软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(6):1849−1866 [doi: 10.13328/j.cnki.jos.006253]

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

面向 MSVL 的智能合约形式化验证*

王小兵[1], 杨潇钰[1], 舒新峰[2], 赵 亮[1]

[1](西安电子科技大学 计算机科学与技术学院,陕西 西安 710071)

[2](西安邮电大学 计算机学院,陕西 西安 710121)

通讯作者: 赵亮, E-mail: lzhao@xidian.edu.cn

摘 要: 智能合约是运行在区块链上的计算机协议,被广泛应用在各个领域中,但是其安全问题层出不穷,因此在智能合约部署到区块链上之前,需要对其进行安全审计.然而,传统的测试方法无法保证智能合约所需的高可靠性和正确性.说明了如何使用建模、仿真与验证语言(MSVL)和命题投影时序逻辑(PPTL)对智能合约进行建模和验证:首先介绍了 MSVL 与 PPTL 的理论基础;之后,通过分析和对比 Solidity 与 MSVL 语言的特性,开发了能够将 Solidity 程序转换为 MSVL 程序的 SOL2M 转换器,并详细介绍了 SOL2M 转换器的设计思路;最终,通过投票智能合约和银行转账智能合约两个实例,给出了 SOL2M 转换器的执行结果.使用 PPTL 从功能一致性、逻辑正确性以及合约完备性这 3 个方面描述了合约的性质,给出了使用统一模型检测器(UMC4M)对合约进行验证的过程.

关键词: 区块链;智能合约;形式化方法;MSVL

中图法分类号: TP311

中文引用格式: 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

英文引用格式: 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). http://www.jos.org.cn/1000-9825/6253.htm

Formal Verification of Smart Contract Based on MSVL

WANG Xiao-Bing[1], YANG Xiao-Yu[1], SHU Xin-Feng[2], ZHAO Liang[1]

[1](School of Computer Science and Technology, Xidian University, Xi'an 710071, China)

[2](School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an 710121, China)

Abstract: Smart contract is a computer protocol running on the blockchain, which is widely used in various fields. However, its security problems continue to emerge. Therefore, it is necessary to audit the security of a smart contract before it is deployed on the blockchain. Traditional testing methods cannot guarantee the high reliability and correctness required by smart contracts. This study shows how to use modeling, simulation, and verification language (MSVL) and propositional projection temporal logic (PPTL) to model and verify smart contracts. First, the theoretical basis of MSVL and PPTL is introduced. Then, by analyzing and comparing the characteristics of solidity and MSVL, an SOL2M converter which can convert a solidity program to an MSVL program is developed and its design idea is introduced in detail. Finally, the execution results of SOL2M converter are given by two examples of a vote smart contract and a bank transfer smart contract. The properties of contracts are described by PPTL on three aspects: function consistency, logic correctness, and contract completeness. And the process of using UMC4M (Unified Model Checker for MSVL) to verify the contract is also given.

Key words: blockchain; smart contract; formal methods; MSVL

* 基金项目: 国家自然科学基金(61672403, 61972301); 陕西省重点研发计划(2020GY-043, 2020GY-210)

Foundation item: National Natural Science Foundation of China (61672403, 61972301); Key Research and Development Projects of Shaanxi Province (2020GY-043, 2020GY-210)

本文由"形式化方法与应用"专题特约编辑姜宇副教授推荐.

收稿时间: 2020-08-30; 修改时间: 2020-10-26; 采用时间: 2020-12-19; jos 在线出版时间: 2021-02-07

2008年,Satoshi Nakamoto首次提出了区块和链的概念[1],指出了当前的线上交易几乎都需要依赖可信第三方,这增加了交易成本且扩大了不必要的交易规模.因此,中本聪提出了一种点对点的数字货币交易系统,也就是比特币系统,这也标志着区块链时代的到来.区块链本质上是分布式的数据账本,使用了加密算法、共识机制等技术,保证了用户的隐私且在交易期间不再依赖可信第三方的支持.早在1994年,Nick Szabo就提出了智能合约的概念[2],并将其定义为可以自动执行复杂业务操作的数字合约.但由于无法保证在没有第三方介入的情况下正确执行合约条款,因此智能合约一直没有在实际中应用,直到区块链的出现.区块链所具备的去中心化、公开共享、可信性、集体维护以及不可篡改的特质,使其成为智能合约天然的执行平台.

目前,智能合约已在许多区块链系统上成功实现,比较著名的系统有以太坊(Ethereum)[3]和超级账本(hyperledger)[4].以太坊在2013年由俄罗斯学者Vitalik Buterin所推出的《以太坊:下一代智能合约和去中心化应用平台》[5]中被首次提出,是目前最大的区块链平台.在以太坊中,大多数智能合约程序由图灵完备语言Solidity编写[6],然后编译为以太坊虚拟机(Ethereum virtual machine,简称EVM)的字节码,并运行在EVM之上.随着区块链技术的出现和发展,智能合约已经被应用于医疗(如存储电子病历和基因数据)、金融(如P2P网络借贷)、资产管理等领域,但其安全性问题却层出不穷[7].2016年,攻击者利用The Dao智能合约的脚本代码漏洞[8],从中窃取了300多万以太币.2018年,BEC智能合约出现重大漏洞,攻击者利用批量转账函数中的整数溢出问题无限生成代币,造成了约60亿人民币的损失.

在实际应用中,智能合约涉及了大量用户的数字财产,一旦出现漏洞,就会造成无法挽回的损失.因此,在智能合约被部署到区块链平台之前,需要对其进行漏洞排查.传统的测试方法不能达到保证智能合约正确性和高可靠性的需求,国内外许多学者提出了使用形式化方法对智能合约进行验证的方式,主要包括定理证明、符号执行以及模型检测这3种.

本文提出一种基于时序逻辑语言(modeling, simulation and verification language,简称MSVL)的智能合约安全性的模型检测方法:首先,使用MSVL对智能合约程序进行建模,为了减少建模时大量的人工操作,开发了将智能合约语言Solidity转换为MSVL的转换器工具SOL2M,实现了智能合约建模程序的自动化生成;然后,使用命题投影时序逻辑(propositional projection temporal logic,简称PPTL)公式描述该智能合约的安全性性质;最后,在统一模型检测器(unified model checker for MSVL,简称UMC4M)中自动化验证建模程序是否满足给定的安全性性质,并通过实例说明该方法运用到智能合约验证中的可行性和实用性.

本文第1节将回顾MSVL与PPTL相关理论.第2节将介绍Solidity语言特性以及智能合约的安全问题.第3节将给出SOL2M转换器的研究与实现.第4节将给出使用MSVL对智能合约建模和验证的应用实例.第5节将给出相关工作.第6节将给出总结及展望.

1 MSVL与PPTL

1.1 MSVL

MSVL的初始版本是Framed Tempura[9],在引入了等待语句和非确定选择语句之后,得到了建模、仿真与验证语言MSVL.MSVL是投影时序逻辑(projection temporal logic,简称PTL)的可执行子集,具有与高级程序设计语言类似的语法.下面介绍MSVL的基本语句.

(1) 区间长度语句:*empty*,*skip*,*len*(*n*);
(2) 区间框架语句:*frame*(*x*);
(3) 并行语句:*p*||*q*;
(4) 顺序语句:*p*;*q*;
(5) 非确定选择语句:*p* or *q*;
(6) 合取语句:*p* and *q*;
(7) 等待语句:*await*(*b*);
(8) 立即赋值语句:*x*<=*e*;

(9) 下一状态赋值语句:x:=e;

(10) Always 语句:*alw*(p);

(11) Next 语句:next p;

(12) 循环语句:while b do p;

(13) 条件语句:if b then p else q;

(14) 投影语句:(p_1,…,p_m) *prj* q;

(15) 函数调用语句:*fun*(e_1,…,e_m).

其中,x 为变量,e 为算数表达式,b 为布尔表达式 p_1,…,p_m,p,q 为 MSVL 程序.

区间长度语句 *empty*,*skip*,*len*(n)分别声明了当前区间长度为 0,1,n;并行语句 p||q 表明 p 与 q 在当前状态下同时开始执行,并有可能在不同时间结束;非确定选择语句 p or q 表示在当前状态下可以执行 p 或 q 中的任意一个;顺序语句 p;q 表示 p 与 q 按照顺序执行;合取语句 p and q 表示 p 与 q 在当前状态下同时开始执行并同时结束;等待语句 *await*(b)将会循环判断表达式 b 的真假,直到 b 为真时结束循环;立即赋值语句 x:=e 与下一状态赋值语句x<=e 分别表示在当前状态和下一状态对变量进行赋值;*alw*(p)表示在所有状态下执行 p; next p 表示在下一状态执行 p; while b do p; if b then p else q 以及函数调用语句的用法与其他高级程序设计语言相同.MSVL 不仅包含赋值语句、循环语句、条件判断语句等基本语句,还加入了框架结构和投影结构,为描述软硬件系统提供了更强的表达能力.区间框架语句 *frame*(x)使得变量 x 的值能够在区间上自动遗传,否则,变量 x 仅在被赋值时的状态下有确定的值,在其他状态下变量值是不确定的;投影语句(p_1,…,p_m) *prj* q 使得 p_1,…,p_m 与 q 能够并行执行,且 p_1,…,p_m 顺序执行,而 q 在另一个状态区间上执行.

1.2 PPTL

本文描述智能合约性质使用的 PPTL 是 PTL[10]的一个可判定子集,具有完备的公理系统[11],与 MSVL 均属于 PTL 的子集,可以共同完成对软硬件系统的形式化验证.在表达能力方面,PPTL 等价于完全正则表达式,严格强于 LTL,能够很好地描述智能合约的性质.PPTL 的语法和语义介绍如下:

(1) PPTL 的语法

Prop 代表原子命题集合,p 代表原子命题,且 p∈*Prop*;P,P_1,…,P_m 以及 Q 代表 PPTL 公式;O(next)和 *prj* (projection)是 PPTL 中的时序操作符.PPTL 公式的归纳定义如下:

$$P,Q::=p|\neg P|P\wedge Q|OP|(P_1,\ldots,P_m)\ prj\ Q.$$

(2) PPTL 的语义

- PPTL 的状态:状态 s 被定义为集合 *Prop* 到{true,false}的一个映射关系 s:*Prop*→{true,false},表明一个原子命题在状态 s 上可以为真或为假.假设命题 p 在状态 s 上的布尔值定义为 $s[p]$,若 $s[p]$=true,表明是 p 在状态 s 上为真,否则为假;
- PPTL 的区间:令符号σ代表由一个或多个状态 s 组成的状态序列,即区间状态.|σ|代表区间长度,当区间中的状态为有限个时,|σ|=状态数−1,此时,区间为有穷区间;当区间中的状态为无限个时,区间长度|σ|=ω,此时,区间为无穷区间.为了统一有穷区间和无穷区间的表达,需要扩展非负整数集.令 N_0 代表非负整数集,N_ω代表 $N_0\cup\{\omega\}$,已知ω=ω,对于任意的 i∈N_0,都满足 i<ω.令$\preceq$代表≤−{(ω,ω)},区间σ=⟨s_0,s_1,…,$s_{|\sigma|}$⟩,区间$\sigma_{(i-1)}$($0\le i\preceq j\le|\sigma|$)为$\sigma$的子区间.区间操作符·可以将两个区间$\sigma$=⟨$s_0$,$s_1$,…,$s_{|\sigma|}$⟩和$\sigma'=\langle s'_0,s'_1,...,s'_{|\sigma'|}\rangle$连接为$\sigma\cdot\sigma'=\langle s_0,s_1,...,s_{|\sigma|},s'_0,s'_1,...,s'_{|\sigma'|}\rangle$,但前提是前一个区间$\sigma$必须为有穷区间;
- PPTL 公式的解释:令三元组 I=(σ,k,j)表示 PPTL 公式的解释,其中,σ代表区间,k 和 j 为整数且满足:$0\le k\preceq j\le|\sigma|$.令 P 代表一个 PPTL 公式,那么 P 的解释 I 就定义为 P 解释在σ的子区间σ(k…j)上并且在区间上满足.这里令 I^k_{prop} 代表在状态;
- s_k 上的解释,令⊨表示可满足关系,则公式 P 与解释 I 的可满足关系如下:

- $I\models p$:当且仅当 $s_k[p]=I^k_{prop}[p]=\text{true}$;
- $I\models\neg p$:当且仅当 $I\not\models p$;
- $I\models P_1\wedge P_2$:当且仅当 $I\models P_1$ 并且 $I\models P_2$;
- $I\models OP$:当且仅当 $k<j$ 并且$(\sigma,k+1,j)\models P$;
- $I\models(P_1,\ldots,P_m)\ prj\ Q$:当且仅当存在 $k=r_0\leqslant\ldots\leqslant r_{m-1}\preceq r_m\leqslant j$,使得对于所有的 $1\leqslant l\leqslant m$,有$(\sigma,r_{l-1},r_l)\models P_l$且对于下面情况有$(\sigma',0,|\sigma'|)\models P$:

  (1) $r_m<j$ 且$\sigma'=\sigma\downarrow(r_0,\ldots,r_m)\cdot\sigma(r_{m+1},\ldots,j)$;

  (2) $r_m=j$ 且$\sigma'=\sigma\downarrow(r_0,\ldots,r_h)$(其中,$0\leqslant h\leqslant m$).

1.3 UMC4M统一模型检测器

面向 MSVL 和 PPTL 的模型检测[12]是建立在形式化建模和形式化规约基础之上的.形式化建模采用 MSVL 程序 M 对待验证的系统进行建模,形式化规约一般采用 PPTL 逻辑公式 P 描述待验证系统需要满足的性质.由于 MSVL 和 PPTL 均是 PTL 的子集,因此可同时将模型 M 与性质 P 统一在 PTL 框架下.验证 $M\rightarrow P$ 是否成立来验证系统的正确性,根据逻辑公式推理可得 $M\rightarrow P$ 等价于$\neg(M\wedge\neg P)$.在传统模型检测方法中,只需判定 $M\wedge\neg P$ 是否成立便可验证性质满足性:如果公式成立,则模型违反性质;否则,模型满足性质.

UMC4M 工具对传统模型检测方法进行了改进,基础架构如图 1 所示.首先,将 PPTL 公式 P 取反,并转换为 MSVL 程序 M';再将 MSVL 建模程序 M 与 M'合取为 MSVL 程序 M and M',即:将$(M\wedge\neg P)$是否成立转化为 M and M'是否能够正确执行的问题;然后,通过 MSVL 编译器(MSVL compiler,简称 MC)[13]执行 M and M'程序,并生成可执行文件 *M_and_M'.exe* 和中间代码 MM'_{IR} (intermediate representation);接着,通过 KLEE$^+$处理 MM'_{IR} 并生成验证用例(verification case,简称 VC);最终,运行 *M_and_M'.exe* 并接受 VC,然后依据执行结果判定性质是否满足.

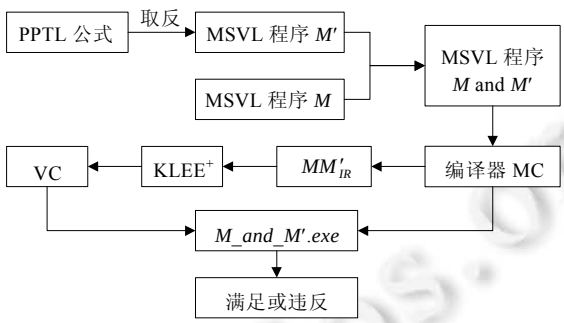


Fig.1 Infrastructure of UMC4M

图 1 UMC4M 基础架构

2 Solidity

Solidity 是用于开发智能合约的高级语言,其语法与 Javascript 相似.相比于高级程序设计语言,Solidity 缺少多线程以及并发等高级特性.使用 Solidity 语言编写的智能合约通过编译生成二进制字节码,并运行在以太坊虚拟机 EVM 上.

Solidity 源文件的后缀为 sol,主要由版本标识语句、导入其他源文件以及合约的定义这 3 部分组成.其中:版本标识语句使用保留字 pragma 声明 Solidity 的版本号,例如,pragma solidity^0.4.0,表明源文件不允许低于 0.4.0 版本的编译器编译;导入其他源文件部分使用保留字 import,相当于将多个源文件写到一个文件中,例如,import “filename”将“filename”中所有的全局符号导入到当前全局作用域中;合约的定义部分由关键字 contract 进行声明,例如,contract{...},{…}中封装了合约,实现了合约的具体功能.

Solidity 中的合约 contract 类似面向对象高级语言中的类 Class,包含状态变量(state variables)、函数

(functions)、事件(events)、结构体(struct)等.状态变量能够描述系统状态,类似于整个代码中的全局变量;函数是指包含一系列操作的可执行单元,能够完成特定的功能,由关键字 function 声明;结构体是指包含任意成员变量的自定义类型,由关键字 struct 声明,与高级程序设计语言中的结构体类似;事件是调用以太坊虚拟机 EVM 日志功能的接口,使用关键字 event 声明.

以下通过一个 Solidity 实例进行简单介绍(如图 2 所示).

```
1  Contract Demo {
2     uint v;
3     function set(uint x) {
4        v=x;
5     }
6     function get(·) returns (uint r) {
7        return v
8     }
9  }
```

Fig.2 Solidity example

图 2 Solidity 实例

合约 Demo 包含变量声明和函数两部分,声明了一个类型为无符号整数 uint 的状态变量 *v*,声明了一个 set 函数用于修改变量 *v* 的值,声明了一个 *get* 函数用于查询变量 *v* 的值.

3 SOL2M 转换器的研究与实现

本文使用 MSVL 对智能合约 Solidity 程序进行建模,为减少建模过程中大量的人工操作,开发了一种能够实现 Solidity 语言到 MSVL 语言的等价转换工具 SOL2M,实现了建模过程的自动化,其具体结构以及工作流程如图 3 所示.

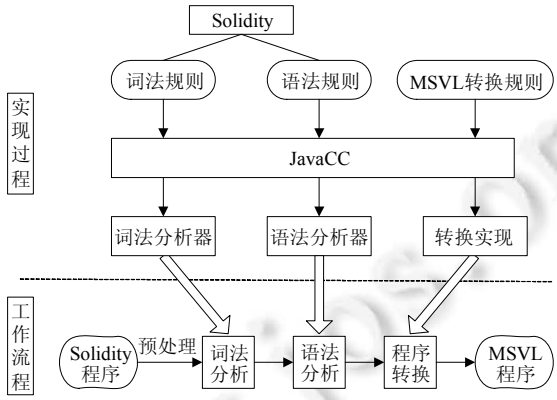


Fig.3 Architecture of SOL2M converter

图 3 SOL2M 转换器架构

SOL2M 转换器主要分为 4 个部分.

(1) 预处理:处理 Solidity 源程序中的版本标识语句和导入其他源文件语句;

(2) 词法分析:通过 JavaCC 工具生成词法分析器,对 Solidity 程序进行词法分析,将源程序识别为特定的单词流;

(3) 语法分析:通过 JavaCC 工具生成语法分析器,对 Solidity 程序进行语法分析,把词法分析生成的单词流识别为程序语句;

(4) 程序转换:通过分析 Solidity 和 MSVL 的词法以及语法的异同,制定出 Solidity 到 MSVL 的转换规则,并将转换代码嵌套在语法分析的 BNF 范式中,实现 Solidity 程序到 MSVL 程序的动态转换.

3.1 预处理模块

在预处理模块中,需要处理版本标识语句和导入其他源文件语句.由于版本标识语句并不会影响Solidity源程序的正确性和逻辑性,因此只需要对其进行移除.导入其他源文件是使用保留字 import 进行声明,导入语句的作用是待导入文件的所有全局变量导入到当前文件中的全局作用域中,以便于模块化代码.因此,需要先将待导入文件的全局变量合并到当前文件中,之后再进行词法分析.若不存在导入语句,直接进行后续分析.

3.2 词法分析模块

词法分析模块顺序扫描Solidity源程序,根据构词规则将其识别为具有特定含义的单词,并按照属性对其进行分类和标识.本文使用 JavaCC 工具自动生成词法分析器.一般的词法分析器包含 SKIP 与 TOKEN 两部分:SIKP 定义了源程序中需要忽略的部分,如注释部分和空白部分等;TOKEN 中使用正则表达式定义了需要特定识别的单词,在扫描源程序的过程中对字符序列进行分割并识别为单词及其属性,最终将所得到的 TOKEN 序列交给语法分析器.

SKIP 部分定义了需要自动跳过的部分,包括单行注释、多行注释、空格、缩进符、换行符以及回车符,使用正则表达式描述其构词规则如图 4 所示.

```
SKIP
{

  " "|"\t"|"\n"|"\r"

|("//"(~["\r","\n"])*("\n"|"\r"|"\r\n"))

|("/*"(~["*"])*"*"(~["/"](~["*"])*"*")*"/")

}
```

Fig.4 SKIP part

图 4 SKIP 部分

TOKEN 部分定义了需要特定识别的内容,主要包括保留字、标识符、常量以及运算符等.

(1) 保留字

如表 1 所示:int 与 uint 均可表示整型类型,且可以限定整型变量的长度,如 int8 到 int256,当省略后面数字时,默认为 int256;保留字 public/private/internal 用于表明函数或者变量的可见性,但由于可见性并不影响程序的逻辑性和正确性,因此予以忽略;保留字 event 用于定义一个事件,用于调用以太坊虚拟机日志功能的接口,起到储存记录的作用,并不影响程序的逻辑性和正确性,因此忽略 event 保留字.

Table 1 Reserved words

表 1 保留字

保留字	对应标记	保留字	对应标记
"contract"	CONTRACT	"return"	RETURN
"solidity"	SOLIDITY	"payable"	PAYABLE
"int"(⟨ISIZE⟩)?	INT	"if"	IF
"bytes"(⟨BSIZE⟩)?	BYTES	"while"	WHILE
"bool"	BOOL	"throw"	THROW
"function"	FUNCTION	"struct"	STRUCT
"public"	PUBLIC	"uint"(⟨ISIZE⟩)?	UINT
"address"	ADDRESS	"mapping"	MAPPING
"private"	PRIVATE	"internal"	INTERNAL
"returns"	RETURNS	"else"	ELSE

(2) 标识符

标识符包含用户自定义的合约名、函数名以及变量名等,其首字母为下划线或字母,其他部分为下划线、字母以及数字.当一个单词既能够被识别为保留字,又能够被识别为标识符时,将其首先识别为保留字.标识符的正则表达式如图 5 所示.

```
⟨IDENTIFIER:⟨LETTER⟩(⟨LETTER⟩|⟨DIGIT⟩)*⟩
⟨LETTER:["A"−"Z","_","a"−"z"]⟩
⟨DIGIT:["0"−"9"]⟩
```

Fig.5 Regular expressions of identifiers

图 5 标识符的正则表达式

(3) 运算符与界限符

运算符是语言固有的操作符号,Solidity 语言中的运算符见表 2.

Table 2 Operators

表 2 运算符

运算符	对应标记	运算符	对应标记
"+"	PLUS	"/"	DIVIDE
"−"	MINUS	"*"	MULTIPLY
"++"	SELFPLUS	"−−"	SELEFMINUS
"%"	REMAINDER	">"	GREAT
"<"	LOWER	">="	GREATER
"=="	EQUAL	"!="	UNEQUAL
"<="	LOWERW	"!"	NOT
"&&"	AND	"||"	OR
"&"	BIT_ADD	"|"	BIT_OR
"^"	BIT_XOR	"~"	BIT_NOT
"="	ASSIGNMENT	"+="	PLUS_ASSIGNMENT
"−="	MINUS_ASSIGNMENT	"*="	MULTIPLY_ASSIGNMENT
"/="	DIVIDE_ASSIGNMENT	"%="	REMAINDER_ASSIGNMENT
"&="	BIT_ADD_ASSIGNMENT	"|="	BIT_OR_ASSIGNMENT
"^="	BIT_XOR_ASSIGNMENT	"~="	BIT_NOT_ASSIGNMENT

界限符主要用于分割不同语句或标识符,Solidity 中的界限符见表 3.

Table 3 Boundary characters

表 3 界限符

界限符	对应标记	界限符	对应标记
"("	LC	") "	RC
"["	LM	"]"	RM
"{"	LB	"}"	RB
"."	SPOT	":"	COLON
";"	SEMICOLON	","	DOT

(4) 常量

常量为不会发生改变的内容,如整型常量、布尔常量、地址常量和字符常量,这 4 类常量通常能满足大部分智能合约的要求,如果合约需要更复杂的常量,只需在 JavaCC 的常量定义部分进行扩充,常量的正则表达式如图 6 所示.

```
INTEGER_LITERAL:⟨HEX_LITERAL⟩|⟨CONSTANT⟩
⟨HEX_LITERAL:"hex"(["0"−"9","a"−"f","A"−"F"])+⟩
⟨CONSTANT:"0"|["1"−"9"](⟨DIGIT⟩)*⟩
⟨DIGIT:["0"−"9"]⟩
```

Fig.6 Regular expressions of constants

图 6 常量的正则表达式

3.3 语法分析模块

语法分析建立在词法分析的基础之上,对词法分析生成的单词序列按照语法规则组成有特定含义的语句,并生成与源代码等价的语法树.本文使用 BNF 范式描述智能合约的语法规则.

首先,智能合约由结构体、状态变量、函数以及事件 4 部分组成,其 BNF 定义如图 7 所示.合约以 Start 为开始,以 Procedure 为终止.合约主体部分为 ContractBlock 声明,包含 Struct,Statedef 以及 Method,分别用于声明

结构体、状态变量与函数.

```
1  Start::=Procedure
2  Procedure::=⟨CONTRACT⟩⟨IDENTIFIER⟩⟨LB⟩(ContractBlock)*⟨RB⟩
3  ContractBlock::=Struct
4                  |Statedef
5                  |Method
6  Struct::=⟨STRUCT⟩⟨IDENTIFIER⟩⟨LB⟩(Type⟨IDENTIFIER⟩⟨SEMICOLON⟩)*⟨RB⟩
7  Type::=⟨UINT⟩(⟨LM⟩⟨RM⟩)?
8        |⟨ADDRESS⟩(⟨LM⟩⟨RM⟩)?
9        |⟨BOOL⟩(⟨LM⟩⟨RM⟩)?
10       |⟨MAPPING⟩⟨LC⟩Type"=>"Type⟨RC⟩(⟨LM⟩⟨RM⟩)?
11       |⟨IDENTIFIER⟩(⟨LM⟩⟨RM⟩)?
12 Statedef::=Type(Limit)?⟨IDENTIFIER⟩(⟨ASSIGNMENT⟩Expression)?⟨SEMICOLON⟩
13 Method::=⟨FUNCTION⟩⟨IDENTIFIER⟩⟨LC⟩(Type⟨IDENTIFIER⟩(⟨DOT⟩Type⟨IDENTIFIER⟩)*)?
14           ⟨RC⟩(Limit)?(FUNReturn)?⟨LB⟩(Statement)*⟨RB⟩
15 Limit::=⟨PRIVATE⟩
16       |⟨PUBLIC⟩
17       |⟨INTERNAL⟩
18 FUNReturn::=⟨RETURNS⟩⟨LC⟩Type(⟨IDENTIFIER⟩)?(⟨DOT⟩Type(⟨IDENTIFIER⟩)?)*⟨RC⟩
```

Fig.7 BNF definitions of smart contracts

图 7 智能合约的 BNF 定义

其次,智能合约源程序由各种基本语句组成,主要包括声明语句、表达式语句、条件判断语句、循环语句、转向语句、复合语句(也称语句块)以及空语句,其 BNF 定义如图 8 所示.非终结符 Statement 表示基本语句,IF, While,For 分别定义了条件判断语句与循环语句,Block 定义了复合语句,Statedef 定义了声明语句,return 与 throw 定义了两个跳转语句.

```
1  Statement::=IF
2              |While
3              |For
4              |Block
5              |⟨RETURN⟩(Expression)?⟨SEMICOLON⟩
6              |⟨THROW⟩⟨SEMICOLON⟩
7              |Expression⟨SEMICOLON⟩
8              |Statedef
9  IF::=⟨IF⟩⟨LC⟩Expression⟨RC⟩Statement(⟨ELSE⟩Statement)?
10 For::=⟨FOR⟩⟨LC⟩Statedef Expression⟨SEMICOLON⟩Expression⟨RC⟩Statement
11 While::=⟨WHILE⟩⟨LC⟩Expression⟨RC⟩Statement
12 Block::=⟨LB⟩(Statement)*⟨RB⟩
```

Fig.8 BNF definitions of basic statements

图 8 基本语句的 BNF 定义

表达式结构包含了程序中的各类运算与操作,其 BNF 最为复杂,如图 9 所示.第 1 行中的 Literal 表示可能出现的字面值,包括变量、结构体、数组、整型变量和布尔变量等;第 2 行的 Unop 表示单目运算;第 3 行~第 8 行的 Binop 表示双目运算符:Assign 表示赋值运算符,PLUS_MINUS 表示自增自减运算符.除此之外,还定义了[·],(·),.操作,可以描述 Solidity 中成员函数的调用.

```
1  Expression::=Literal Expression_left
2               |Unop Expression Expression_left
3  Expression_left::=Binop Expression Expression_left
4                    |Assign Expression Expression_left
5                    |"["Expression"]"Expression_left
6                    |"("(Expression","")*")"Expression_left
7                    |"."⟨IDENTIFIER⟩Expression_left
8                    |PLUS_MINUS Expression_left
```

Fig.9 BNF definitions of expression statements

图 9 表达式语句的 BNF 定义

3.4 语义分析与程序转换模块

语义分析是编译过程中的一个逻辑阶段,其功能为对源程序进行上下文有关性质的分析,审查是否存在语义错误,最终将源程序翻译为机器能够读懂的语言.本文对语法分析之后得到的 Solidity 源程序进行语义分析,根据每条语句的实际含义制定出 Solidity 与 MSVL 之间的等价转换规则,并在语义分析的过程中实现程序转换.

通过分析和对比 Solidity 语言与 MSVL 语言,制定了两者之间的转换规则.下面从合约以及基本语句的转换规则两方面进行介绍,部分转换规则见表 4.

Table 4 Conversion rules from Solidity to MSVL

表 4 Solidity 到 MSVL 的转换规则

类型	Solidity 程序	MSVL 程序

结构体	struct Candidate{
	bytes8 name;
	uint voteCount;
	}
		struct Candidate {
		char name and
		int voteCount
		};

声明语句	*uint* public *a*=0;	*int a* and *a*<=0 and skip;
	uint[·] public *arr*;	*int arr*[MAX] and skip;

函数	function *vote*(*uint id*) public{
	uint a=*id*;
	}
		function *vote*(*int id*){
		frame(*a*) and (
		int *a* and *a*<=*id* and skip
		)
		};

转向语句	throw; block;	skip;

	function *fun*(·) {
	return 0;
	}
		function *fun*(*int return_value*) {
		frame(*return_flag*)(
		int return_flag<=0 and skip;
		return_flag:=1 and *return_value*:=0)
		};

条件语句	if (expression){
	block
	}else{
	block
	}
		if (expression) then {
		block
		}else{
		block
		}

循环语句	for (*statedef*;*e*1;*e*2){
	block
	}
		while (*e*1) {
		block
		*e*2;
		}

含算术运算符的表达式	*e*1[+,−,*,/,%]*e*2 *e*1[++,−−]	*e*1[+,−,*,/,%]*e*2 *e*1:=*e*1[+,−]1
含赋值运算符的表达式	*e*1=*e*2 *e*1[+,−,*,/,%]=*e*2	*e*1:=*e*2 *e*1:=*e*1[+,−,*,/,%]*e*2
含逻辑运算符的表达式	*e*1&&*e*2 *e*1‖*e*2	*e*1 AND *e*2 *e*1 OR *e*2

智能合约主要由状态变量、结构体和函数构成.Solidity 中的 int8,uint256 等声明整型变量转换为 MSVL 程序后均使用 int.同理,Solidity 中的 bytes 转换为 MSVL 的 char 类型,地址变量 address 转换为 char 类型.在 Solidity 中,结构体的声明与 MSVL 的用法相同,只需要在结构体内部加入相应的时序操作即可.Solidity 与 MSVL 中函数声明的用法也相同,但函数中需要加入 frame 框架操作来保证变量值的遗传.

基本语句包括声明语句、表达式语句、条件判断语句、循环语句、转向语句、复合语句以及空语句.若 MSVL 当中存在 Solidity 某个语句的近似语句时,可以直接替换,例如 while 语句和 if 语句等;若 MSVL 中不存在近似语句,则需要对该语句进行抽象表达,例如 throw 语句.当抛出异常时后面的语句不再执行,由于在 MSVL 中不存在抛出异常的语句,转换时将 throw 语句与将要执行的语句一并转换为 skip,其区间长度为 1,并在下一状态执行空语句 empty.

根据 JavaCC 的特性,本文采用在语法分析过程中进行语义分析的方法来实现程序转换,根据制定的转换规则在每个语法分析函数中预定义了转换的语义动作.程序转换的基本流程如图 10 所示:SOL2M 转换器首先读取 Solidity 源程序文件(solidity.sol);然后进入语法分析阶段,提取 Solidity 程序语法单元的语义信息,并根据制定

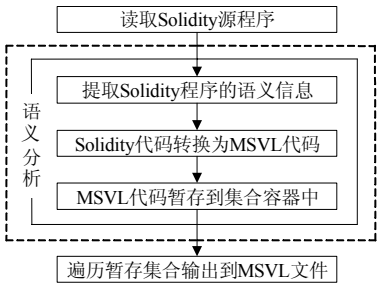


Fig.10 Basic flow of program conversion

图 10 程序转换的基本流程

的转换规则将 Solidity 代码转换为与其功能等价的 MSVL 代码;转换后的 MSVL 代码被暂存在预先创建的 Java 集合对象中,待语法分析结束后,遍历暂存集合并写入到 MSVL 程序文件(msvl.m)中.设 m 表示 Solidity 程序中变量声明的数量,n 表示语句数,p 表示每条语句中包含表达式的平均数,不难证明,SOL2M 转换流程的时间复杂度是 $O(n\times p+m)$.通常,Solidity 程序中一条语句的表达式数量、变量声明数量均不会超过一个常数,因此,SOL2M 转换流程的时间复杂度为 $O(n)$.UMC4M 验证 MSVL 程序基于 PPTL 的判定算法[10]:设 PPTL 公式的长度为 l,包含的命题组成集合 φ,则该算法的时间复杂度为 $O(l\times 2^{|\varphi|})$.

4 实例验证

本文使用 SOL2M 转换器对智能合约进行自动化建模,并提出了一种面向 MSVL 和 PPTL 的形式化验证方法.MSVL 是 PTL 的可执行子集,PPTL 是 PTL 的可判定子集,因此可以将 MSVL 与 PPTL 统一在 PTL 的框架中进行自动验证.验证流程如图 11 所示,具体为:使用 MSVL 对智能合约进行自动建模,存放在 m 文件中;使用 PPTL 描述智能合约待验证性质并存放在 p 文件中;将建模程序与待验证性质输入统一模型检测器 UMC4M 进行验证;若智能合约模型出错,则对 m 文件进行修改;若是待验证性质出错,则对 p 文件进行修改;若验证成功,UMC4M 检测器会给出模型是否满足性质的结果,或给出反例以便排查智能合约漏洞.

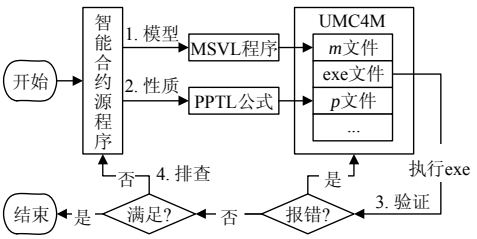


Fig.11 Basic flow of program conversion

图 11 程序转换的基本流程

本节将通过两个具体智能合约实例,使用 SOL2M 转换器对合约进行建模,使用 PPTL 公式描述合约需要满足的性质,最后在 UMC4M 统一模型检测器中完成验证.

4.1 投票智能合约的建模与验证

本小节以投票智能合约为例,给出 SOL2M 对其进行建模的详细流程以及验证过程.如图 12 所示,该投票协议主要包括状态变量、结构体和函数这 3 部分.合约名称为 voteContract,合约内部首先声明了候选者 Candidate 结构体,包含候选者名称与所获票数两个成员变量;其次创建了一个从候选者到票数的映射类型,以及使用结构体类型的数组 *Candidate*[ ]来存储所有候选者;合约内还声明了两个函数——*vote*( )函数实现了为特定候选者投票的功能,*winner*( )函数从所有候选者中选择出票数最高的获胜者.

首先对投票智能合约进行建模,将投票智能合约作为 SOL2M 转换器的输入被自动存入 solidity.sol 文件中,如图 13 所示,将 Solidity 代码转换为 MSVL 代码,结果存放在 msvl.m 文件中.建模之后,使用 PPTL 描述投票智能合约的性质,PPTL 公式中常用的时序操作符见表 5.本文将智能合约的性质归纳为 3 个层面:功能一致性、逻辑正确性以及合约完备性.下面从这 3 个方面描述投票智能合约应该满足的性质.

(1) 功能一致性.

每一个智能合约应满足的基本要求,是指智能合约的函数功能与合约设计要求之间的一致性,即待验证合约的功能应当符合实际要求.对于投票智能合约来说,其 *winner* 函数需要从所有候选者中选择出票数最高的赢家,若最终存在比赢家的票数更高的候选者,则说明 *winner* 函数没有实现实际要求,即不满足功能一致性.因此,描述性质 1 为:投票结束后,不存在比赢家票数更高的候选者.从原子命题、命题逻辑与时序逻辑这 3 个方面对性质 1 进行描述见表 6,最终使用 PPTL 公式描述性质 1 为 *fin*(!*m*).

```
1  contract voteContract {
2     struct Candidate {
3        bytes8 name;
4           uint voteCount;
5  }
6     uint public winnerId=0;
7     mapping(address=>uint) public voters;
8     Candidate[ ] public candidates;
9     function vote(address sender,uint id) public {
10       if (voters[sender]==0) {
11          return;
12       }else{
13          voters[sender]=voters[sender]-1
14          candidates[id].voteCount=candidates[id].voteCount+1
15       }
16    }
17       function winner(·) public {
18          uint maxCount=0;
19       for (uint i=0; i<candidates.length; i++) {
20          if (candidates[i].voteCount>maxCount) {
21             maxCount=candidates[i].voteCount;
22             winnerId=i;
23          }
24       }
25    }
26 }
```

Fig.12 Vote smart contract

图 12 投票智能合约

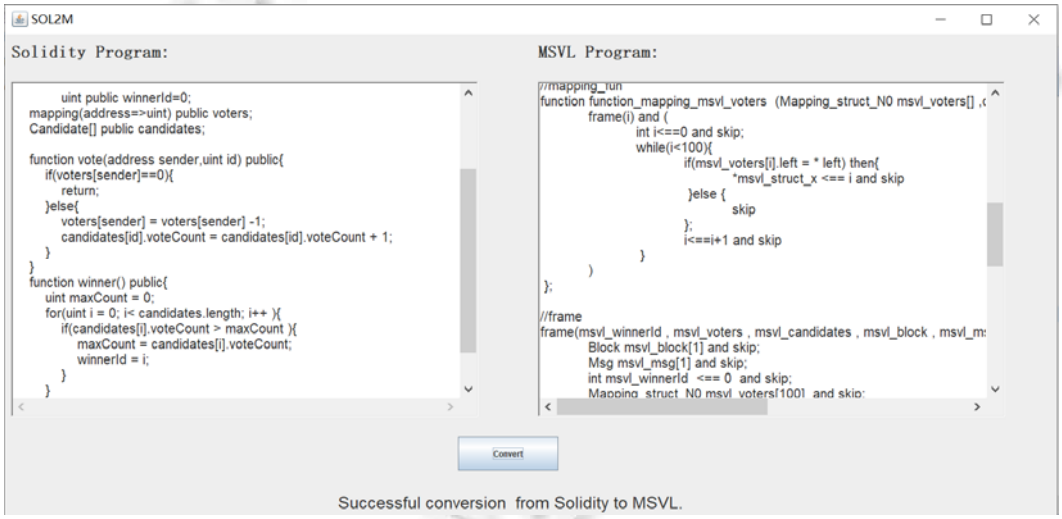


Fig.13 Modeling results of SOL2M converter

图 13 SOL2M 转换器建模结果

Table 5 Temporal operators in PPTL

表 5 PPTL 中的时序操作符

操作符	文本表示	操作符	文本表示
¬	!	∧	and
□	*alw*	∨	or
○	next	→	->
◇	*som*	*fin*	*fin*

Table 6 Description of property 1

表 6 性质 1 相关描述

类别	信息描述	操作符
原子命题	有其他候选者的票数比 *winner* 函数得到的赢家票数高	*m*
命题逻辑	不存在 *m* 成立的情况	!*m*
时序逻辑	投票结束后,不存在 *m* 成立的情况	*fin*(!*m*)

(2) 逻辑正确性.

关乎智能合约的安全问题,是指保证合约内函数的逻辑正确,即不存在逻辑漏洞.对于投票智能合约来说,*vote* 函数是整个合约的逻辑核心部分,其逻辑为:当投票者调用 *vote* 函数发起投票时,其手中拥有的票数至少为1,且当投票者手中有票时,一定会投票成功.将该逻辑精简为性质 2 与性质 3.

- 性质 2:当投票者投票成功时,则其手中一定有票.从原子命题、命题逻辑与时序逻辑这 3 个方面对性质 2 进行描述见表 7,最终使用 PPTL 公式描述性质 2 为 *alw*(*q*->*r*);

Table 7 Description of property 2

表 7 性质 2 相关描述

类别	信息描述	操作符
原子命题	投票成功 手中有票	*q* *r*
命题逻辑	若投票成功,则手中一定有票	*q*->*r*
时序逻辑	在所有状态下,(*q*->*p*)都成立	*alw*(*q*->*r*)

- 性质 3:当投票者调用 *vote* 函数且手中有票时,在将来某一状态下一定投票成功.从原子命题、命题逻辑与时序逻辑这 3 个方面对性质 3 进行描述见表 8,最终使用 PPTL 公式描述性质 3 为

alw((*p* and *r*) ->*som q*).

Table 8 Description of property 3

表 8 性质 3 相关描述

类别	信息描述	操作符
原子命题	调用 *vote* 函数 手中有票 投票成功	*p* *r* *q*
命题逻辑	调用 *vote* 函数且手中有票 若调用 *vote* 函数且手中有票,则投票成功	*p* and *r* (*p* and *r*)->*q*
时序逻辑	将来某一状态下投票成功 若调用 *vote* 函数且手中有票,则将来一定投票成功 在所有状态下,((*p* and *r*)->*som q*)都成立	*som q* (*p* and *r*)->*som q* *alw*((*p* and *r*)->*som q*)

(3) 合约完备性.

功能一致性与逻辑正确性的综合表达,只有满足功能一致性与逻辑正确性,才能认为合约满足完备性.对于投票智能合约来说,其合约完备被描述为性质 4:性质 1~性质 3 同时满足.使用 PPTL 公式描述性质 4 为

fin(!*m*) and (*alw*(*q*->*r*)) and (*alw*((*p* and *r*) ->*som q*)).

最后,将 MSVL 模型与 PPTL 公式输入到 UMC4M 中进行验证.在验证之前,需要在 MSVL 建模程序中加入对性质的定义,类似于代码插桩技术.

投票智能合约应满足的 PPTL 公式 *fin*(!*m*) and (*alw*(*q*->*r*)) and (*alw*((*p* and *r*)->*som q*))等价于 *fin*(!*m*) and *alw*(!(*q*) or *r*) and *alw*(!(*p* and *r*) or (*som q*)).将上述性质输入到 *p* 文件中,具体如下所示:

```
〈/
define m: m_flag=1;          //定义命题 m
define p: p_flag=1;          //定义命题 p
define q: q_flag=1;          //定义命题 q
```

```
define r: r_flag=1;              //定义命题 r
fin(!m) and alw(!(q) or r) and alw(!(p and r) or (som q))
/\
```

运行 UMC4M 检测器的结果如图 14 所示,验证结果为模型满足性质,总用时 653ms.

```
MC vote.p vote.m
  $$$$$$$$$$$$$$$$$
  ThreadPool init success!!!
  Verification Result: Val id!!!
Release ThreadPoo1 success!!!
  The run time is: 653ms
```

Fig.14 Verification results of the vote smart contract

图 14 投票智能合约验证结果

4.2 银行转账合约的建模与验证

本小节以具体的银行转账智能合约与常见的重入攻击漏洞为例,给出 SOL2M 对转账合约与重入攻击进行建模的详细流程以及验证过程.建模时,在银行转账合约中抽象加入了重入攻击操作,即将转账合约与重入攻击代码抽象为合约.如图 15 所示,其中,*deposit* 函数的功能是用户把钱存入银行,*withDraw* 函数实现了转账功能,用户通过 *withDraw* 函数与 *send* 函数从银行取钱.*Attack* 函数实现了重入攻击的功能,其具体步骤为:首先,攻击者调用 *Attack* 函数,通过 *deposit* 函数向银行转账合约中存入一些以太币;之后调用 *withDraw* 函数,从银行转账合约中取钱,使用 require 语句判断用户余额以及银行总余额是否足够;最后,调用 *send* 函数给用户地址转账.问题出现在第 10 行与第 11 行:合约执行 *send* 操作后修改用户余额,当合约发送以太币给攻击合约时会执行回退函数,建模过程中使用 fallback 模拟,其中会再次调用 *withDraw* 函数取钱.当执行 require 语句时,由于第 11 行还未执行,用户余额未改变,require 语句顺利通过,银行转账合约继续向攻击合约转账.不断重复这些操作,攻击者可以从合约中提取所有以太币.

```
1  pragma solidity>=0.4.22<0.6.0;
2    contract ReTry {
3      mapping(address>=uint) public userBalance;
4      uint public Limit=1;
5      uint public bankbalance=0;
6      function withDraw(address sender,uint toWithdraw) public {
7        uint amount=userBalance[sender];
8        require(amount>=toWithdraw);
9        if (amount>0) {
10         send(sender,toWithdraw);
11         userBalance[sender]-=toWithdraw;
12       }
13   }
14     function deposit(address sender,uint value) public {
15       userBalance[sender]+=value;
16       bankbalance+=value;
17   }
18     function send(address sender,uint toWithdraw) public {
19       msg.value+=toWithdraw;
20       fallback(sender);
21   }
22     function Attack(address sender) public {
23       deposit(sender,1);
24       withDraw(sender,1);
25   }
26     function fallback(address sender) public {
27       require(bankbalance>0);
28       withDraw(sender,1);
29   }
30 }
```

Fig.15 Bank transfer smart contract

图 15 银行转账智能合约

对合约进行建模,将银行转账合约与重入攻击合并为一个合约并作为 SOL2M 转换器的输入,转换结果如图 16 所示.

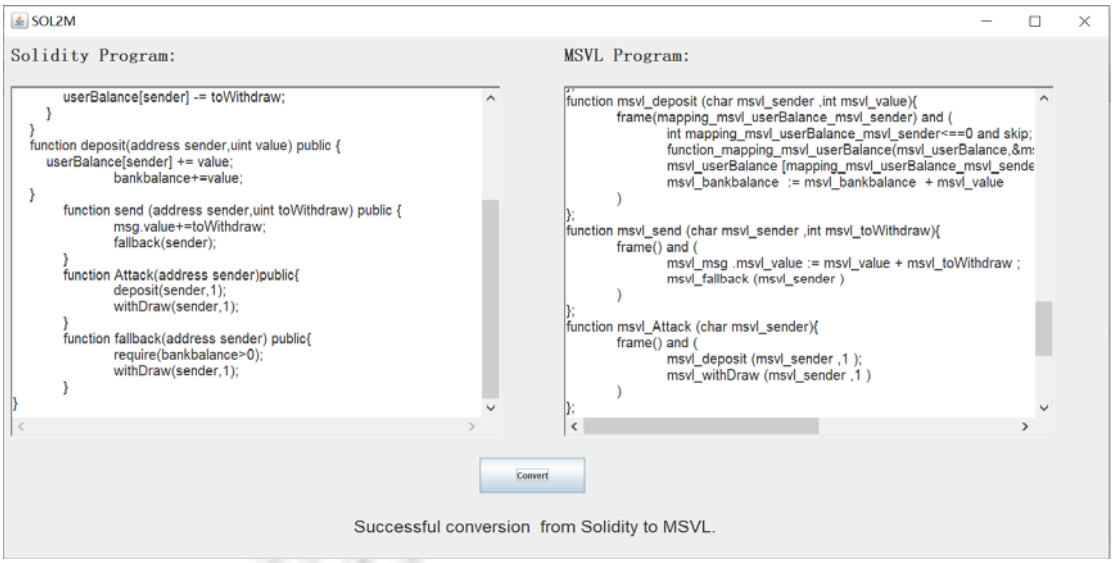


Fig.16 Modeling results of SOL2M converter

图 16 SOL2M 转换器建模结果

使用 PPTL 公式描述银行转账合约应满足的性质,下面从功能一致性、逻辑正确性以及合约完备性 3 个层面进行详细描述.

(1) 功能一致性.

对于银行转账合约来说,其 *deposit* 函数用于实现存钱操作.若用户调用了 *deposit* 函数,但用户余额没有增加,则说明 *deposit* 函数没有满足实际要求,即合约不满足功能一致性.将该功能精简为性质 5:在任何状态下,若用户调用 *deposit* 函数,则该用户余额在未来某一状态一定增加.从原子命题、命题逻辑、时序逻辑 3 个层面描述性质 5 见表 9,最终使用 PPTL 公式描述性质 5 为

$$alw(m\text{->}som\ n).$$

Table 9 Description of property 5

表 9 性质 5 相关描述

类别	信息描述	操作符
原子命题	用户调用 *deposit* 函数 用户余额增加	*m* *n*
命题逻辑	若用户调用 *deposit* 函数,则该用户余额一定增加	$m\rightarrow n$
时序逻辑	在将来某一状态下 *n* 一定成立 在所有状态下(*m*->*som n*)都成立	*som n* $alw(m\rightarrow som\ n)$

(2) 逻辑正确性.

在银行转账合约中,将以太币发送至地址需要提交外部调用.一旦被攻击者劫持,将会迫使合约执行更多的代码(即 *fallback* 回退函数).攻击者在 *fallback* 函数中回调受攻击合约的任意函数,即为重入合约.在本例中,攻击者可以多次回调 *deposit* 函数实现无限取钱操作.无重入攻击的合约应当在执行完外部调用时保证没有调用合约内所有函数.可能导致恶意情况发生的内部函数有 *deposit* 函数和 *withDraw* 函数,因此描述性质 6 为:在所有状态下都不存在当 *withDraw* 函数执行完外部调用时,*deposit* 函数或 *withDraw* 函数被再次调用.从原子命题、命题逻辑、时序逻辑这 3 个层面描述性质 6 见表 10,最终使用 PPTL 公式描述性质 6 为

alw(!(*f* and (*h* or *r*))).

Table 10 Description of property 6

表 10 性质 6 相关描述

类别	信息描述	操作符
原子命题	执行完外部调用 执行过 *deposit* 函数 执行过 *withDraw* 函数	*f* *h* *r*
命题逻辑	执行完外部调用时 *deposit* 或 *withDraw* 函数被调用	*f* and (*h* or *r*)
时序逻辑	在所有状态下(*g* and !*f*)都不成立	*alw*(!(*f* and (*h* or *r*)))

除此之外,当用户要求银行转账时,若该用户余额大于转账金额,则一定能够转账成功;并且当转账成功时,该用户余额一定大于转账金额.从原子命题、命题逻辑与时序逻辑这 3 个方面对性质 7 进行描述见表 11,最终使用 PPTL 公式描述性质 7 为

alw((*p* and *q*)->*g*) and alw(*g*->*q*).

Table 11 Description of property 7

表 11 性质 7 相关描述

类别	信息描述	操作符
原子命题	用户请求转账 用户余额大于转账金额 转账成功	*p* *q* *g*
命题逻辑	当用户请求转账且其余额大于转账金额时,则转账一定成功 当转账成功时,用户余额一定大于转账金额	(*p* and *q*)->*g* *g*->*q*
时序逻辑	在所有状态下((*p* and *q*)->*g*)与(*g*->*q*)都成立	*alw*((*p* and *q*)->*g*) and *alw*(*g*->*q*)

(3) 合约完备性.

合约完备性是功能一致性与逻辑正确性的综合表达,只有满足功能一致性与逻辑正确性,才能认为合约满足完备性.对于银行转账合约来说,其合约完备被描述为性质 8:性质 5~性质 7 同时满足.使用 PPTL 公式描述性质 8 为

alw(*m*->*n*) and *alw*(!(*f* and (*h* or *r*))) and *alw*((*p* and *q*)->*g*) and *alw*(*g*->*q*).

将 MSVL 模型与 PPTL 公式输入到 UMC4M 中进行验证:首先,将性质的定义加入到 MSVL 程序中,并把 MSVL 程序放在 *m* 文件中,PPTL 公式 *alw*(*m*->*som n*) and *alw*(!(*f* and (*h* or *r*))) and *alw*((*p* and *q*)->*g*) and *alw*(*g*->*q*)等价于 *alw*(!*m* or (*som n*)) and *alw*(!(*f* and (*h* or *r*))) and *alw*(!(*p* and *q*) or *g*) and *alw*(!*g* or *q*),在 *p* 文件中输入待验证的 PPTL 公式,具体如下所示:

```
〈/
define m: m_flag=1;          //定义命题 m
define n: n_flag=1;          //定义命题 n
define f: f_flag=1;          //定义命题 f
define h: h_flag=1;          //定义命题 h
define r: r_flag=1;          //定义命题 r
define g: g_flag=1;          //定义命题 g
define p: p_flag=1;          //定义命题 p
define q: q_flag=1;          //定义命题 q
alw(!m or (som n)) and alw(!(f and (h or r))) and alw(!(p and q) or g) and alw(!g or q)
/〉
```

运行 UMC4M 检测器,验证结果为性质不满足.经过对性质 5~性质 7 的分别验证,发现性质 5 与性质 7 满足而性质 6 不满足,表明存在重入攻击漏洞,攻击者劫持了外部调用,并在 *fallback* 回退函数中再次调用本合约中

的函数,实现了无限转账.

4.3 对比实验

实验环境为 64 位 Windows 7 系统,3.2GHz Intel(R) Core(TM) i5 处理器,32GB 内存.为了展示 SOL2M 工具将 Solidity 程序转换为 MSVL 程序的能力,如表 12 所示挑选了 9 个典型 Solidity 程序.程序列给出了 Solidity 程序的名称,LOC 列给出了 Solidity 程序的规模(行),LOM 列给出了从 Solidity 程序转换而来的 MSVL 程序的规模,时间列显示了完成转换任务所花费的时间.实验结果表明:对于这些典型的 Solidity 程序,SOL2M 工具都可以有效地转换为 MSVL 程序,生成的 MSVL 程序的规模约是 Solidity 程序的 2.25 倍.

Table 12 SOL2M 转换的实验结果

表 12 Experimental results of SOL2M conversion

程序	LOS	LOM	时间(ms)	增长倍数
overflow.sol	18	72	30	4.00
votecontract.sol	29	121	69	4.17
crowdfunding.sol	41	137	91	3.34
govermental.sol	44	114	73	2.59
racecondition.sol	49	110	55	2.24
reentrancy.sol	54	132	78	2.44
safemath.sol	65	128	75	1.97
sushirestaurant.sol	110	156	256	1.42
multisigwallet.sol	118	219	124	1.86

下面通过与 Oyente 工具进行比较,来分析 UMC4M 的有效性.从以上 9 个程序中挑选 5 个进行实验,表 13 给出了验证的实验结果.程序列给出了 Solidity 程序的名称,堆栈深度攻击漏洞、交易订单依赖、时间戳依赖、可重入漏洞列给出了验证的安全性质,每列下分别给出了 UMC4M 和 Oyente 的验证结果.√表示智能合约不存在该安全漏洞,而×表示存在该安全漏洞.验证时间列给出了 UMC4M 和 Oyente 验证所花费的时间.实验结果表明:UMC4M 和 Oyente 的验证结果大部分相同,但前者能发现 governmental.sol 的堆栈深度攻击漏洞和时间戳依赖,且能发现 racecondition.sol 的交易订单依赖,后者均不能发现这些安全漏洞.另外,UMC4M 的验证时间已经包含了 Solidity 程序到 MSVL 程序的转换时间,只占 Oyente 的 31%.

Table 13 UMC4M 与 Oyente 对比结果

表 13 Comparison results of UMC4M and Oyente

程序	堆栈深度攻击漏洞		交易订单依赖		时间戳依赖		可重入漏洞		验证时间(ms)	
	UMC4M	Oyente	UMC4M	Oyente	UMC4M	Oyente	UMC4M	Oyente	UMC4M	Oyente
overflow.sol	√	√	√	√	√	√	√	√	811	1 158
votecontract.sol	√	√	√	√	√	√	√	√	836	6 786
governmental.sol	×	√	√	√	×	√	√	√	787	2 190
racecondition.sol	√	√	×	√	√	√	√	√	837	888
reentrancy.sol	√	√	√	√	√	√	×	×	820	2 175

5 相关工作

目前,国内外学者针对智能合约安全问题给出了多种形式化验证方法,主要分为 3 类:定理证明、符号执行以及模型检测.

Bhargavan 等人提出了智能合约验证框架[14]并开发了 Solidity*工具,将 Solidity 智能合约源代码转换为等价的 F*程序,在源代码级别上来验证合约安全性.同时开发了 EVM*工具,在字节码级别上反编译 EVM 字节码,分析字节码属性并产生等价的 F*程序,然后进行交互式证明.杨霞等人提出一种面向智能合约的高度自动化的形式化验证系统及方法,将智能合约开发语言自动化转换为中间层语言 M+代码,然后通过形式化验证辅助器对其进行验证.在此基础上,提出一种基于定理证明的以太坊智能合约形式化验证方法,使用辅助验证工具 Coq 与 Isabelle 分别实现对 Solidity 源代码以及以太坊智能合约字节码的形式化验证[15].一方面,在 Coq 工具中建立源代码的形式化模型;另一方面,基于形式化操作码集在 Isabelle 中建立智能合约字节码的形式化模型,之后对

合约中的安全属性进行定理证明,验证其可满足性.以上方法属于定理证明,需要一定的人工证明,验证效率受到一定影响.

Luu 等人提出一种基于符号执行的智能合约安全性漏洞检测方法[16],开发了符号执行工具 Oyente,并非针对智能合约语句 Solidity,而是直接将 EVM 字节码和当前以太坊的全局状态输入工具,使用 Z3 求解器来判定可满足性.Oyente 能够在合约部署前检测其漏洞,该方法属于符号执行,一般只考虑符号路径的可满足性,自动化程度较高.当合约中的判断条件较多时,符号执行路径数量就会呈现指数级增长,且符号执行过度依赖约束求解器的求解能力,从而影响验证效率.

Kalra 等人开发了一种基于抽象解释和符号模型检测智能合约安全性的工具 ZEUS[17],其中包含一个能够将合约源代码转换为 LLVM 位码的代码转换器,同时也能够将 XACML 格式的待验证性质转换为 LLVM 位码.最后,将与合约及其性质等价的 LLVM 程序作为输入,使用 CHCs 工具进行了符号模型检测.胡凯等人提出了通过 Promela 建模语言结合 SPIN 工具对智能合约安全性进行验证的方法[18],使用 Promela 对智能合约进行建模,使用 LTL 描述待验证性质,在 SPIN 中自动化验证模型是否满足所期待的性质,最终给出代码是否与需求一致的结论.以上方法属于模型检测,由于自动化程度较高,因此能够很好地避免验证成本较大和效率低下的问题.

K 框架是一个语言无关、基于可达性的逻辑验证框架,KEVM 使用 K 框架来描述智能合约的形式化语义[19],并实现了语义分析工具,例如 Gas 分析工具、语义调试器、程序验证器等.KEVM 较难实现完整的程序分析,且需要投入较多的人力.本文采用基于 MSVL 的智能合约安全性的模型检测方法,性质描述采用的时序逻辑 PPTL 具有完全正则表达能力,严格强于 LTL.验证过程由模型检测器 UMC4M 自动完成,当状态空间过大时会产生状态空间爆炸问题[20].但是智能合约一般并非大型软件系统,因此该方法能够降低验证成本并提高验证效率.

6 总结与展望

为了更好地在智能合约部署之前对其进行安全验证,本文提出了面向 MSVL 与 PPTL 对智能合约进行安全性验证的模型检测方法,开发了 SOL2M 转换器,用于将智能合约 Solidity 程序转换为 MSVL 程序,即实现了对智能合约的自动化建模.使用 PPTL 公式从功能一致性、逻辑正确性以及合约完备性这 3 个方面描述了智能合约模型的性质.通过投票智能合约与银行转账智能合约两个实例来说明方法的可行性.目前阶段的转换是 Solidity 语言的子集,因此仍需继续对 SOL2M 转换器进行扩展和完善.另外,本文所给出的转换规则是基于两个语言的词法规则和语法规则所制定,将来的研究工作包括从数学角度证明 Solidity 与 MSVL 在表达式和基本语句上的语义等价性.

References:

[1] Nakamoto S. Bitcoin: A Peer-to-Peer Electronic Cash System. White Paper, 2008.

[2] Szabo N. Formalizing and securing relationships on public networks. First Monday, 1997,2(9).

[3] Wood G. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper, 2014,151(2014):1−32.

[4] Androulaki E, Barger A, Bortnikov V, Cachin C, Christidis K, De Caro A, Enyeart D, Ferris C, Laventman G, Manevich Y. Hyperledger fabric: A distributed operating system for permissioned blockchains. In: Proc. of the 13th EuroSys Conf. 2018. 1−15.

[5] Halstead MH. Elements of Software Science. New York: Elsevier Science Inc, 1977.

[6] Dannen C. Introducing Ethereum and Solidity. Berkeley: Apress, 2017.

[7] Bartoletti M, Pompianu L. An empirical analysis of smart contracts: Platforms, applications, and design patterns. In: Proc. of the Int'l Conf. on Financial Cryptography and Data Security. 2017. 494−509.

[8] Mehar MI, Shier C, Giambattista A, Gong E, Fletcher G, Sanayhie R, Kim HM, Laskowski M. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. Journal of Cases on Information Technology (JCIT), 2019,21(1):19−32.

[9] Ma YT, Duan ZH, Wang XB, Yang XX. An interpreter for framed tempura and its application. In: Proc. of the 1st Joint IEEE/IFIP Symp. on Theoretical Aspects of Software Engineering. Shanghai: IEEE Press, 2007. 251−260.

[10] Shu XF, Zhang N, Wang XB, Zhao L. Efficient decision procedure for propositional projection temporal logic. Theoretical Computer Science, 2020,838:1−16.

[11] Zhao L, Wang X, Shu XF, Zhang N. A sound and complete proof system for a unified temporal logic. Theoretical Computer Science, 2020,838:25−44.

[12] Duan ZH, Tian C. A unified model checking approach with projection temporal logic. In: Liu SY, Maibaum T, Araki K, eds. Proc. of the ICFEM 2008. LNCS 5256. London: Springer-Verlag, 2008. 167−186. [doi: 10.1007/978-3-540-88194-0_12]

[13] Yang K, Duan ZH, Tian C, Zhang N. A compiler for MSVL and its applications. Theoretical Computer Science, 2018,749:2−16.

[14] 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 PLAS 2016. 2016. 91−96. [doi: 10.1145/2993600.2993611]

[15] Fang Y. Research and implementation of formal verification technology based on Ethereum smart contract. University of Electronic Science and Technology of China, 2019 (in Chinese with English abstract).

[16] Luu L, Chu DH, Olickel H, Saxena P, Hobor A. Making smart contracts smarter. In: Proc. of the CCS 2016. ACM, 2016. 254−269.

[17] Kalra S, Goel S, Dhawan M, Sharma S. ZEUS: Analyzing safety of smart contracts. In: Proc. of the NDSS 2018. 2018. 1−12.

[18] Hu K, Bai XM, Gao LC, Dong AQ. Formal verification method of smart contract. Journal of Information Security Research, 2016, 2(12):1080−1089 (in Chinese with English abstract). [doi: 10.3969/j.issn.2096-1057.2016.12.003]

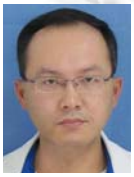
[19] Hildenbrandt E, Saxena M, Rodrigues N, Zhu X, Daian P, Guth D, Moore B, Park D, Zhang Y, Stefanescu A, Rosu G. Kevm: A complete formal semantics of the Ethereum virtual machine. In: Proc. of the IEEE 31st Computer Security Foundations Symp. (CSF 2018). Oxford, 2018. 204−217.

[20] Clarke E, Grumberg O, Jha S, Lu Y, Veith H. Progress on the state explosion problem in model checking. In: Proc. of the Informatics 2001. Berlin, Heidelberg: Springer-Verlag, 2001. 176−194.

附中文参考文献:

[15] 方言.基于以太坊智能合约的形式化验证技术研究与实现.电子科技大学,2019.

[18] 胡凯,白晓敏,高灵超,董爱强.智能合约的形式化验证方法.信息安全研究,2016,2(12):1080−1089. [doi: 10.3969/j.issn.2096-1057.2016.12.003]



王小兵(1979－),男,博士,副教授,博士生导师,CCF 高级会员,主要研究领域为形式化方法,形式化验证,时序逻辑.

舒新峰(1975－),男,博士,教授,CCF 专业会员,主要研究领域为可信软件技术及应用,智能信息处理.

杨潇钰(1999－),女,硕士,CCF 学生会员,主要研究领域为形式化方法,形式化验证,时序逻辑.

赵亮(1984－),男,博士,副教授,CCF 专业会员,主要研究领域为形式化方法,形式化验证,时序逻辑.