行实例研究, 结果表明本文所提炼模式的普遍性以及本文所提出形式化验证方案的可行性和有效性.

本文第 1 节对相关工作进行介绍.第 2 节介绍 Solidity 智能合约及其时间约束以及验证工具 UPPAAL. 第 3 节–第 5 节是本文的主要工作, 其中, 第 3 节定义智能合约到时间自动机的转换规则, 第 4 节总结 5 种时间约束模式并对其进行形式化, 第 5 节展示实例研究及结果. 第 6 节进行总结与展望.

1 相关工作

在智能合约的安全漏洞检测方面, 较多研究是基于静态分析技术, 主要工具有 Oyente[15], Manticore[16,17], Mythril[18,19], Securify[20]等. 静态方法通常是利用已总结好的安全漏洞模式, 无法验证合约的可靠性问题. 这方面, 形式化验证可以很好地补充. 形式化验证方法大致分为两种: 定理证明和模型检测. 基于定理证明的方法通常将合约转化为工具语言, 利用定理证明器去验证智能合约[21−26]. 基于定理证明的方法在验证过程中需要人工介入, 对使用者的专业知识要求比较高. 而基于模型检测方法的验证过程可以全自动地执行. 本文方法基于模型检测技术, 下面介绍利用模型检测工具验证智能合约的相关工作.

Bai 等人在文献[26]中提出使用合约状态机表示智能合约, 将整个合约按照规范形式化建模为非确定性自动机, 将需要验证的性质表示为线性时态逻辑, 最后利用工具 SPIN[27]验证智能合约模板和性质的正确性. Nehai 等人以区块链能源市场合约为例, 提出了 Ethereum 智能合约的通用建模方法, 模型采用 NuSMV[28]输入语言编写, 将检验的性质形式化为时序逻辑公式, 利用 NuSMV 工具验证模型是否满足待检验的性质, 如果合约模型不满足性质给出反例. 此外, Abdellatif 和 Brousmiche[29]使用 BIP(behavior interaction priorities)框架为智能合约、区块链、用户、矿工、交易池和攻击者建模, 最后使用统计模型检测方式来验证模型是否满足某个性质. Kalra 等人[30]研制了工具 ZEUS, 构建 Solidity 到 LLVM 位码转换器, 自动插入验证条件给定的策略规范, 可以验证特定的安全性质. Permenev 等人[31]研制了 VERX 工具, 结合可达性检查、有效的符号执行引擎和延迟抽象, 对一类合约性质进行验证. 这方面已有的工作主要是利用模型检测工具对合约的功能性和可靠性进行验证, 本文提出的方法将利用模型检测工具 UPPAAL 对智能合约的实时性质进行验证.

目前, 对智能合约时间相关性质的验证工作还很少. 有一些研究者针对智能合约的时序问题进行了研究. 如 Shishkin 等人[32]针对不带循环、递归和动态内存管理的简单 Solidity 语言子集, 提出了一种符号模型检测技术和一种形式化规范方法, 可以检测交易执行不当造成损失的问题, 使用 SMT 求解器作为后端工具来查找给定性质的反例. Zhu 等人[33]利用现有的命题投影时序逻辑 PPTL(propositional projection temporal logic)模型检测算法对区块链中是否具有真正的并发性进行了建模和验证. 除了在时序方面的工作, 也有人意识到合约时间性质会引发问题. 李书霞等人在文献[34]中介绍, 他们尝试利用 UPPAAL 对智能合约进行了手动化建模, 但是仅针对投票合约实例, 对合约的安全性以及时效性进行了验证.

目前的研究工作中, 尚未见到有人系统分析带有时间约束的智能合约, 仅有人尝试手动建模或针对一类案例对问题进行说明. 本文比较系统化地对智能合约的时间性质进行了研究, 通过对智能合约时间约束的不同表现形式进行梳理, 提炼出相应模式, 并利用时间自动机及分支时序逻辑公式进行形式化, 最后用实时模型检测工具 UPPAAL 进行验证.

2 准备知识

Solidity[4]是以太坊的合约编程语言. 本文工作以 Solidity 合约及其时间性质为验证对象, 以实时模型检测工具 UPPAAL 为后端支撑工具, 本节对相关内容做简要介绍.

2.1 Solidity智能合约及其时间约束

智能合约存在着多种定义[26,27,35], 它允许匿名的各方之间进行可信的交易, 而不需要中央权威机构、法律系统或外部执行机制. 从狭义上来说, 智能合约是一种涉及到商务逻辑和算法的程序代码, 可以将用户、法律和网络三方的复杂关系程序化. 从广义上来说, 智能合约是一种可自动执行的计算机协议[36].

Solidity 是以太坊的智能合约语言, 类似于 JavaScript. 合约代码被编译为 EVM 字节码并由区块链交易驱动, 在以区块链作为基础的虚拟机上运行, 在整个过程中都可能遇到不同的安全威胁和功能错误[10]. Solidity 智能合约的安全威胁及功能错误来自于 3 个层面: 语言、虚拟机、区块链. 语言层面主要是因合约编写语言的设计失误引入, 或者是合约开发者在代码设计时的失误造成. 虚拟机层面的威胁主要有两方面: 一是以太坊设计智能合约字节码的规范和运行机制存在一些缺陷, 二是以太坊客户端没有严格按照手册实现虚拟机功能[36]. 对于智能合约来说, 区块链是其安全可信任的根基, 但区块链本身很多特性也给智能合约带来了安全隐患[37]. 在智能合约的编写过程中, 设计人员不仅要设计执行过程, 还要考虑合约执行时与区块链平台的交互关系. 本文主要关注语言层面的设计所带来的安全问题. 以下以图 2 所示的竞拍合约[13]为例, 简要介绍 Solidity 智能合约.

```
1  pragma solidity ^0.7.0;
2  contract SimpleAuction {
3     address payable beneficiary;
4     uint auctionEnd;
5     address highestBidder;
6     uint highestBid;
7     …
8  constructor
9        (uint_biddingTime,address payable_beneficiary){
10       beneficiary =_beneficiary;
11       auctionEnd=block.timestamp+_biddingTime;
12    }
13    function bid(·) public payable {
14       require(block.timestamp≤auctionEnd);
15       require(msg.value>highestBid);
16       …
17       highestBidder=msg.sender;
18       highestBid=msg.value;
19    }
20    function withdraw(·) public returns (bool) {
21       …
22          if (!msg.sender.send(amount)) {
23             …
24          }
25       …
26    }
27    function AuctionEnd(·) public {
28       require(block.timestamp≥auctionEnd);
29       …
30       ended=true;
31       beneficiary.transfer(highestBid);
32    }
33 }
```

图 2 一个简单竞拍合约

合约设置了受益人(*beneficiary*)以及竞拍期限(*auctionEnd*), 用户可以在竞拍期限内发起竞拍(*bid*(·)), 如果竞拍金额高于当前最高价(*highestBid*), 则成为最高竞拍人(*highestBidder*), 用户也可以撤回自己竞拍资金(*withdraw*(·)). 只有过了竞拍期限, 才能关闭竞拍(*AuctionEnd*(·)), 并将最后的成交额(*highestBid*)交给受益人(*beneficiary*), 并将整个事务终止(ended=true).

构造函数 *constructor*(·)仅在创建合约期间(部署上链时)运行一次, 一般是对合约中的变量进行初始化, 也可为空. 本例中, 构造函数通过参数设置了当前合约受益人 *beneficiary*(第 10 行), 通过获取系统变量 *block.timestamp* 的值来得到当前区块的时间戳, 将合约期限 *auctionEnd* 设置为从当前时间起的*_biddingTime* 之后(第 11 行). 在 *bid*(·)函数中, *require*(·)检测当前时间是否在规定的时间期限内, 若不在则异常终止(第 14 行), 同时调用 *bid*(·)的交易也应该失败. 接下来, 竞拍函数检测当前竞拍人的出价是否更高(第 15 行), 满足条件后进行最高竞拍人和最高竞拍价的更新(第 17 行、第 18 行). *withdraw*(·)函数是竞拍者用来撤回自己竞拍资金的函数, 第 22 行中的 *msg.sender.send*(*amount*)是 Solidity 合约中用于转账的调用函数, *msg.sender* 为账户接受者, *amount* 为转账金额. 这里, *send*(·)是有返回值的, 如果转账失败返回 false. 除了 *send*(·)之外, 还有 *transfer*(·), *call.value*(·)也用于合约的转账, *call.value*(·)与 *send*(·)操作基本相同, 如第 31 行的 *transfer*(·)转账失败没有返回值, 会异常终止不执行后续语句. *AuctionEnd*(·)函数先判断时间是否到达预先设定的期限值(第 28 行), 如果已到期限检测就关闭整个交易(第 30 行), 合约受益人可以获取自己的收益(第 31 行).

区块链上的时间戳是保证其数字文件安全的核心. 时间戳是使用数字签名技术产生的数据, 签名的对象包括原始文件、签名参数、签名时间等信息. 区块时间戳是指当前的合约调用交易所属的区块被打包的时间戳. Solidity 中, 时间戳由系统变量 *block.timestamp* 来获取, 它是 Solidity 的预留关键字, 最小单元值为 1, 相当于 1s, 不需要在合约代码中声明. 在 Solidity 编译器早期的版本中, 变量 *now* 也起同样的作用, 但在 0.7.0 之后, *now* 被移除了. 在图 2 的合约中出现了时间期限设置(第 11 行)以及时间约束判断(第 14 行、第 28 行)语句, 这类操作是本文的主要关注点.

2.2 实时模型检测工具UPPAAL

UPPAAL 是由 Uppsala 大学和 Aalborg 大学联合开发的实时模型检测工具[38], 它适用于建模并验证带时间的系统及性质, 且已被成功用于验证实时控制器、通信协议等实时系统中[39].

UPPAAL 的建模语言为时间自动机网络, 性质描述语言为分支时序逻辑语言 CTL[40]. 时间自动机网络是多个时间自动机的并发. 时间自动机在有限状态自动机的基础之上扩充了取实数值的时钟(clock)变量, 用来表示系统中的时间约束. UPPAAL 中 CTL 公式形式的语法如下:

$$Formula::=A[]p|A<>p|E[]p|E<>p|p\rightarrow q.$$

其中, A 和 E 为路径算子, A 表示所有路径, E 表示存在某个路径; []和<>为时序算子, []表示路径上的所有状态, <>表示未来某个状态; p, q 为表达式(p, q 可含有形如 $x\sim d$ 的时间约束, 这里, x 是时钟变量, d 是非负整数, ~是比较符). 例如: 公式 $A[]p$ 表示在系统状态树的所有路径上, 每个状态都满足性质 p; 公式 $A<>p$ 表示在系统状态树的所有路径上, 都存在某个状态满足性质 p; 公式 $E[]p$ 表示系统状态树中存在一条路径, 该路径上所有状态都满足 p; 公式 $E<>p$ 表示系统状态树中至少存在一条路径, 该路径上存在某个状态满足 p; 需要注意的是, UPPAAL 中, imply 是逻辑蕴含, 而→为时序算子 lead to, 公式 $p\rightarrow q$ 表示一旦 p 满足, 将来某个时刻 q 一定满足, 也即时序公式 $A[]$ (p imply $A<>q$)的缩写.

时间自动机由一组结点和一组边组成: 结点表示状态, 边表示状态转移. 每条边包含 3 个属性: 条件(guard)、状态更新操作(update)和同步操作(sync). 同步操作由信号发送和信号接收实现, 信号经由通道传递, 用于自动机之间的通信. 当边上条件被满足且所需接收的信号达到时, 这条边所表示的由一个状态到另一个状态的转移才可能发生. 以下以如图 3 所示的台灯系统为例介绍. 该系统是由图 3 左边的台灯自动机和右边的开关自动机组成的自动机网络. 台灯自动机有 3 个状态: 关闭(off)、弱亮(soft)和亮(bright), 其中, off 为初始状态. 两个本地时钟 x 和 y; 开关自动机仅有一个 idle 状态和一条边, 表示开关按键被按下(press!). 台灯在 off 状态, 收到开关被按下信号(press?)后变 soft 状态, 并将时间 x 重置(x=0); 接着, 如果在 3 个时间单位内($x\leqslant 3$)收到开关被按下信号(press?), 对时钟 x 重置(x=0), 变为 bright 状态, 若在 3 个时间单位之后($x>3$)收到开关被按下信号(press?), 则变回 off 状态; 在 bright 状态下, 最多维持 100 个时间单位(不变式 $x\leqslant 100$). 开关自动机没有时钟, 任意时刻都可能发送 press 信号, 用于模拟外界对台灯的使用行为.

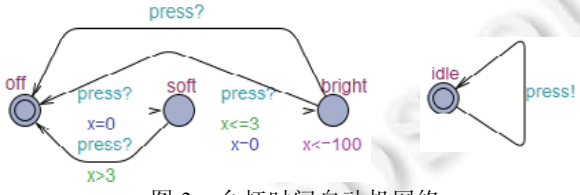


图 3 台灯时间自动机网络

对于节能台灯, 我们希望在长时间不使用时, 它总能回到关闭(off)状态. 这条性质用 CTL 公式表示为

$$A<>lamp.off,$$

其中, lamp 为表示台灯的时间自动机. 下文中, 我们将等价地使用 UPPAAL 模型和时间自动机网络.

3 智能合约形式化

为验证智能合约行为是否满足所要求的时间约束, 首先需要将待验证 Solidity 智能合约转换为 UPPAAL 时间自动机模型. 本文考虑的 Solidity 子集如图 4 所示, 其中, *send*/*transfer*/*call.value* 是 Solidity 智能合约进行转账操作的系统函数.

type::=address|bool|uint|mapping
operator::= +|–|*|/|++|––|+=|–=|*=|/=
logicoperator::=|| | &&|>|<|≥|≤|!=|==|!

statement::=assignment|condition statement|
for statement|while statement|do while statement|
return|*require*|*send*|*transfer*|*call.value*

图 4 工具支持的 Solidity 语言子集

3.1 合约自动机网络

我们将整个合约抽象为一个时间自动机网络, 包含两类自动机: 每一个函数对应一个时间自动机, 用于描述函数的行为, 称为进程自动机; 用一个时间自动机模拟外部用户行为, 称为环境自动机. 待验证的智能合约是由全局变量、环境自动机及所有函数的进程自动机组成的时间自动机网络, 称为合约自动机网络, 如图 5 所示. 外部对合约发起的调用、合约函数之间的调用分别由环境自动机与进程自动机之间、进程自动机互相之间的同步操作实现. 本节介绍合约的环境自动机和进程自动机如何定义, 全局时钟如何定义将在第 3.2 节介绍. 我们在全局变量部分增加了辅助变量 *glb_balance* 数组表示参与用户的账户余额, *contract_ balance* 表示合约余额, 其他全局变量一般为合约原有的全局变量.

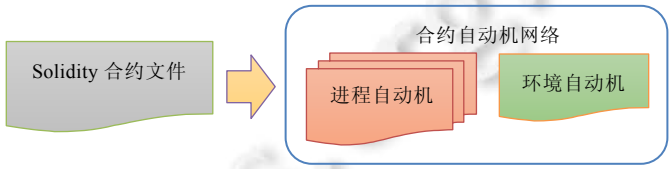


图 5 合约自动机网络

3.1.1 环境自动机

合约的执行由外界触发, 我们定义环境自动机来模拟外部用户行为. 用户的调用行为即交易的最初发起事件. 合约中构造函数仅会在部署合约时执行一次, 其他可供外界调用的函数可任意组合方式打包成交易. 图 6 为环境自动机示意图, 其中, 红色框内为环境自动机, 两个绿色框内为进程自动机. 通道 fun1Name_fun (fun2Name_fun)用来实现环境自动机对第 1 个(第 2 个)进程自动机的调用, 通道 fun_return 用来表示一个进程的结束(即函数执行结束). 区块链上交易的执行特点(交易之间顺序执行)所有进程共用一个进程结束通道 fun_return.

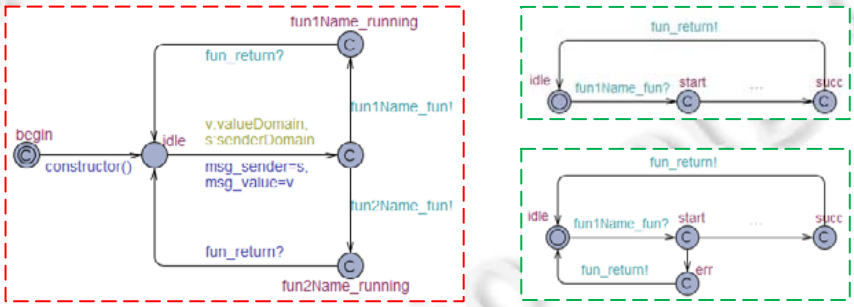


图 6 环境自动机示意图

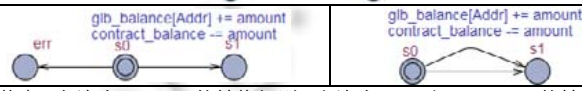
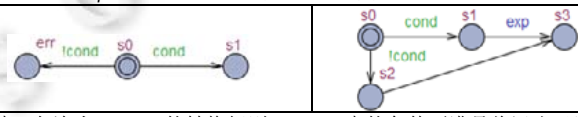
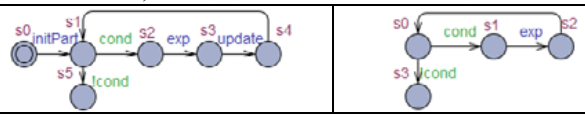
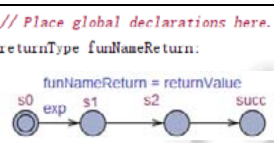
由于模型检测技术只能处理有限系统, 我们设置了有限域 *senderDomain* 存放所有调用者, 设置了有限域 *valueDomain* 存放可能的交易值. 合约的构造函数(*constructor*)仅当合约部署到区块链上时执行一次, 一般仅对合约中的变量进行初始化. 我们用 begin 到 idle 的一条边模拟合约部署到链上的行为. 合约部署后, 环境自动机处于 idle 状态, 表示合约上链后等待调用. 接着, 随机选取当前调用者 *msg.sender*(调用者地址)和 *msg.value*(调用者携带的以太币个数)来模拟任意用户携带任意个以太币, 然后不确定地发送表示调用函数的同步控制信号来模拟用户的可能调用行为. 环境自动机发送对应的同步控制信号(如 fun1Name_fun!, fun2Name_fun!), 对应的进程自动机接收到信号(如 fun1Name_fun?, fun2Name_fun?)后执行从 idle 到 succ(或 err)再到 idle 的路径, 结束后发送函数执行结束信号(fun_return!), 环境自动机接收到信号(fun_return?)后回到 idle 完成一次合约调用.

3.1.2 进程自动机

针对如图 4 所示的 Solidity 语言子集, 本文提出了如表 1 所示的转换规则, 实现合约函数到进程自动机的

自动转换. 其中, *updateExp* 表示一个或多个赋值语句, *Addr* 表示转账对象的账户地址, *amount* 表示转账金额, *cond* 为布尔条件, exp 表示任意一组表达式. 此外, 用户账户保存在 *glb_balance* 数组中, 合约账户地址为 *contract_balance*. 每个进程自动机初始时处于 idle 状态, 接收到调用信号后到达 start 状态, 表示函数开始执行. 若函数正常执行到结束, 经由 succ 状态回到 idle 状态等待下次调用; 若异常退出, 则经由 err 状态返回 idle 状态.

表 1 Solidity 到 UPPAAL 模型的转换规则

序号	Solidity 语句	UPPAAL 对应成分
1	*updateExp*; //assignment	
2	*Addr.transfer*(*amount*); *Addr.send*(*amount*); *Addr.call.value*(*amount*);	 其中, 左边为 *transfer* 的转换规则, 右边为 *send* 和 *call.value* 的转换. *transfer* 异常转转账后将退出函数执行到达 err 结点
3	*require*(*cond*); **if** (*cond*){ exp } 其中, exp 为一组任意表达式的集合	 其中, 左边为 *require* 的转换规则, *require* 中的条件不满足将退出函数执行到达 err 结点, 而 if 条件最终会汇合 s3 结点再继续分析以后的语句
4	**for** (*initPart*;*cond*;*updateExp*) { exp; } **while** (*cond*){ exp } 其中, *initPart* 为循环初始化部分, exp 为一组任意表达式的集合	 其中, 左边为 for 循环转换规则, 需要执行一次初始化, 而 while 循环直接到判定条件
5	function *funName*(·) public **returns** (*returnType*) { exp; **return** *returnValue*; } 其中, *returnType* 为返回值的类型要求, *returnValue* 为返回值的数值	 其中, 全局变量部分声明全局返回值 *funNameReturn*, 函数返回之前要对全局返回值赋值, 到达 succ 结点

根据上述转换规则, 现以图 2 所示竞拍合约的 *withdraw*(·)函数为例介绍进程自动机.

withdraw(·)函数的 Solidity 描述如图 7 所示, 其中: 变量 *pendingReturns* 用于存放每个用户可撤回的资金; *msg.sender.send* 是外部转账函数, 转账成功返回 true, 转账失败返回 false 值; if 语句判断其返回值, 如果转账失败, 恢复 *pendingReturns* 值. 该函数对应的 UPPAAL 进程自动机如图 8 所示. 在收到环境自动机或者其他进程发起的调用信号(withdraw_fun?)后, 从 idle 状态转换到 start 状态, 表示函数执行开始, 最后经由 succ 状态返回 idle. 该函数没有异常退出情况, 故无 err 状态. 其他各类语句的转换规则如下.

1. 合约中的条件判定语句(if/else 如图 7 中红色框)在自动机中用相应条件成立和不成立两条边描述(对应图 8 中红色箭头指向的两条边). 关键字 *require* 引导的语句也相当于条件判定语句, 不同的是, 条件不满足时异常结束到 err 结点(见表 1 规则 3);
2. 赋值语句对应于自动机边上的更新操作; 连续的赋值合并为一条边上的更新操作(见表 1 规则 1);
3. 转账操作(*transfer*/*send*/*call.value*, 如图 7 中绿色框)调用系统函数与外部账号或合约交互, 成功与否不确定. 这种情况由自动机中从同一状态转出的不带条件的边描述(对应于图 8 中绿色箭头指向的两条边), 分别对应于转账成功和转账失败操作(见表 1 规则 2);
4. 返回语句是函数的正常执行结束点, 由转到 succ 状态的一条边描述, 在边上对返回值赋值. 图 7 中蓝色框中为 3 个 return 语句, 对应如图 8 中蓝色箭头指向的 3 条边(见表 1 规则 5);
5. 智能合约调用外部函数获取数据称为 oracle. oracle 作为一个数据传送者, 可以在以太坊的 DApps 与

Web APIs 之间提供可靠连接, 基于智能合约的 Dapp 应用可信地取得外部信息和数据. 比如在航班延误保险合约中, 不可避免地需要获取航班信息. 为了处理这种情况, 我们在合约中以“//@”开头的注释声明所调用 oracle 的可能返回值; 当需要使用返回值时, 利用 UPPAAL 的关键字 select 不确定地选择一个来模拟外界的不确定返回值.

此外, 图 8 中除了 idle 结点, 其余结点均为 committed 结点(在这类结点上没有时间流逝). 这是因为在区块链上, 每个区块只有一个时间戳, 而一个交易(函数的执行)只能存在于一个区块中, 可以看作在一个函数的执行过程中不存在时间的流逝.

```
function withdraw() public returns (bool) {
    uint amount = pendingReturns[msg.sender];
    if (amount > 0) {
        pendingReturns[msg.sender]=0;
        if (!msg.sender.send(amount)) {
            pendingReturns[msg.sender] = amount;
            return false;
        }
        return true;
    }
    return true;
}
```

图 7　竞拍合约中的 withdraw 函数

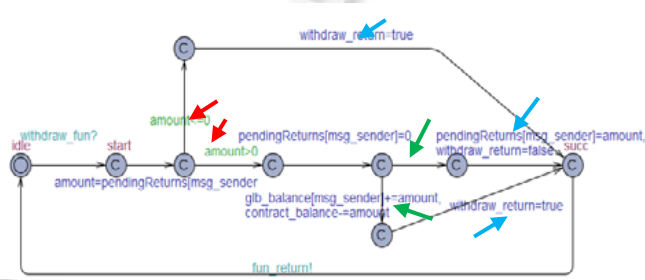


图 8　withdraw 函数对应的 UPPAAL 模板

3.2　时间相关操作

智能合约中与时间相关的操作可以分成两类: 时间期限设置和时间约束判断, 我们利用 UPPAAL 的时钟操作描述合约中的时间相关行为.

3.2.1　时间期限设置

智能合约在设置时间期限时采用的是绝对期限. 合约通过获取当前区块上的时间戳, 加上给定的相对期限(*rDL*)来设置绝对期限(*aDL*). 我们在 UPPAAL 模型中为合约中的每个 *aDL* 定义一个全局时钟 *aDL_clk* 及存放期限的变量 *aDL_v*, 将当前时钟值设为 0 后, 将 *aDL_v* 的值设为合约的相对期限 *rDL*. 具体的转换规则如表 2 所示, 左边是合约时间期限设置的一般表现形式, 右边是它在 UPPAAL 模型中的对应成分.

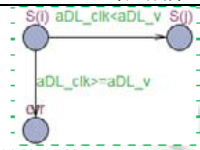
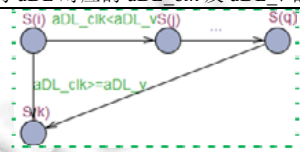
表 2　时间期限设置转换规则

Solidity 语句	UPPAAL 对应成分
function *A*(·){ … *aDL=block.timestamp+rDL*; … } 其中, *aDL* 是绝对期限, *rDL* 为给定的相对期限, *block.timestamp* 获取当前区块的时间戳.	// Place global declarations here. clock aDL_clk; int aDL_v; S(i) → S(j) aDL_clk=0,aDL_v=rDL 对应于合约中的每个期限(*aDL*), 定义一个全局时钟 *aDL_clk* 及存放期限的变量 *aDL_v*; 为期限设置语句定义一条转换边, 在其上将时钟重置, 并将期限 *aDL_v* 设为相对期限 *rDL*.

3.2.2　时间约束判断

对应合约中的时间约束判断语句, 在自动机中考察 *aDL* 对应的时钟 *aDL_clk* 及期限变量 *aDL_v* 的比较. 与其他变量的判断语句相同, 由带条件的两条边表示. *require* 语句与 if 语句有所不同, 转换规则见表 3. *require* 语句不满足时为异常退出, 而 if 语句各分支都为正常执行.

表 3 时间约束判断转换规则

Solidity 语句	UPPAAL 对应成分
function *A*(·){ … *require*(*block.timestamp*<*aDL*); … } 其中, *aDL* 为在其他函数中设置的期限.	 *require* 语句转换与第 3.1.2 节相同, 两条边上的条件为 *aDL* 对应的 *aDL_clk* 及 *aDL_v* 的比较.
function *A*(·){ if (*block.timestamp*<*aDL*) { … } } 其中, *aDL* 为在其他函数中设置的期限.	 if 语句转换与第 3.1.2 节相同, 两条边上的条件为 *aDL* 对应的 *aDL_clk* 及 *aDL_v* 的比较.

4 时间约束模式及其形式化

通过研究 Solidity 合约典型实例, 我们总结出 5 种时间约束模式: 单一时间期限、分段时间期限、时间期限延长、相对时间期限和多重时间期限. 本节从模式描述、模式在合约中的一般表现形式、模式实例几个方面对每种模式及其形式化进行详细介绍.

4.1 模式1: 单一时间期限

- 模式描述

合约设置了一个期限(*aDL*), 承诺在此期限内完成某些任务, 期限外拒绝服务.

模式在合约中的一般表现形式如图 9 所示.

```
function A(..) {
   …
   aDL=block.timestamp+rDL;
   …
}
```

```
function B(..) {
   …
   require(block.timestamp≤aDL);
   //if (block.timestamp≤aDL) {…}
   …
}
```

图 9 单一时间期限模式

在函数 *A* 中, 通过获取当前区块上的时间戳后, 设置合约期限 *aDL*. 当函数 *B* 被执行, 获取时间戳来判断此刻是否超过期限.

这种模式的合约一般需要满足以下承诺.

1) 在规定期限内(*block.timestamp*≤*aDL*), 合约必须完成承诺的任务;

2) 在期限外(*block.timestamp*>*aDL*), 合约拒绝某些服务.

特别注意边界情况(*block.timestamp*=*aDL*), 合约应执行所承诺的任务.

- 模式实例及形式化

这种模式是合约中最为常见的模式, 下面以简单竞拍合约[13]为例介绍本模式的形式化, 具体代码见第 2.1 节. 合约代码中与时间相关内容及 UPPAAL 模型中相关内容如图 10 所示. 在这个合约中, 合约期限内所有投标应能成功, 并且受益人应能获得最高投标额. 图 10 左边合约代码中, 红色字体为时间设置语句, 加粗字体为时间判断语句, 符合单一时间期限模式. 图 10 右边是两个带有时间判断函数对应的进程自动机. 橙黄色箭头指向为时间约束转换情况. 时间设置语句在构造函数中, 以函数方式作为环境自动机中边上的更新操作, 红框中对应时间设置语句.

```
1  pragma solidity ^0.7.0;
2  contract SimpleAuction {
3     …
4     constructor(_biddingTime,address payable_beneficiary) {
5        …
6        auctionEnd=block.timestamp+_biddingTime;
7     }
8        function bid(·){
9           require(block.timestamp≤auctionEnd);
10          …
11       }
12       function withdraw(·) {…}
13       function AuctionEnd(·) {
14          require(block.timestamp≥auctionEnd);
15          …
16          beneficiary.transfer(highestBid);
17       }
18 }
```

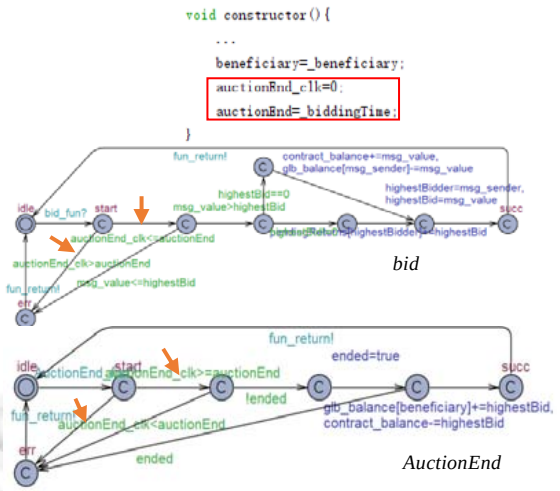


图 10　SimpleAuction 合约代码及其合约自动机网络

该合约所需满足的时间约束性质及其形式化如下(假设所有人的初始账户余额为 10).

(1) 在合约期限(*auctionEnd*)内, 若竞拍者携带的出价(*msg.value*)高于当前最高出价(*highestBid*), 一定可以竞标成功. 它的 CTL 公式表示如下:

$$auctionEnd\_clk \leqslant auctionEnd \;\&\&\; msg\_value > highestBid \;\&\&\; bid.start \rightarrow bid.succ.$$

其中, *bid.start* 表示 *bid* 函数已被执行, *bid.succ* 为竞拍成功到达的结点. 合约满足这条性质. 在这里, →为时序算子 lead to;

(2) 超过合约期限(*auctionEnd*)时, 竞拍事务不成功. 它的 CTL 公式表示如下:

$$auctionEnd\_clk > auctionEnd \;\&\&\; bid.start \rightarrow bid.err.$$

其中, *bid.err* 为竞拍函数未成功到达的最终结点. 合约满足这条性质;

(3) 在合约期限(*auctionEnd*)边界, 若有更高的投拍者, 投拍成功并且合约受益人拿到最高出价. 它的 CTL 公式表示如下:

$$auctionEnd\_clk == auctionEnd \;\&\&\; msg\_value > highestBid \;\&\&$$
$$bid.start \rightarrow glb\_balance[beneficiary] == 10 + highestBid.$$

其中, *glb_balance*[*beneficiary*]为受益人的账户, 起始为 10 且不参与竞拍. 合约不满足这条性质. 这是由于合约中 *bid*(·)和 *AuctionEnd*(·)都可以在时间等于 *auctionEnd* 这刻执行, 如果先执行 *bid*(·)再执行 *AuctionEnd*(·), 受益人将得到实际的最高投标额; 但如果 *AuctionEnd*(·)先执行, 将当前最高出价给了受益人并关闭合约, 再执行 *bid*(·)后, *highestBid* 增加, 但由于合约已关闭, 受益人得到是次高投票额而不是最高投标额. 这类错误是由交易顺序的不确定引起的, 模型检测所使用的全路径探索技术可准确地发现不满足要求的路径并给出反例.

4.2 模式2: 分段时间期限

- 模型描述

合约中设置了分段期限, 不同期限段内承诺的任务不同.

模式在合约中的一般表现形式如图 11 所示(以两个时间段为例).

在函数 *A* 中设置了两个期限 *aDL* 和 *aDL′*, 其中, *aDL′*>*aDL*. 在函数 *B* 执行时, 判断当前时间处于第 1 期限内、第 1 期限外但第 2 期限内、或第 2 期限外, 不同时间段内执行任务不同.

这种模式的合约一般需要满足以下承诺.

1) 在期限 *aDL* 范围内, 合约必须完成承诺的任务 *A*;

2) 在期限 *aDL* 外但 *aDL′*范围内, 合约必须完成承诺的任务 *B*;

3) 超过期限 *aDL′*, 合约拒绝服务.

特别注意: 在时间边界上, 合约应该只能有一个任务成功.

```
function A(..){
  …
  aDL=block.timestamp+rDL;
  aDL′=block.timestamp+rDL′;
  …
}
```

```
function B(..){
  …
  if (block.timestamp≤aDL) {…}
  else if (block.timestamp≤aDL′) {…}
  else {…}
  …
}
```

图 11 分段时间期限模式

- 模式实例及形式化

下面以食品溯源合约[41]为例, 介绍本模式的形式化. 为描述清晰起见, 我们将站点简化为两个: 起始点(*productAt*=1)和终点(*productAt*=2). 合约规定了应到达时间(*arrivedTime*)和一个最长配送期限(*delayTime*), 不同时间到达有不同措施. 配送员(*owner*)到达终点时汇报到达时间, 如果时间在应到达时间内到达, 将获得奖励(*driverProcess*=1); 如果超过应到达时间但是还在最长配送期限内到达, 则需要提醒配送员注意时间(*driverProcess*=2); 如果超过最长配送期限到达, 配送员将被惩罚(*driverProcess*=3). 合约代码和相应的进程自动机如图 12 所示. 自动机中橙黄色箭头对应合约中时间判断语句, 红框中对应时间设置语句.

```
1  pragma solidity ^0.7.25;
2  contract TrackingSystem {
3      …
4        constructor(…) {
5      …
6      arrivedTime=block.timestamp+3;
7      delayTime=block.timestamp+5;
8    }
9    function updateProductStage(·){
10     …
11       if (block.timestamp≤arrivedTime) {
12         driverProcess=1;
13       } else {
14         if (block.timestamp≤delayTime) {
15           driverProcess=2;
16         } else {driverProcess=3;}
17   }
18 }
```

```
void constructor(){
  arrivedTime_clk=0;
  arrivedTime=3;
  delayTime_clk=0;
  delayTime=5;
  productAt=1;
}
```

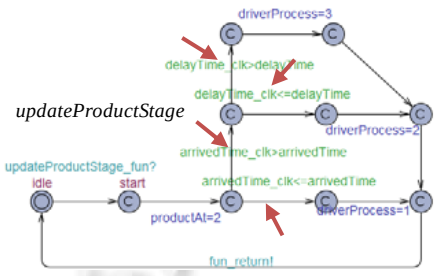


图 12 TrackingSystem 合约代码及其合约自动机网络

该合约所需满足的时间约束性质及其形式化如下.

1) 配送员从起始点(*productAt*=1)出发, 在应到达时间(*arrivedTime*)内到达目的地(*productAt*=2), 将得到奖励(*driverProcess*=1). 它的 CTL 公式表示如下:

$$arrivedTime\_clk \leqslant arrivedTime \ \&\& \ productAt==2 \ \&\& \ updateProductStage.start \rightarrow driverProcess==1.$$

其中, *updateProductStage.start* 表示到站后汇报情况;

2) 配送员从起始点(*productAt*=1)出发, 在超出应到达时间(*arrivedTime*)但是未超出最长配送期限(*delayTime*)到达目的地(*productAt*=2), 一定会被提醒(*driverProcess*=2). 它的 CTL 公式表示如下:

$$arrivedTime\_clk>arrivedTime \ \&\& \ delayTime\_clk \leqslant delayTime \ \&\& \ productAt==2 \ \&\& \ updateProductStage.start \rightarrow driverProcess==2.$$

3) 配送员从起始点(*productAt*=1)出发, 在超出最长配送期限(*delayTime*)到达目的地(*productAt*=2), 一定会被惩罚(*driverProcess*=3). 它的 CTL 公式表示如下:

$$delayTime\_clk>delayTime \ \&\& \ productAt==2 \ \&\& \ updateProductStage.start \rightarrow driverProcess==3.$$

这 3 条性质的验证结果均为满足.

4.3 模式3: 时间期限延长

- 模式描述

为吸引更多参与者, 在接近原定合约期限时延长期限. 模式在合约中的一般表现形式如图 13 所示.

```
function A(..){
   …
   aDL=block.timestamp+rDL;
   …
}
```

```
function B(..){
   …
   if (block.timestamp<aDL-t) {
        aDL=block.timestamp+extime;
      }
   …
}
```

图 13 时间期限延长模式

在函数 *A* 中设置原始期限 *aDL*. 在函数 *B* 执行时, 判断当时的时间戳是否接近原期限值(相差 *t* 单位时间), 如果接近, 要对合约期限 *aDL* 进行延长, 延长一个时间段(*extime*). 函数 *B* 的每次执行都有可能做此延长.

在这种模式的合约一般需要满足以下承诺.

1) 合约可终止;

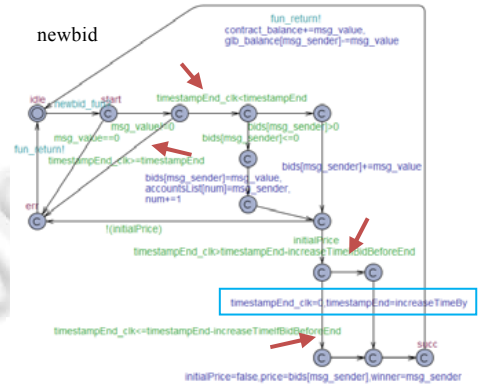
2) 合约终止后能完成规定任务.

- 模式实例及形式化

下面以另一个竞拍合约[42]为例, 介绍本模式的形式化. 为了吸引更多的人来竞拍, 合约规定在每一次竞拍都要判断时间是否接近合约期限(*timestampEnd*), 如果接近(相差时间段不超过 *increaseTimeIfBidBeforeEnd*), 就将期限延长(*increaseTimeBy*). 合约代码的简化版和相应的进程自动机如图 14 所示, 其中, 橙黄色箭头指向为时间的判断, 蓝框为时间期限延长.

```
1  pragma solidity≥0.4.23;
2  contract Auction {
3     …
4        constructor(…) {…
5           timestampEnd=block.timestamp+_timestampEnd;
6     }
7     function finalize(·){
8        require(block.timestamp>timestampEnd);
9        finalized=true;
10       beneficiary.send(price);
11       …
12    }
13    function refund(·){…}
14    function newbid(·){
15       require(block.timestamp<timestampEnd);
16       …
17       if (block.timestamp>timestampEnd
18          –increaseTimeIfBidBeforeEnd) {
19          timestampEnd=block.timestamp+increaseTimeBy;
20       }
21       …
22    }
23 }
```

```
void constructor(){
    timestampEnd_clk = 0;
    timestampEnd=_timestampEnd;
    num=0;
}
```



finalize

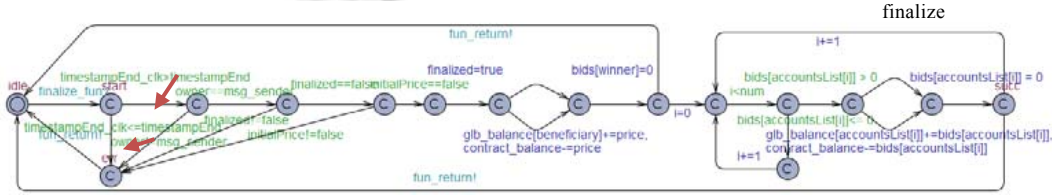


图 14 延长期限 Auction 合约代码及其合约自动机网络

该合约所需满足的时间约束性质及其形式化如下(假设所有人的初始账户余额为 10).

1) 合约可终止. 它的 CTL 公式表示如下:

$$A\diamond finalized==\text{true}.$$

合约不满足这条性质. 这是由于如果一直有人来竞拍, 就会不断延长期限导致无法终止交易, 那么合约受益人将无法获取收益;

2) 当合约终止后, 合约受益人可以拿到最高竞拍额(*price*):

$$finalized\rightarrow glb\_balance[beneficiary]==10+price.$$

其中, *glb_balance*[*beneficiary*]为受益人的账户, 起始为 10 且不参与竞拍. 合约不满足这条性质, 但不是由于时间操作不正确引起的, 而是由于合约中用 *send* 进行资金转账, 而未对其返回值进行检测; 当转账不成功时, 受益人无法拿到最高竞拍额.

4.4 模式4: 多重时间期限

- 模式描述

合约对不同用户设置不同期限.

模式在合约中的一般表现形式如图 15 所示.

```
function A(..){                          function B(..){
  …                                        …
  aDL[msg.sender]=block.timestamp+rDL[msg.sender];   require(block.timestamp<aDL[msg.sender]);
  …                                        …
}                                        }
```

图 15 多重时间期限模式

在函数 *A* 中, 对不同用户设置时间期限 *aDL*[*msg.sender*]. 函数 *B* 被执行时, 判断是否满足当前用户的时间期限.

这种模式的合约一般需要满足以下承诺.

1) 用户 *k* 在自己的期限(*aDL*[*k*])内请求服务, 合约必须完成承诺的任务;

2) 对于用户 *k* 在自己的期限(*aDL*[*k*])外的请求, 合约拒绝服务.

- 模式实例及形式化:

以下以时间锁合约[43]为例, 介绍本模式的形式化. 合约规定投资人投资后, 有一个释放资金期限(*releaseTime*), 期限过后才能取回资金. 合约代码的简化版和相应的进程自动机如图 16 所示. 这种情况需在 UPPAAL 模型中为每个投资人定义一个时钟, 用以独立跟踪用户操作时间. 图 16 中, 橙黄色箭头指向为时间的判断转换情况, 红框中对应时间设置语句.

```
1  pragma solidity≥0.4.23;
2  contract TimeLock {
3      …
4        constructor(·) {…}
5    function payIn(…) {
6      …
7      releaseTime[msg.sender]=
       block.timestamp+lockTimeS[msg.sender];
8    }
9    function withdraw(·) {
10     …
11     require(block.timestamp≥
       releaseTime[msg.sender]);
12   }
13 }
```

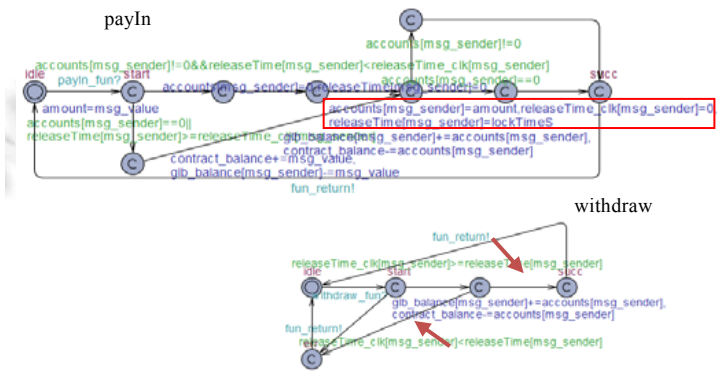


图 16 Timelock 合约代码及其合约自动机网络

该合约所需满足的时间约束性质及其形式化如下(假设所有投资人的初始账户余额为 10).

1) 投资人 *k* 投资后, 可在自己的释放期限(*releaseTime*[*k*])后将自己的资金撤回. 其 CTL 公式表示如下:

$msg\_sender==k \ \&\& \ releaseTime\_clk[k]>releaseTime[k] \ \&\& \ withdraw.start \rightarrow glb\_balance[k]==10$.

其中, *glb_balance* 中存放所有投资人的账户余额初始为 10, 所以撤资成功账户恢复. 合约不满足这条性质. 这是由于合约在 *withdraw* 函数中使用 *send* 外部函数且未检测返回值, 转账不成功时, 投资人无法撤回资金;

2) 未到自己的释放期限(*releaseTime*[*k*])时, 投资人 *k* 无法撤资. 它的 CTL 公式表示如下:

$msg\_sender==k \ \&\& \ releaseTime\_clk[k] \leqslant releaseTime[k] \ \&\& \ withdraw.start \rightarrow withdraw.err$.

合约满足这条性质.

4.5 模式5: 相对时间期限

- 模式描述

以合约期限内的某一时刻为起点, 设置一个相对期限, 承诺在此相对期限内完成某任务.

模式在合约中的一般表现形式如图 17 所示.

```
function A(..) {
  …
  aDL = block.timestamp+rDL;
  …
}
```

```
function B(..) {
  …
  require(block.timestamp≤aDL);
  aDL′ = block.timestamp+rDL′;
  …
}
```

```
function C(..) {
  …
  require(block.timestamp<aDL′);
  …
}
```

图 17 相对时间期限模式

在函数 *A* 中, 设置原始期限 *aDL*. 在函数 *B* 执行时, 判断当时的时间戳是否在合约期限内: 如果在, 设置一个从当前时刻开始的相对期限 *aDL′*. 在函数 *C* 执行时, 根据当前时间是否在相对期限 *aDL′* 内进行相应操作.

这种模式的合约一般需要满足以下承诺.

1) 如果在 *aDL* 期限内完成任务 *B*, 那么一定可以在其后的 *aDL′* 内完成任务 *C*;

2) 如果超过 *aDL* 还没完成 *B*, 那么合约拒绝服务.

- 模式实例及形式化

下面以一个购物合约[44]为例, 介绍本模式的形式化. 合约中设定了一个总的下单期限(*shoppingEnd*)以及每一个买家下单后对应的发货时间(*deliveryEnd*). 如果买家在下单期限内下单并付款, 商家(*owner*)应在发货时间内发货; 否则, 买家可以申请退款. 合约代码的简化版和相应的进程自动机如图 18 所示, 其中, 橙黄色箭头指向为时间的判断转换情况, 红框中对应时间设置语句.

该合约所需满足的时间约束性质及其形式化如下.

1) 如果在下单期限(*shoppingEnd*)内, 买家 *k* 成功下单后(*goodsOrder*[*k*]=true), 商家应该在规定发货时间(*deliveryTime*[*k*])内发货. 它的 CTL 公式表示如下:

$$shoppingEnd\_clk \leqslant shoppingEnd \ \&\& \ goodsOrder[k]==\text{true} \rightarrow$$
$$goodsDeliveryTime\_clk[k] \leqslant goodsDeliveryTime[k] \ \&\& \ deliveryOk[k]==\text{true}.$$

其中, *deliveryOk*[*k*]表示买家 *k* 下单后商品的发货状态, 为 true 表示商家已发货. 合约不满足这条性质. 这是因为商家何时发货不在合约范围内, 需要通过 oracle 获取, 对合约来说是不确定信息;

2) 如果买家 *k* 没有下在单期限(*shoppingEnd*)内下单成功, 商家不会为他发货. 它的 CTL 公式表示如下:

$$shoppingEnd\_clk>shoppingEnd \ \&\& \ goodsOrder[k]==\text{false} \rightarrow deliveryOk[k]==\text{false}.$$

这条性质结果为满足.

```
1  pragma solidity≥0.4.23;
2  contract shoppingApp {
3        …
4          constructor(…) {
5          shoppingEnd=block.timestamp+_shoppingTime;
6        …
7     }
8     function buyProduct(·) {
9        …
10       require(block.timestamp≤shoppingEnd);
11       deliveryTime[msg.sender]=block.timestamp+3;
12    }
13    function refundGoods {
14       …
15       require(block.timestamp>deliveryTime[msg.sender]);
16       …
17    }
18 }
```

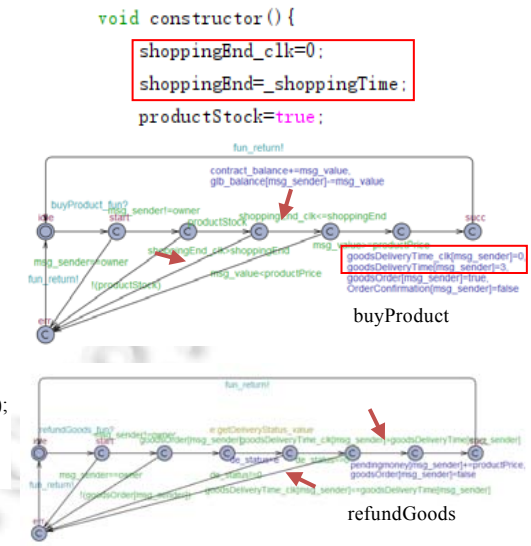


图 18 shoppingApp 合约代码及其合约自动机网络

5 实例研究及实验

我们实现了如图 4 所示的 Solidity 子集到 UPPAAL 时间自动机的自动化转换以及环境自动机的自动生成. 本节介绍两个实例研究结果, 有 3 个目的: 评估本文所提炼模式是否具有普遍性、评估本文所提出形式化验证方案的可行性以及评估验证效率.

实验的运行环境为: 32 核 CPU、主频为 2.90GHz、内存为 384G、Ubuntu 18.04 操作系统. 环境自动机中的两个参数 *senderDomain*(*msg.sender* 地址抽象的取值范围)和 *valueDomain*(*msg.value* 取值范围)取不同值, 以评估验证时间. 在环境自动机中, *msg.sender* 和 *msg.value* 分别不确定地从其中选取任意值. 由于模型检测是全空间探索技术, 验证时会检查所有可能取值的执行是否满足待验证性质.

本文相关实例及实验结果见 https://gitee.com/fmpa/dataset-for-mcVer-timed.

5.1 航班延误保险合约

航班延误保险合约[12]规定: 乘客购买机票之后的 30 分钟内需要购买保险, 对应的保险公司会预存相应的赔偿金; 如果保险公司没有按时预存赔偿金, 系统就直接将保费退还给用户; 如果保险公司预存了赔偿金, 乘客乘坐的航班没有延误, 保险公司可以获得保费并撤回自己预存的赔偿金; 如果航班延误, 保险公司可以获得保费, 但将保险公司预存的赔偿金赔偿给乘机人.

该合约中主要的变量见表 4, 函数功能见表 5.

表 4 FlightDelay 合约中的变量

变量名称	含义
DELAYED_PAYOUT	保险公司赔付额
processStatus[*k*]	乘客 *k* 对应购买机票的状态, 初始为 0, 购买机票后值为 1, 如果退票值为 2
InsuranceSet[*k*]	乘客 *k* 对应购买保险的状态, 初值为 false, 购买保险后值为 true
InsuranceLimit[*k*]	乘客 *k* 购买机票后设定的购买保险的时间期限
DepositLimit[*k*]	乘客 *k* 购买保险后, 给保险公司预存赔偿金设定的期限
DepositOk[*k*]	保险公司为乘客 *k* 预存赔偿金的状态, 初始为 false, 预存成功后为 true
caims[*k*]	乘客 *k* 所有退款金都会累计到这里
claimStatus[*k*]	乘客 *k* 获取保险赔偿金的状态, 当成功计算其应得的赔偿金后修改值为 true

表 5 FlightDelay 合约中的函数

函数名称	功能
buyTicket(·)	乘客 *k* 可以调用该函数购买机票, 并设定购买保险的期限 *InsuranceLimit*[*k*]
refundTicket(·)	乘客 *k* 调用该函数退机票
buyInsurance(·)	乘客 *k* 调用该函数购买保险, 并设定保险公司对该乘客购买保险后预存赔偿金时间期限 *DepositLimit*[*k*]
depositInsurance(·)	保险公司对每个用户购买保险后预存赔偿金
refundInsurance(·)	乘客 *k* 发现保险公司没有按时预存保费, 撤回自己的保险费
claimInsurance(·)	乘客 *k* 根据飞机的状态去申请赔偿
claimPayouts(·)	乘客 *k* 通过该函数撤回自己的退款

对应乘客购买保险时间期限 *InsuranceLimit*[*k*], 我们在 UPPAAL 模型中定义了时钟变量数组 *InsuranceLimit_clk*[*k*]及期限变量 *InsuranceLimit_v*[*k*]; 对应保险公司预存期限 *DepositLimit*[*k*], 我们定义了 *DepositLimit_clk*[*k*]及期限变量 *DepositLimit_v*[*k*], 其他语句根据第 3 节的相关规则转换.

该合约应满足的时间相关性质及其形式化如下.

1) 乘客 *k* 购买机票后(*processStatus*[*k*]=1), 在购买保险时间期限(*InsuranceLimit*[*k*])内能够成功购买保险(*InsuranceSet*[*k*]=true). 它的 CTL 公式表示如下:

$$msg\_sender==k \;\&\&\; processStatus[k]==1 \;\&\&\; InsuranceLimit\_clk[k]\leqslant InsuranceLimit\_v[k] \;\&\&\; buyInsurance.start\rightarrow InsuranceSet[k];$$

2) 如果乘客 *k* 在购买保险时间期限(*InsuranceLimit*[*k*])内购买了保险(*InsuranceSet*[*k*]=true), 保险公司能够在保险公司预存期限(*DepositLimit*[*k*])内成功预存赔偿金(*DepositOk*[*k*]=true). 它的 CTL 公式表示如下:

$$addr==k \;\&\&\; InsuranceLimit\_clk[k]\leqslant InsuranceLimit\_v[k] \;\&\&\; InsuranceSet[k] \;\&\&\; depositInsurance.start\rightarrow DepositLimit\_clk[k]\leqslant DepositLimit\_v[k] \;\&\&\; DepositOk[k].$$

其中, *depositInsurance.start* 表示保险公司预存保费的函数开始执行, *addr* 为 *depositInsurance* 的参数, 为对应乘客的地址. 合约不满足这条性质. 这是因为当合约检测到保险公司的资产不足时, 保险公司无法预存赔偿金;

3) 乘客 *k* 买票后(*processStatus*[*k*]=1), 在超过购买保险时间期限(*InsuranceLimit*[*k*])之后购买保险, 合约拒绝服务. 它的 CTL 公式表示如下:

$$msg\_sender==k \;\&\&\; processStatus[k]==1 \;\&\&\; InsuranceLimit\_clk[k]>InsuranceLimit\_v[k] \;\&\&\; buyInsurance.start\rightarrow buyInsurance.err.$$

其中, *buyInsurance.start* 表示购买保险的进程被调用了, *buyInsurance.err* 表示无法成功购买保险到达的结点;

4) 乘客 *k* 购买机票后退票(*processStatus*[*k*]=2), 则在购买保险时间期限(*InsuranceLimit*[*k*])内也无法购买保险. 它的 CTL 公式表示如下:

$$msg\_sender==k \;\&\&\; processStatus[k]==2 \;\&\&\; InsuranceLimit\_clk[k]\leqslant InsuranceLimit[k] \;\&\&\; buyInsurance.start\rightarrow buyInsurance.err;$$

5) 乘客 *k* 购买保险后, 保险公司未在预存期限(*DepositLimit*[*k*])内预存赔偿金(*DepositOk*[*k*]=false), 则乘客 *k* 申请保险金退款可以成功:

$$msg\_sender==k \;\&\&\; InsuranceSet[k] \;\&\&\; DepositLimit\_clk[k]>DepositLimit[k] \;\&\&\; !DepositOk[k] \;\&\&\; refundInsurance.start\rightarrow claims[k]==INSURANCE\_PRICE.$$

其中, *refundInsurance.start* 表示乘客申请保险金退款. *claims*[*k*]存放乘客 *k* 的赔付金额, 如果成功, 将存放合约中规定的保险费. 合约并不满足该性质. 这是由于合约的 *refundInsurance* 函数并未检测乘客是否已经申请过保险退款, 如果是恶意用户, 可以利用这个函数反复去申请退款, 造成保险公司损失;

6) 乘客 *k* 成功购买了保险(*InsuranceSet*[*k*]=true), 并且乘坐的航班超延迟到达(*claimInsurance.status*=4),

该乘客可按赔偿金额(*DELAYED_PAYOUT*)获得赔付. 它的 CTL 公式表示如下:

$$msg\_sender==k \;\&\&\; InsuranceSet[k] \;\&\&\; claimInsurance.status \;\&\&\&\&\; claimInsurance.start==4 \rightarrow claims[k]==DELAYED\_PAYOUT.$$

其中, 航班情况是通过 oracle 获取. 该返回值情况有 5 种: 1(未起飞), 2(航行中), 3(在规定到达时间内到达), 4(超时抵达)和 5(航班取消). 所以在这里, *claimInsurance.status*=4 表示当前航班延迟到达, *claims*[*k*]存放乘客 *k* 的赔付金额. 合约不满足这条性质. 原因与第 5)条相似, 恶意用户可以反复申请赔付, 使得 *claims*[*k*]累加起来, 造成保险公司亏损;

7) 乘客 *k* 成功购买了保险(*InsuranceSet*[*k*]=true), 并且乘坐的航班按时到达(*claimInsurance.status*=3), 不会获得赔偿. 它的 CTL 公式表示如下:

$$msg\_sender==k \;\&\&\; InsuranceSet[k] \;\&\&\; claimInsurance.status==3 \;\&\&\; claimInsurance.start \rightarrow claims[k]==0.$$

这个实例包含了相对时间期限和多重时间期限两种模式. 我们实验了 3 组不同的参数(*senderDomain* 和 *valueDomain* 的 3 种不同取值组合), 验证结果相同, 但时间随参与人数及取值范围的增加而增加. 合约的验证结果及所用时间见表 6.

表 6 航班延误保险合约实验结果 (m:分, s:秒)

性质	模式	验证结果	验证时间 *senderDomain*: [0,2] *valueDomain*: [1,1]	验证时间 *senderDomain*: [0,2] *valueDomain*: [1,3]	验证时间 *senderDomain*: [0,3] *valueDomain*: [1,4]
性质 1	4, 5	满足	8m9s	47m9s	81m9s
性质 2	4, 5	不满足	0.10s	0.44s	0.80s
性质 3	4, 5	满足	7m14s	42m2s	98m8s
性质 4	4	满足	7m1s	42m7s	88m22s
性质 5	4	不满足	9.06s	35.24s	5m16s
性质 6	4	不满足	1.08s	3.52s	1m6s
性质 7	4	满足	6m8s	37m8s	75m10s

5.2 购物合约

购物合约[44]中规定了优享时间、实惠购物时间、发货时间限制和商品下架时间, 购物者优享时间之前下单可享受 5 折优惠, 超过优享时间但早于实惠购物时间下单可享受 8 折优惠, 之后在商品下架前按原价购买; 当购物者下单后, 商家应该在发货时间限制内发货, 如果没有按时发货, 购物者有权申请退款, 但是在下单 3 天后才可以申请退款需要.

该合约中主要的变量见表 7, 函数功能见表 8.

表 7 Shopping 合约中的变量

变量名称	含义
productPrice	商品的定价
productNum	商品的库存
shoppingEnd	商家设定的总的购物时间期限
OptimalEnd	商家设定的最优惠(享 5 折)的购物时间期限
preferentialEnd	商家设定的普通优惠(享 8 折)的购物时间期限
productStock	控制整个合约是否可以来购物, 商家可以选择关闭修改其值为 false
goodsDeliveryTime[*k*]	用户 *k* 下单后, 商家进行发货的时间期限
deliveryStatus[*k*]	用户 *k* 下单后, 商家进行发货的状态, true 表示已发货
goodsOrder[*k*]	用户 *k* 下单状态, true 表示已成功下单
pendingmoney[*k*]	用户 *k* 发起退款后, 存放退款金额
claimStatus[*k*]	用户 *k* 发起退款成功的变量, 退款成功为 true

对应商品下架时间 *shoppingEnd*, 我们在 UPPAAL 模型中定义了时钟变量 *shoppingEnd_clk* 及期限变量 *shoppingEnd_v*; 对应优享时间 *OptimalEnd*, 定义了 *OptimalEnd_clk* 及期限变量 *OptimalEnd_v*; 对应实惠购物时间 *preferentialEnd*, 定义了时钟变量 *preferentialEnd_clk* 及期限变量 *preferentialEnd_v*; 为每个买家的发货时间 *goodsDeliveryTime*[*k*], 定义了时钟变量数组 *goodsDeliveryTime_clk*[*k*]及期限变量 *goodsDeliveryTime_v*[*k*]. 其他语句根据第 3 节的相关规则转换.

表 8 Shopping 合约中的函数

函数名称	功能
Close_trasition(·)	商家可以通过检测货物库存, 选择关闭购物交易
Close_all(·)	商家设定的购物时间, 到达交易自动关闭
buyProduct(·)	用户 *k* 调用该函数购买商品, 判定当前是不是在规定时间内购物, 如果在判定当前时间是在那个优惠期限内, 享受那种折扣金额, 下单后设定商家的发货最迟期限 *goodsDeliveryTime*[*k*]
refundGoods(·)	当商家未按时发货, 用户 *k* 调用该函数去申请退款
withdraw(·)	用户 *k* 调用该函数拿回自己的退款额

该合约应满足的时间相关性质及其形式化如下.

1) 买家 *k* 在商品下架时间(*shoppingEnd*)内购买商品(*goodsOrder*[*k*]=true)后, 商家一定要在规定时间(*goodsDeliveryTime*[*k*])内发货(*deliveryStatus*[*k*]==true). 它的 CTL 公式表示如下:

$$msg\_sender==k \;\&\&\; shoppingEnd\_clk \leqslant shoppingEnd\_v \;\&\&\; goodsOrder[k] \rightarrow goodsDeliveryTime\_clk[k] \leqslant goodsDeliveryTime\_v[k] \;\&\&\; deliveryStatus[k].$$

其中, *deliveryStatus* 需要获取 oralce 信息用来赋值发货情况. 合约不满足该性质. 这是由于外部 oracle 返回值不确定导致的;

2) 买家 *k* 在优享购物时间 (*OptimalEnd*)内下单,可以享受 5 折优惠.它的 CTL 公式表示如下:

$$msg\_sender==k \;\&\&\; OptimalEnd\_clk \leqslant OptimalEnd\_v \;\&\&\; buyProduct.start \rightarrow buyProduct.price==productPrice/2.$$

其中, *buyProduct.price* 为 *buyProduct* 进程中的变量, 表示实际购买价格;

3) 买家 *k* 在优享购物时间之后实惠折扣时间(*preferentialEnd*)之前下单, 可以享受 8 折优惠. 它的 CTL 公式表示如下:

$$msg\_sender==k \;\&\&\; OptimalEnd\_clk>OptimalEnd\_v \;\&\&\; preferentialEnd\_clk \leqslant preferentialEnd\_v \;\&\&\; buyProduct.start \rightarrow buyProduct.price==productPrice*8/10;$$

4) 在实惠折扣时间(*preferentialEnd*)之后, 买家 *k* 只能原价下单. 它的 CTL 公式表示如下:

$$msg\_sender==k \;\&\&\; preferentialEnd\_clk>preferentialEnd\_v \;\&\&\; shoppingEnd\_clk \leqslant shoppingEnd\_v \;\&\&\; buyProduct.start \rightarrow buyProduct.price==productPrice;$$

5) 买家 *k* 成功下单(*goodsOrder*[*k*]=true), 如果商家超过规定时间(*goodsDeliveryTime*[*k*])还未发货, 买家如果申请退款, 将取回购买金(*claimStatus*[*k*]=true). 它的 CTL 公式表示如下:

$$msg\_sender==k \;\&\&\; goodsOrder[k] \;\&\&\; goodsDeliveryTime\_clk[k]>goodsDeliveryTime\_v[k] \;\&\&\; !deliveryStatus[k] \;\&\&\; refundGoods.start \rightarrow claimStatus[k] \;\&\&\; pendingmoney[k]==OrderPrice[k].$$

其中, *goodsDeliveryTime*[*k*]为发货时间期限, 超过这个期限, 买家就可以退款. *pendingmoney*[*k*]存放 *k* 的退款金额, *OrderPrice*[*k*]为用户 *k* 下单的金额. 合约不满足该性质. 退款函数应检测是否已申请退款, 合约并未检测, 可以反复调用导致商家损失;

6) 在商品下架时间(*shoppingEnd*)之后, 合约拒绝任何买家 *k* 购买商品. 它的 CTL 公式表示如下:

$$msg\_sender==k \;\&\&\; shoppingEnd\_clk>shoppingEnd\_v \;\&\&\; buyProduct.start \rightarrow buyProduct.err.$$

在这个实例中包含了单一时间期限、多重时间期限和相对时间期限这 3 种模式. 3 组不同的参数的验证结果见表 9.

表 9 购物合约实验结果 (m:分, s:秒)

性质	模式	验证结果	验证时间 *senderDomain*: [0,2] *valueDomain*: [1,1]	验证时间 *senderDomain*: [0,2] *valueDomain*: [1,3]	验证时间 *senderDomain*: [0,3] *valueDomain*: [1,4]
性质 1	1, 4, 5	不满足	0.001s	0.005s	0.015s
性质 2	1, 2	满足	2m0s	8m3s	17m9s
性质 3	1, 2	满足	5m52s	14m4s	21m6s
性质 4	1, 2	满足	7m52s	12m3s	23m4s
性质 5	1, 4, 5	不满足	0.55s	0.83s	1.48s
性质 6	1	满足	4m2s	11m14s	18m5s

5.3 实验结果讨论

两个案例包含了多种时间约束模式. 从验证时间上看, 可满足性质的验证时间要长于不满足的性质. 这是由于在验证时只要找到一条不满足的路径即可报告不满足; 而对于满足的性质, 必须保证所有路径都被检测过并且没有发现不满足情况方可. 在实验结果中, 环境自动机需要模拟所有 *msg.sender* 地址抽象的取值(*senderDomain*)和 *msg.value* 的取值(*valueDomain*)组合. *msg.value* 取值范围不变, 每增加一个用户地址, 将增加一整组 *msg.value* 的取值. 用户数目不变, 扩大 *msg.value* 的取值范围, 相当于扩大了每个用户携带金额的取值范围. 除了用户个数及携带值范围大小对合约验证时间有影响外, 若合约包含 oracle 调用, 其返回值的范围对验证时间也会有一定影响.

6 总结与展望

本文通过分析典型智能合约, 总结出智能合约的 5 种时间约束模式, 并从模式描述、模式在合约中的一般表现形式、模式实例几个方面对每种模式及其形式化进行详细介绍; 定义了 Solidity 智能合约到时间自动机的转换规则, 并实现其到实时模型检测工具 UPPAAL 入口模型的自动转换; 利用 UPPAAL 验证合约的时间相关性质. 最后, 对航班延误保险合约和购物合约进行实例研究. 结果表明: 本文所提炼模式具有一定的普遍性, 本文所提出的形式化验证方案具有可行性和有效性. 针对智能合约的时间性质可能引起的安全问题, 本文提出的方案可以提前检测这类智能合约是否满足期望的时间约束性质, 从而避免部署后造成损失. 在今后的工作中, 将进一步总结每种模式性质的通用模板, 提高本文工作的自动化程度. 目前的版本尚不能给出 Solidity 层的反例, 我们将在下一步工作中, 研究从 UPPAAL 返回的反例中抽取 Solidity 反例.

References:

[1] 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).

[2] Zhu XY. Formal analysis of the DAO exploit . Information Technology and Network Security, 2021, 40(5): 13−19 (in Chinese with English abstract).

[3] Zhou Y. Research on promising blockchain smart Contract [MS. Thesis]. Shanghai: Shanghai Jiao Tong University, 2018 (in Chinese).

[4] Dannen C. Introducing Ethereum and Solidity. New York: Berkeley Press, 2017. 69−88.

[5] Liu DL. Research and application status, problems and suggestions of blockchain smart contract technology in the financial field. Hainan Finance, 2016(10): 27−31 (in Chinese with English abstract).

[6] Zhao YH, Yuan BH, Liang J. Application of blockchain technology in medical field. China Medical Education Technology, 2018, 32(1): 1−7 (in Chinese with English abstract). [doi: 10.13566/j.cnki.cmet.cn61-1317/g4.201801001]

[7] Cai YB. On the conformity of smart contract to the private law system. Oriental Law, 2019, 68(2): 68−81 (in Chinese with English abstract). [doi: 10.19404/j.cnki.dffx.20190304.002]

[8] Zhao YQ, Zhu XY, Li GY, *et al*. Verification of smart contracts with time constraints. Journal of Applied Sciences, 2021, 39(1): 1−16 (in Chinese with English abstract).

[9] Gao F. The difficult on fix vulnerabilities of blockchain smart contracts. Computer and Network, 2018, 44(12): 50−51 (in Chinese with English abstract).

[10] Ni YD, Zhang C, Yin TT. A survey of smart contract vulnerability research. Journal of Cyber Security, 2020, 5(3): 78−99 (in Chinese with English abstract).

[11] Qiu XX, Ma ZF, Xu MK. Ethereum smart contract security vulnerability scenario analysis. Information Security And Communications Privacy, 2019(2): 46−55 (in Chinese with English abstract).

[12] Eric. FlightDelay smart contract. 2018. https://github.com/causztic/archwing/blob/master/contracts/UserInfo.sol

[13] Ethereum, HiBlock. Auction. 2021. https://learnblockchain.cn/docs/solidity/solidity-by-example.html#index-1

[14] Behrmann G, David A, Larsen KG, *et al*. Uppaal 4.0. In: Proc. of the Int'l Conf. on the Quantitative Evaluaiton of Systems (QEST). IEEE Computer Society, 2016. 125−126.

[15] Luu L, Chu DH, Olickel H, *et al*. Making smart contracts smarter. In: Proc. of the 2016 ACM SIGSAC Conf. on Computer and Communications Security. 2016. 254−269.

[16] Maintainicore. https://github.com/trailofbits/manticore

[17] Mossberg M, Manzano F, Hennenfent E, *et al*. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2019. 1186−1189.

[18] Mythril. https://github.com/ConsenSys/mythril

[19] Mythril: Smashing Ethereum smart contracts for fun and real profit. 2018. https://github.com/b-mueller/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf

[20] Tsankov P, Dan A, Drachsler-Cohen D, *et al*. Securify: Practical security analysis of smart contracts. In: Proc. of the 2018 ACM SIGSAC Conf. on Computer and Communications Security. 2018. 67−82.

[21] Ahrendt W, Bubel R, Ellul J, *et al*. Verification of smart contract business logic. In: Proc. of the Int'l Conf. on Fundamentals of Software Engineering. Cham: Springer, 2019. 228−243.

[22] Alt L, Reitwiessner C. SMT-based verification of solidity smart contracts. In: Proc. of the Int'l Symp. on Leveraging Applications of Formal Methods. Cham: Springer, 2018. 376−388.

[23] Amani S, Bégel M, Bortin M, *et al*. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In: Proc. of the 7th ACM SIGPLAN Int'l Conf. on Certified Programs and Proofs. 2018. 66−77.

[24] Bhargavan K, Delignat-Lavaud A, Fournet C, *et al*. Short paper: Formal verification of smart contracts. In: Proc. of the 11th ACM Workshop on Programming Languages and Analysis for Security (PLAS), in conjunction with ACM CCS. 2016. 91−96.

[25] Ma FC, Fu Y, Ren M, *et al*. EVM*: From offline detection to online reinforcement for ethereum virtual machine. In: Proc. of the 26th IEEE Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2019. 554−558.

[26] Bai XM, Cheng ZJ, Duan ZB, *et al*. Formal modeling and verification of smart contracts. In: Proc. of the 7th Int'l Conf. on Software and Computer Applications. 2018. 322−326.

[27] Holzmann GJ. The model checker SPIN. IEEE Trans. on Software Engineering, 1997, 23(5): 279−298.

[28] Cimatti A, Clarke E, Giunchiglia F, *et al*. NUSMV: A new symbolic model checker. Int'l Journal on Software Tools for Technology Transfer, 2000, 2(4): 410−425.

[29] Abdellatif T, Brousmiche KL. Formal verification of smart contracts based on users and blockchain behaviors models. In: Proc. of the 9th IFIP Int'l Conf. on New Technologies, Mobility and Security (NTMS). IEEE, 2018. 1−5.

[30] Kalra S, Goel S, Dhawan M, *et al*. Zeus: Analyzing safety of smart contracts. In: Proc. of the NDSS. 2018. 1−12.

[31] Permenev A, Dimitrov D, Tsankov P, *et al*. Verx: Safety verification of smart contracts. In: Proc. of the IEEE Symp. on Security and Privacy (SP). 2020. 1661−1677.

[32] Shishkin E. Debugging smart contract's business logic using symbolic model checking. Programming and Computer Software, 2019, 45(8): 590−599.

[33] Zhu WJ. PPTL model checking for blockchains. In: Proc. of the 5th IEEE Information Technology and Mechatronics Engineering Conf. (ITOEC). IEEE, 2020. 792−795.

[34] Li SX, Wang GQ, Zhuang L. Reverse real-time model detection method for blockchain smart contract security. Journal of Chinese Computer Systems, 2020, 41(10): 2030−2035 (in Chinese with English abstract).

[35] Sun TY, Yu WS. A formal verification framework for security issues of blockchain smart contracts. Electronics, 2020, 9(2): 255.

[36] Ouyang LW, Wang S, Yuan Y, *et al*. Smart contracts: Architecture and research progresses. Acta Automatica Sinica, 2019, 45(3): 445−457 (in Chinese with English abstract).

[37] Wang PW, Yang HT, Meng J, *et al*. 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]

[38] Bengtsson J, Larsen K, Larsson F, *et al*. UPPAAL—A tool suite for automatic verification of real-time systems. In: Proc. of the Int'l Hybrid Systems Workshop. Berlin, Heidelberg: Springer, 1995. 232−243.

[39] Larsen KG, Pettersson P, Yi W. UPPAAL in a nutshell. Int'l Journal on Software Tools for Technology Transfer, 1997, 1(1−2): 134−152.

[40] Pnueli A. The temporal logic of programs. In: Proc. of the 18th Annual Symp. on Foundations of Computer Science (SFCS'77). IEEE, 1977. 46−57.

[41] TrackingSystem. 2019. https://github.com/Omprakash143/provenance_tracking_SmartContract/tree/master/provenance_tracking_smartContract

[42] Auction smart contract. 2018. https://github.com/astralship/eos/blob/master/contracts/Auction.sol

[43] TimeLock smart contract. 2016. https://github.com/SCBuergel/timeLock-smartContract/blob/master/TimeLock.sol

[44] Shopping smrt contract. 2018. https://github.com/asutosh05/SmartContracts/blob/master/ShoppingApp.sol

附中文参考文献:

[1] 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.

[2] 朱雪阳. The DAO 事件的形式化分析. 信息技术与网络安全, 2021, 40(5): 13−19.

[3] 周匀. 基于承诺的区块链智能合约研究 [硕士学位论文]. 上海: 上海交通大学, 2018.

[5] 刘德林. 区块链智能合约技术在金融领域的研发应用现状、问题及建议. 海南金融, 2016(10): 27−31.

[6] 赵延红, 原宝华, 梁军. 区块链技术在医疗领域中的应用探讨. 中国医学教育技术, 2018, 32(1): 1−7. [doi: 10.13566/j.cnki.cmet.cn61-1317/g4.201801001]

[7] 蔡一博. 智能合约与私法体系契合问题研究. 东方法学, 2019, 68(2): 68−81. [doi: 10.19404/j.cnki.dffx.20190304.002]

[8] 赵颖琪, 朱雪阳, 李广元, 高雅, 包玉龙. 带时间约束的智能合约验证. 应用科学学报, 2021, 39(1): 1−16.

[9] 高枫. 区块链智能合约漏洞修复困难. 计算机与网络, 2018, 44(12): 50−51.

[10] 倪远东, 张超, 殷婷婷. 智能合约安全漏洞研究综述. 信息安全学报, 2020, 5(3): 78−99.

[11] 邱欣欣, 马兆丰, 徐明昆. 以太坊智能合约安全漏洞分析及对策. 信息安全与通信保密, 2019(2): 46−55.

[34] 李书霞, 王国卿, 庄雷. 区块链智能合约安全的逆向实时模型检测方法. 小型微型计算机系统, 2020, 41(10): 2030−2035.

[36] 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, 2019, 45(3): 445−457.

[37] 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]

赵颖琪(1994－), 女, 硕士, 主要研究领域为智能合约, 形式化方法.

李广元(1962－), 男, 博士, 研究员, CCF 专业会员, 主要研究领域形式化方法, 实时系统模型检测.

朱雪阳(1971－), 女, 博士, 副研究员, CCF 高级会员, 主要研究领域为形式化方法, 嵌入式系统设计.

包玉龙(1995－), 男, 硕士, 主要研究领域为形式化方法, 智能合约.