一种高级智能合约转化方法及竞买合约设计与实现

朱岩[1)] 秦博涵[1)] 陈娥[1)] 刘国伟[2)]

[1)](北京科技大学 计算机与通信工程学院, 北京 100083)

[2)](北京市经济和信息化局， 北京 100744)

摘 要 智能合约是运行在区块链上的数字协议，智能合约的开发涉及计算机、金融、法律等多个领域，近年来高级智能合约语言已被提出用于解决不同领域人员阅读、交流与协同开发难的问题，然而上述语言与可执行智能合约语言之间仍缺少有效的转化方法。针对这一问题，本文设计了一种 SPESC 到目标程序语言（Solidity）的转化规则，并提出了一种包括高级智能合约层、智能合约层和机器代码执行层的三层智能合约系统框架。首先，转化规则给出了根据 SPESC 合约当事人定义生成目标语言当事人子合约、以及 SPESC 其余部分生成目标语言主体子合约之间的对应关系；其次，除程序框架与存储结构外，目标语言程序还包含当事人人员管理、程序时序控制、异常检测等机制，这些机制能辅助编程人员半自动化地编写智能合约程序；进而，通过两个实验验证了上述高级智能合约框架的易读性以及转换的正确性，第一个实验邀请了计算机与非计算机人员分组阅读 Solidity 和 SPESC 的智能合约并回答问卷，结果表明阅读 SPESC 的速度约为阅读 Solidity 两倍，准确率也更高。然后以竞买合约为实例，给出了根据上述转化规则从 SPESC 合约转化到可执行 Solidity 合约语言程序，并通过以太坊私链部署运行来验证转化过程的正确性。实例表明上述转化规则和系统框架可简化智能合约的编写、规范智能合约的程序结构、辅助编程人员验证代码的正确性。

关键词 智能合约；面向领域语言；代码生成；SPESC

中图法分类号 TP319

An Advanced Smart Contract Conversion and Its Design and Implementation for Auction Contract

ZHU Yan[1)] QIN Bo-Han[1)] CHEN E[1)] LIU Guo-Wei[2)]

[1)](Department of Computer and Communication Engineering, University of Science and Technology, Beijing 100089)

[2)](Beijing Municipal Bureau of Economy and Information Technology, Beijing 100744)

Abstract As second-generation blockchain technology, smart contracts have greatly enriched the functional expression of blockchain to make application development more convenient. Smart contracts are a set of digitally executable protocols which concerns business, finance, contract law, and information technology. In recent years, advanced smart contract languages (ASCLs) have been proposed to solve the problem of difficult reading, comprehension, and collaboration when writing a smart contract among people in different fields.

本课题得到国家科技部重点研发计划(018YFB1402702)、国家自然科学基金(61972032)资助. 朱岩(**通信作者**)，男，1974年生，博士学位，教授，主要研究领域为信息安全与密码学、安全计算、区块链、移动云计算，E-mail: zhuyan@ustb.edu.cn. 秦博涵，男，1995年生，硕士研究生，主要研究领域为区块链、智能合约，E-mail:qinbohan@126.com.. 陈娥，女，1991年生，博士研究生，主要研究密码学和网络安全.刘国伟，男，高级工程师，主要研究大数据，安全系统，安全管理和标准.

However, this kind of languages are still hard to put into practice due to the lack of an effective conversion method from the ASCLs to executable smart contract programs. Aiming at this problem, we propose a three-layer smart contract framework, including advanced smart-contract layer, basic smart-contract layer, and executable machine-code layer. After comparing and analyzing the pros and cons of several ASCLs, we take SPESC as an example to explore how to design conversion rules from its contract to target language contract in Solidity. We specify the conversion rules from two aspects. One is program architecture of the target language, which consists of main-contract and party-contracts. The corresponding rules provide an approach to convert the definition of SPESC-based contracting parties into party sub-contracts on target language, as well as to produce the rest of SPESC contract into main sub-contract on target language. The other is the approach to specify not only program architecture and storage structure on basic smart-contract layer, but also important mechanisms, including personnel management, timing control, anomaly detection, etc. These mechanisms can assist programmers to semi-automatically write smart contract programs. Moreover, by introducing the notation of group, the SPESC-based smart contract can support the operation of dynamically adding participants into the contract. We also veify the legibility of SPESC and the correctness of the conversion process through two case studies. First, we invite some students from department of computer science and department of law. They, divided into four groups, are asked to read voting and auction contracts in SPESC and Solidity, and answer questions designed for the contracts. The result shows that the speed of reading SPESC is about twice as fast as that of reading Solidity, and the accuracy of reading SPESC is higher. Then, taking the auction contract as an instance, we analyze the process of bidding contracts and compile them into contracts in SPESC, and then provide the whole process of converting from a SPESC-based contract to an executable contract program in Solidity according to the above conversion rules, and verify the correctness of the conversion process, including coding, deploying, running, and testing, through Ethereum private chain. The instance results show that the conversion rules and the three-layer framework can simplify the writing of smart contracts, standardize the program structure, and help programmers to verify the correctness of the contract program. In our future work, a formal representation shall be established on the existing SPESC language model. Through formal methods, we can further provide formal analysis tools to verfy pre-and-post conditions of contract terms, as well as time sequence between terms. Secondly, in view of the correctness of the generated Solidity target code, we can continue to improve the generated target code based on existing researches on analysis or detection vulnerabilities, optimize the program structure and specifications, and enhance the security of the contract.

Key words Smart Contract, Domain Specific Language, Code generation, SPESC.

1 引言

智能合约[1]作为第二代区块链的技术核心，它是区块链从虚拟货币、金融交易到通用平台发展的必然结果。广义上讲，智能合约就是一套数字形式的可自动执行的计算机协议[2]。由于它极大地丰富了区块链的功能表达，使得应用开发更加便利，因此近年来它已引起学术界与工业界的广泛关注。

狭义上讲，智能合约就是部署并运行在区块链上的计算机程序。智能合约的代码、执行的中间状态、及执行结果都会存储在区块链中，区块链除了保证这些数据不被篡改外，还会通过每个节点以相同的输入执行智能合约来验证运行结果正确性。区块链的这种共识验证机制，保证了智能合约的不可篡改性和可追溯等特性，从而使得它具备了被法律认可的可能。

智能合约相较于比特币的脚本指令系统，可以处理更加复杂的业务逻辑，并可以更加灵活地在区块链中存储包括合约状态在内的各种数据。目前各大区块链平台和厂商都添加了智能合约模块，较为流行的智能合约平台包括以太坊（Ethereum）、超级账本（Hyperledger Fabric）等。

在智能合约语言方面，以太坊的智能合约目前

支持 Serpent 和 Solidity 两种编程语言，Serpent 类似于 Python 语言，而 Solidity 类似于 JavaScript 语言；超级账本支持如 Go、Java 等传统编程语言进行编写；此外，其它平台也都在已有编程语言（如 C、C++、Java）基础上给出了智能合约开发工具。总之，从语言形式和运行环境上讲，目前的智能合约可以分为以下三种：

1） 脚本型智能合约：通过区块链中定义好的脚本指令和堆栈式类 Forth 语言完成基本的计算与条件控制，如比特币脚本系统。
2） 传统编程语言智能合约：其语言直接采用传统程序语言，部署在虚拟机（VM）或容器（Docker）里，通过规定好的接口与区块链进行交互。如超级账本平台中的链码采用 Java、Go 等语言，Neo 平台支持将 C#、Java 和 Python 等多种语言编译为 NeoVM 支持的指令集。
3） 专用智能合约语言：模仿传统程序语言并添加了与区块链交互的特殊元素，如以太坊的 Solidity 语言，同时该语言含有 gas 计费等特殊功能。

研究动机。通过上述分类可知，智能合约平台和语言已日益成熟且功能趋于完善。然而，由于智能合约通常涉及到计算机、法律、金融等多领域的协作，而目前的智能合约编程语言存在对于非计算机领域人员不够友好，对没学习过编程的人员来说难以理解等问题。具体而言，目前的智能合约语言存在以下几个缺点：

1） 程序语言与法律合约形式相去甚远；
2） 智能合约程序专业性强，用户和法律人员难以理解；
3） 从法律合约到可执行智能合约代码生成没有建立直接联系。

这些缺点导致合约的编写非常困难，不同领域人员之间交流存在障碍，大大制约了智能合约的开发效率和公众的认可程度。

近年来高级智能合约语言（ASCL）已被一些学者提出来解决上述问题。这种语言是介于现实合同与智能合约之间的一种语言，通过易读且规范化的语法，帮助不同领域人员进行沟通，并可（半）自动化地实现向平台智能合约语言的转化，辅助编程人员进行编写。

相关工作。基于区块链的智能合约概念被提出以来，不少学者在程序设计与平台构造等方面都做了大量研究工作。这些工作中的多数研究是通过形式化语言验证合约的正确性，如文献[3][4][5][6][7]等；也有不少研究是从法律角度讨论智能合约，如文献[8][9][10] [11]等。但以易于读写的语法建立高级智能合约语言模型，并实现向可执行智能合约语言转化的工作比较少。下面将介绍与 ASCL 相关的几种研究：

文献[12]提出了一种被称为 Simplicity 的功能性语言，该语言通过对抽象机器上操作语义的评估，计算空间和时间资源消耗的上限，在执行之前计算出比特币脚本和以太坊虚拟机中的资源消耗费用，有利于解决智能合约的预付费问题。

文献[13]提出了一种类自然智能合约语言（SmaCoNat），以可读性与安全性为目标，通过限制用户定义变量名、限制嵌套的使用、代码分节、拓展基础类型、自然语法、统一身份表示的几种措施增强合约的可读性与安全性。

文章[14]提供了一个新的自动生成智能合约的框架，其框架利用语义规则对特定领域的知识进行编码，然后利用抽象语法树的结构来合并所需的约束，最终可以通过经过约束的语法编码为区块链的智能合约。

文献[15]提出了一种新智能合约语言（Findel），着重从金融的角度描述了合约的资金转移动作及乘法、逻辑、时序表达式。但 Findel 只包含两种基本动作：Zero 与 One；两种乘法运算：Scale 与 ScaleObs；Give 变更执行方；三种逻辑表达式：and、or 和 if；以及 Timebound 时间表达式。因此，该语言功能较单一。

文献[16]提出了一种类自然语言智能合约语言（SPESC），它是一种以解决智能合约语言对于非计算机人员难以理解的问题为目标提出的高级智能合约语言。SPESC 合同结构包括合约名称、合约当事人、合约条款和附加属性四部分，前三者分别与现实合同中的合同名称、合同主体和合同主要内容对应，合约属性则是为了利用区块链不可篡改性记录合约中的重要信息及变更过程。

与其它智能合约语言相比较（见第二节），SPESC 的语法易读、结构清晰，且包含完整的语言模型定义，是智能合约未来的发展趋势之一，更接近本文对高级智能合约语言的要求，但目前 SPESC 还没有转化为可执行智能合约语言的生成器，因此，本文将继续针对这一问题进行研究。

本文主要工作。本文针对高级智能合约语言与可执行智能合约语言之间缺少转化方法的问题，通过常

用合约的实例化研究，设计一种针对 SPESC 语言的可执行代码生成器，给出了 SPESC 与目标程序语言（以 Solidity 为例）之间的转化关系，可简化智能合约的编写、规范智能合约的程序结构、辅助编程人员验证代码的正确性。具体工作如下：

1）提出一种包括高级智能合约层、智能合约层和机器代码执行层的三层智能合约系统框架，该框架是在综合了已有的 SmaCoNat、SWRL、Findel、SPESC 等高级智能合约语言特点基础上而提出的，并给出了高级智能合约语言编写智能合约的基本流程。

2）给出了一种 SPESC 到目标程序语言（Solidity）的转化规则。首先，给出了根据 SPESC 合约当事人定义生成目标语言当事人子合约、以及 SPESC 其余部分生成目标语言主体子合约的对应关系；其次，除程序框架与存储结构外，目标语言程序还包含当事人人员管理、程序时序控制、异常检测等机制，这些机制能辅助编程人员半自动化地编写智能合约程序。

本文以竞买合约为实例，给出了根据上述转化规则从 SPESC 合约转化到可执行 Solidity 合约语言程序、以及该程序的部署、运行、测试的全过程。首先，通过引入当事人群体，实现了支持当事人动态加入的 SPESC 竞买智能合约；然后，在完成可执行 Solidity 竞买合约语言程序后，通过以太坊私链部署并运行测试，验证了转化过程的正确性。

组织结构。第 2 节将 SPESC 与几种相关工作对比分析。第 3 节介绍包含高级智能合约语言的智能合约系统框架；第 4 节介绍 SPESC 的语法；第 5 节分析竞买的流程；第 6 节展示如何通过 SPESC 编写竞买合约；第 7 节说明以 Solidity 为目标语言的 SPESC 编译规则；第 8 节将计算机与法律学生分组阅读 SPESC 与 Solidity 编写的合约，完成问答验证易用性，并将生成的 Solidity 智能合约部署并测试；最后进行总结与展望。

2 相关工作

下面我们对本文相关的几个工作[13][14][15]进行梳理，并将它们与 SPESC[16]相比较，从而介绍 SPESC 的机制与优势。

1）SmaCoNat[13]与 SPESC 同年被提出，两者结构相似。如图 1 所示，SmaCoNat 合约包含合约头（Heading）、账户（AccountSection）、资产（AssetSection）、协议（AgreementSection）、事件（EventSection）。其中，合约头、账户、事件分别类似于 SPESC 中的合约名称、当事人、条款，而资产与协议分别用来声明合约中涉及的资产以及对资产初始化。该合约中展示了 SmaCoNat 语言编写的出售停车票来控制停车场的合约。

SmaCoNat 对于资产做出了更加具体的描述与限定，但是 SmaCoNat 与本文中对 ASCL 的要求相比存在一些差异：①SmaCoNat 中没有表达如何在合约中存储信息，只支持对于资产转移的描述，因此应用范围较小；②SmaCoNat 中没有对于时序的控制，每个 Input Event 之间相互独立，仅通过资产进行联系，条款之间的关系更难梳理与理解。

```
Contract in SmaCoNat version 0.1.

§ Involved Accounts:
Account 'BarrierIn' by 'AComp' by Genesis alias 'BarrierIn'.
Account 'BarrierOut' by 'AComp' by Genesis alias 'BarrierOut'.

§ Involved Assets:
Asset 'TheCoin' by Genesis alias 'TheCoin'.
Asset 'ParkTicket' by Self alias 'Ticket'.
Asset 'OpenBarrier' by Self alias 'Open'.

§ Agreement:
Self issues 'Ticket' with value 42.
Self issues 'Open' with value 1.

§ Input Event:
if Input is equal to 'TheCoin' from Anyone
and if value of Input is equal to 0.3
then
    Self transfers 'Ticket' with value 1 to owner of Input.
    Self transfers 'Open' with value 1 to 'BarrierIn'.
    Self issues 'Open' with value 1.
endif

if Input is equal to 'Ticket' from Anyone then
    Self transfers 'Open' with value 1 to 'BarrierOut'.
    Self issues 'Open' with value 1.
endif
```

图 1 SmaCoNat 合约

2）文献[14]采用改进后的网络本体语言（OWL）——语义网规则语言（SWRL）描述智能合约。如图 2 所示，子图(a)是通过 SWRL 表示的智能合约，子图(b)是由 SWRL 转化的 JSON 格式键值对，子图(c)是最终生成的 Go 语言智能合约。图 2 展示了对病人信息进行校验的合约，该合约要求病人性别为女性，且年龄大于 6 岁。

$$Patient(?\,patient) \wedge hasGender(?\,patient,?\,gender) \wedge swrlb:equal(?\,gender,"Female") \wedge hasAge(?\,patient,?\,age) \wedge swrlb:greaterThanOrEqual(?\,age,6) \rightarrow Eligible(?\,patient)$$

(a)

```
"patient_prop_check_array"
["operator":"equals","value":"Female","property":"gender",
"operator":"greaterThanOrEqual","value":"6","property":"age"]
```

(b)

```
//Generated Patient Inclusion Constraints
age_constraint_lower := 6
gender_constraint := "Female"
//If Patient satisfies all constraints, permit to be written to ledger
if age >= age_constraint_lower
&& strings.Contains(gender_constraint , gender){
    newPatient := Patient(ID: id, Age: age, Gender: gender, Precondition: precondition)
    patientAsBytes, _ := json.Marshal(newPatient)
    APIstub.PutState(id, patientAsBytes)
} else {
    //or else reject CreatePatient Transaction
    return shim.Error("Invalid Patient Info")
}
```

(c)

图 2 通过语义网规则（SWRL）生成合约

尽管该文献提供了从 SWRL 自动转化为智能合约的生成器，但是该语言（如图 2 (a)所示）采用本体论语言的语法表示，不易读写，且主要应用于对数据的限制与检验，缺乏对于数据、合约状态变化以及金融方面的描述。

3）Findel[15]是一种声明式面向金融的智能合约语言，通过两种资金转移动作与乘法、逻辑、时序三类表达式的组合编写合约。其合约最终体现为一个表达式，如下所示：

$$c_{zcb}=\left\{\begin{array}{l}And(Give(Scale(10,One(USD))),\\ At(now+1years,Scale(11,One(USD))))\end{array}\right\}$$

该表达式表示：出借人向借款人借款 10USD，一年后借款人还款 11USD。由此可见，Findel 可以表示具有时序关系的简单金融合约，但无法支持变量的定义，且一个合约只能涉及两个当事人，与高级智能合约语言 ASCL 要求不符。

4）SPESC[16]语言类似于自然语言，通过采用现实世界合同中的语法元素，构建了 SPESC 语法模型，支持合约当事人权利（can）与义务(shall)、以及资产转移规则的定义。例如，SPESC 描述的买卖合约如图 3 所示。

```
contract SimplePurchase2{

    party Seller{
        post()
        collect()
    }
    party Buyer{……}

    info : ProductInfo

    term no1: Buyer can pay…….
    term no2: Seller shall post
        when within 5 day after Buyer did pay.
    term no3: Buyer can confirmReceive…….
    term no4: Seller can collect…….

    type ProductInfo {
        price : Money
        model : String
    }
}
```

图 3 SPESC 合约示例

该合约展示了销售者和购买者之间商品买卖的基本流程，包含四条条款：购买者可以付款；购买者付款后，销售者需在 5 天内发货；购买者可确认收货；销售者有权收取货款。

SPESC 兼顾了现实合约与现有智能合约的特点：通过条款方式明确了合约当事人的权利与义务，同时面向司法与金融领域定义了当事人行为、资产转移操作、履行期限以及变量化的合约信息。因此，SPESC 支持编写买卖、竞买、借贷等金融类合约，同时也可实现投票、存证等司法类合约。与其它方案相比，SPESC 更接近于 ASCL 要求。

3 系统框架

3.1 系统目标

基于区块链的智能合约系统通常是指支持可自动执行的合约代码生成并运行的软件系统。现有的智能合约系统可分为两层：智能合约层与机器代码执行层。为了增强合约的规范性和易读性，本文的智能合约系统在这两层结构之上添加一层高级智能合约层，高级智能合约语言在智能合约语言之上提供了更优化的封装，形成了类似于高级程序语言与低级程序语言之间的关系，如 C++语言与汇编语言。

对本文的智能合约系统提出以下要求：

1） 高级智能合约语言更便于阅读与理解；

2） 规范化智能合约的编写；

3） 能支持高效的智能合约生成；

4） 生成更系统的目标语言智能合约程序框架。

3.2 智能合约编写框架

基于本文所提出的三层智能合约系统，合约当事人通过高级智能合约语言编写智能合约的流程及框架如图 4 所示。在此框架中，首先，用户可以根据现实合同或真实意图进行高级智能合约语言的编写；然后，通过高级智能合约语言的编译器，将合约转化为传统程序语言编写的智能合约；最后，生成机器代码并将其部署运行在区块链中。

1）现实合同在符合法律的情况下签订成立后就具有法律效力。现实合同的书写与形式相对自由，没有严格规定的格式，即使合同中存在歧义或缺省，也可以由司法机构根据法律与合约当事人的真实意图进行裁决。它与目前的智能合约相比，具有以下特点：

① 具有法律效力；

② 采用自然语言，书写自由；

③ 一旦出现争议可由人工裁决。

2）高级智能合约语言将计算机程序、法律、金融等多个方面的特点融合在一起，以一种既比从程序语言更容易被理解、又比自然语言更规范的方式表达合约内容。本文所采用的 SPESC 具有以下特点：

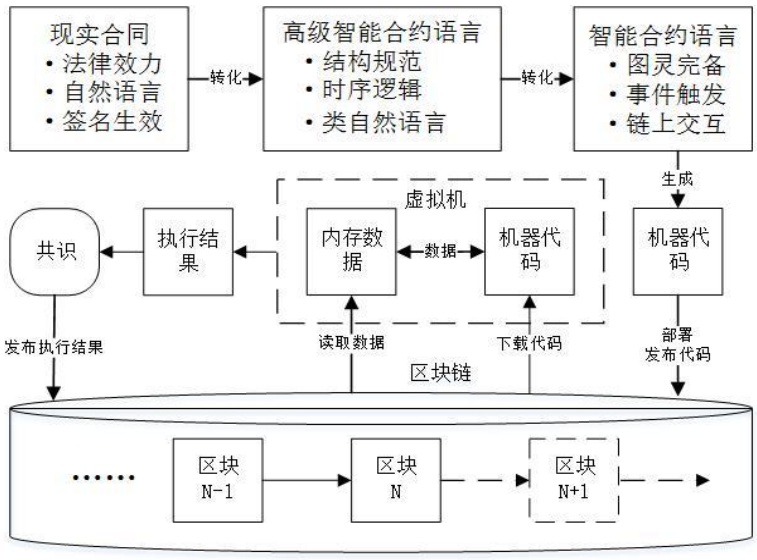

图 4 智能合约编写框架

① 参考现实合同的结构。

② 通过时序逻辑表达时序关系。

③ 采用类自然语言的语法。

3）现有智能合约语言与传统的程序语言类似，并在此基础上设计或预定义了区块链相关特殊元素或操作。由于智能合约主要涉及多个用户之间的交互，因此多采用事件触发的方式，由用户调用并触发智能合约运行。目前主流的智能合约语言如 Solidity、Java、GO 等具有以下特点：

① 语言是图灵完备的，表达能力强。

② 事件触发的方式执行。

③ 预定义了与区块链交互的操作。

4）智能合约通过编译生成机器代码后，可在虚拟机或容器中运行，它的部署执行步骤主要分下面三步：

① 部署智能合约。将机器代码发布到区块链中，使得其它参与共识的节点可以获取智能合约以便验证。

② 执行智能合约。在执行或验证智能合约前，将智能合约代码下载到本地，同时将存储在区块链中的合约状态恢复到内存中，并在本地虚拟机中运行。

③ 发布执行结果。根据输入参数运行智能合约后，将执行结果与其它参与验证节点共识，记录到到区块链中。

4 SPESC 介绍

下面介绍本文采用的高级智能合约语言 SPESC 的基本，SPESC 智能合约由四部分组成：合约名称、合约当事人描述、合约条款、附加信息。

定义 1.（合约）SPESC 合约具体定义如下：

Contract::= *Title*{*Parties*+ *Terms*+ *Additional*+}

合约的当事人可能是个人、组织或是群体，合约中应记录其关键属性和行为。

定义 2.（当事人）合约当事人具体定义如下：

Parties::= **party group**? PartyName {*Field*+ A*ction*+}

在上述定义中，group 关键词表示合约的当事人是群体，Field 表示当事人在合约中的需要记录的关键属性，Action 声明了当事人在合约中的权利与义务。

个体当事人是指合约中拥有一定权利或义务的个体，如买家、卖家等。群体当事人是指在合约中拥有相同权利与义务的多个个体，如投票人、竞拍人等，群体当事人既可以在合约执行前事先指定，也可以在合约运行中动态加入或退出。

当事人定义是为了便于处理与记录当事人的信息。每个个体在区块链中都有对应的的账户地址，因此，当事人属性中默认包含地址属性，用户可以通过设定具体地址来规定当事人的具体身份，也可以不在编写 SPESC 时规定，而在智能合约运行时根据条款与执行情况进行变更。

合约中的条款分为权利条款和义务条款两种，权利表示在一定条件下可以执行的动作，义务表示在一定条件下必须完成的动作，而未满足条件的动作或未被写入合约的动作表示禁止执行的动作。

定义 3.（条款）合约条款具体定义如下：

Terms::= **term** tname: PName **(shall|can)** AName

(**when** *PreCondition*)?

(**while** *TransferOperation*+)?

(**where** *PostCondition*)?.

在上述定义中，PName 表示当事人，AName 表示执行的动作，PreCondition 表示可执行该条款的前置条件，TransferOperation 表示执行该条款的过程中伴随的资产转移，PostCondition 表示该条款执行结束后该满足的后置条件。前置与后置条件区分的依据是：合约的业务逻辑可通过前置条件与时间表达式予以表达，对于程序预期外的情况，可通过后置条件予以限制。

资产转移的操作被分为存入、取出、转移三种。

定义 4.（资产转移）资产转移操作定义如下：

TransferOperation::=

{Deposit} **deposit** (**value** *ROP*)? *AssetExp*

|{Withdraw} **withdraw** *AssetExp*

|{Transfer} **transfer** *AssetExp* **to** *Target*

在上述定义中，ROP 表示关系操作，包含>、<、=、>=和<=，AssetExp 表示资产表达式，用于描述转移的资产，Target 表示资产转移的目标账户。

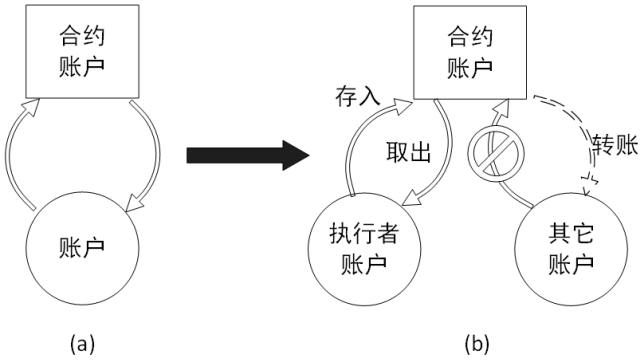

图 5 资产转移操作

为了保证智能合约检查与记录的功能，所有资产转移需通过合约账户实现。如A向B转账的操作，需先由A向合约账户转账，再由合约账户向B转账，通过上述方法，合约便于根据条款对转账条件与金额检查，同时在合约中记录转账信息。因此，合约中涉及的资产转移本质应分为两种，即账户向合约转移和合约向账户转移，如图 5 (a) 所示。

由于从用户账户向合约转移资产的操作只能由该用户自己主动执行，而不能强制执行，因此为了区分执行者与其它账户，将资产转移操作分为三种，如图 5 (b) 所示：

1） 存入（deposit）：用户主动将资产存入合约。

2） 取出（withdraw）：根据合约条款从合约取出资产。

3） 转移（transfer）：根据合约条款从合约向其它账户转移资产。

在三种资产转移操作中，存入操作与后两种操作不同，在后两者中，操作的资产都是事先在合约条款中约定，由常量或是变量可以准确描述的。而在存入操作中，由于是由用户主动执行，存入资产不一定由合约直接规定，合约只能对资产进行限定，如在第六节描述的合约中，合约无法直接规定竞买人在竞拍时的出价，而可以限制为出价一定要大于当前最高价。

此外，资产转移操作还与具体的智能合约平台及语言有关。例如，在以太坊平台中，账户既可以是合约账户也可以是用户账户。由于 Solidity 编写的合约中可以定义在收到以太币时自动执行的接收方法，向未知用户转移资产可能存在安全风险。因此，在以以太坊作为智能合约平台时，应尽量通过用户主动取款的方式代替转账方式。

5 竞买合约

本文将通过基 SPESC 的拍卖合约实例来说明 SPESC 生成器，并验证生成合约的正确性以及三层合约框架的可用性，本节将介绍竞买合约规则及流程。竞买(Auction)是专门从事拍卖业务的机构接受货主的委托，在规定的时间与场所，按照一定的章程和规则，将要拍卖的货物向买主展示，公开叫价竞购，最后由拍卖人把货物卖给符合规则的买主的一种现货交易方式。本文主要讨论以最高价成交的竞买合约。

竞买合约涉及两个当事人：

1）拍卖人：即从事拍卖活动的企业法人；

2）竞买人：即参加竞购拍卖标的的公民、法人或其它组织。

最高价竞买流程如图 6 所示，其流程如下：

1）首先，由拍卖人或主持人开始竞拍，并设置竞拍底价和竞拍结束时间。

2）竞拍期间竞买人随时可以进行出价，同时上交押金。如果出价大于目前最高价，记录为新最高价，押金放入资金池，并将之前最高出价者所交押金退回；否则，出价不大于目前最高价，出价失败，退回押金。

3）在竞拍时间结束后，拍卖人可以收取合约中最高出价的押金。

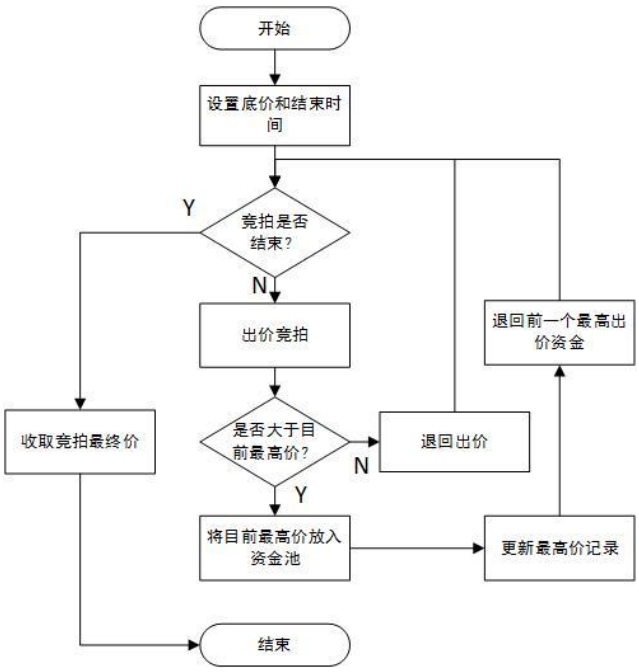

图 6 竞买流程

6 SPESC 编写竞买合约

在上述的流程中，用 SPESC 编写合约分为三个部分：

1）**合约当事人：**如前所述，竞买合约包含竞买人与拍卖人两方。首先，竞买人的定义如下：

```
party group bidders{
    amount : Money
    Bid()
    WithdrawBid ()
}
```

竞买人属于群体当事人，是由用户在执行竞拍操作 Bid 时表明参与竞买并注册成为竞买人。竞买人包含货币类型的属性 amount，用于记录竞买人的押金池，用于回收无效的出价。同时 WithdrawBid 声明了竞买人可以回收竞价失败所交的押金。

```
party auctioneer{
    StartBidding(reservePrice : Money, auctionDuration: Date)
    CollectPayment ()
}
```

拍卖人即拍品所有者属于个体当事人，声明了拍卖人可以执行的两个动作：开始竞拍 StartBidding 和结束竞拍 CollectPayment。在执行开始竞拍时，需要输入两个参数：底价 reservePrice 和竞拍时间 auctionDuration。

2）**合约中记录的附加信息：**本合约中主要需要记录三个变量：

```
highestPrice : Money
highestBidder : bidders
BiddingStopTime : Date
```

第一个变量为货币类型，记录了当前最高价，第二个变量为竞买人类型，记录了当前最高出价者，第三个变量为日期类型，记录了竞拍结束时间。

在此处定义的信息会被记录到区块链中，从而保证区块链不仅记录了信息当前的状态，还会记录合约执行过程中每一步执行后的历史状态。由于区块链数据具有不可篡改性和时序性，保证了智能合约状态的不可篡改性和可追溯性，智能合约与区块链结合的一方面优势就体现在这里。

3）**条款：**本合约中存在五条条款：

在第三节的竞买合约分析中可以看出，有三个需要当事人主动触发的过程，分别是①开始竞拍、②出价竞拍和③收取货款，其余流程可由程序自动完成。

由于任何账户可以注册为竞买人，且本文中以 Solidity 为目标语言，如第四节所述，如果直接向拍卖人发回资金是有安全风险的，让拍卖人自己取钱会更加安全。因此，在编写合约时添加收回押金条款。

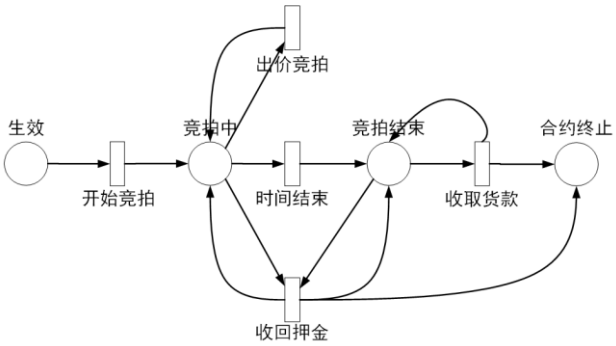

图 7 Petri 网表示的拍卖过程状态转移图

如图 7 所示，通过 Petri 网表示了拍卖过程的状态转移图，其中，合约分为四种状态和五种动作。状态包括：生效、竞拍中、竞拍结束、合约终止；动作包括开始竞拍、出价竞拍、收回押金、时间结束、收取货款。在五种动作中，时间结束是由计算机自动触发，其余四种为用户触发。根据四种动作，编写如下条款：

条款 1 发起竞拍条款定义如下：

```
term no1 : auctioneer can StartBidding,
    where highestPrice = reservePrice and
        BiddingStopTime = auctionDuration+ now.
```

拍卖人（auctioneer）有权触发动作发起竞拍

（StartBidding），在动作执行后，当前最高价（highestPrice）应为拍卖人输入的底价（reservePrice），结束时间（BiddingStopTime）应为当前时间（now）加上输入的竞拍持续时间（auctionDuration）。

条款 2 出价竞拍条款定义如下：

```
term no2 : bidders can Bid,
    when after auctioneer did StartBidding and
        before BiddingStopTime
    while deposit $ value > highestPrice
    where highestPrice = value and highestBidder = this bidder and
        this bidder::amount = this bidder::Origin amount + value.
```

竞买人（bidders）可以在拍卖人发起竞拍后，且在竞拍结束前，向合约转账（deposit）进行出价（Bid）。其中，在拍卖人发起竞拍后由 after auctioneer did StartBidding 表示，在竞拍结束前由 before BiddingStopTime 表示。如果出价（value）大于目前最高价（highestPrice），则出价成功；如果出价小于或等于最高价，则出价失败。

动作执行成功后，最高出价人的属性（this bidder::amount）中应记录了失败的出价总额，其中为方便表达，Origin 关键词表示动作执行前的值，合约当前最高价（highestPrice）与最高价出价人（highestBidder）应为本次出价（value）与出价人（this bidder）。

条款 3 回收押金分为两条子条款定义如下：

```
term no3_1 : bidders can WithdrawBid,
    when this bidder isn't highestBidder and this bidder::amount > 0
    while withdraw $this bidder::amount
    where this bidder::amount = 0.
```

如果竞买人（bidders）不是最高出价者，且当前合约中存有押金（this bidder::amount > 0），可以取回无效的竞价（WithdrawrBid）。条款执行成功后，该竞买人押金记录（this bidder::amount）应为 0。

```
term no3_2 : bidders can WithdrawBid,
    when this bidder is highestBidder and
        this bidder::amount > highestPrice
    while withdraw $this bidder::amount - highestPrice
    where this bidder::amount = highestPrice.
```

如果竞买人是最高出价者，且当前合约中存有该竞买人竞价失败的押金（this bidder::amount > highestPrice），可以取回无效的竞价。条款执行成功后，该竞买人押金记录应为最高价。

条款 4 结束竞拍条款定义如下：

```
term no4 : auctioneer can CollectPayment,
    when after BiddingStopTime
        and before auctioneer did CollectPayment
    while withdraw $highestPrice.
```

拍卖人（auctioneer）在竞拍时间结束后，且没有收取过货款，可以收取货款（CollectPayment）并将最高价货款取出（withdraw）。

7 目标代码生成

本节通过竞买合约的例子，讲述 SPESC 的目标代码生成方法，从而由编写好的 SPESC 合约自动生成 Solidity 代码。

7.1 目标语言合约框架

生成的 Solidity 合约分为两部分：

① **当事人合约**：由当事人定义生成的合约，

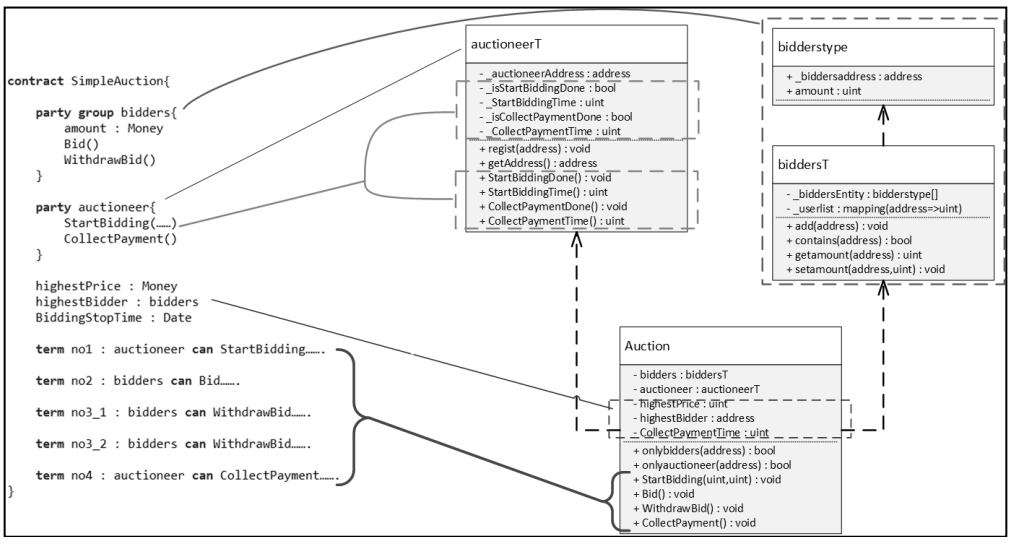

图 8 SPESC 与所生成合约类图对应关系

主要负责当事人人员的管理、关键事件的记录与统计和记录的查询功能；

② **主合约**：由其它部分生成的合约，主要包括变量的定义、修饰器（Modifier）以及条款生成的方法。

下面分别对这两个方面的生成过程进行阐述。

如图 8 所示，图中展示了由 SPESC 自动生成的 Solidity 语言竞买合约类图，并通过带颜色的线标明了 SPESC 元素与其对应关系。Solidity 语言竞买合约共分为三个部分，拍卖人合约、竞买人合约与竞买主体合约。

拍卖人合约中包含拍卖人的地址、人员的注册与查询方法以及对其所属的两个条款 StartBidding 与 CollectPayment 的记录。

竞买人合约包含由地址和 SPESC 中定义的变量 amout 组成的结构体、结构体数组、映射表、人员添加与查询方法以及 amount 的设定与获取方法。

竞买主体合约中包含当事人的定义、SPESC 中三条附加信息 *highestPrice*、*highestBidder*、*BiddingStopTime* 的定义以及根据五条条款生成的四个方法和两个修饰器（onlybidders、onlyauctioneer）。其中，条款 3.1 与 3.2 声明的是同一个动作 WithdrawBid，因此只生成一个方法。

此外，对于当事人的管理，除了基本的操作外，未被使用的管理操作会以注释的方式提供，如获取当事人群体列表、删除个体等，如果用户在方法的实现中需要使用，可以解除注释使用。而对于不参与时序控制的条款，如竞买合约中的出价竞拍（Bid）和取回押金（WithdrawBid），不会在合约中生成记录执行情况的属性与方法，实际执行情况仍可以在区块链中追溯，但不在合约中另行记录与处理。

7.2 当事人合约的生成

每个当事人都会对应生成一个当事人合约。由于当事人可分为个体当事人与群体当事人两类（见第 4 节），因此，当事人合约可分为个体当事人合约（Individual Party-Contract）或群体当事人合约（Group Party-Contract）两类。

个体当事人合约被用来规范一定权利或义务行为的个体，如买家、卖家对应的合约。群体当事人合约则用来规定拥有相同权利与义务行为的多个个体，如投票人、竞拍人对应的合约。群体当事人合约与个体当事人的区别在于：群体当事人合约包含数组及映射结构体及增删操作，使得群体当事人可以在合约运行中动态加入或退出。

两类当事人合约结构如图 9 所示。

参与方个体

个体变量	地址	
	成员属性	
	条款执行记录	
方法	人员管理	注册
		注销
		查询
	属性操作	获取
		设定
	条款执行管理	记录函数
		查询函数

参与方群体

个体变量	地址		
	成员属性		
	条款执行记录		
群体变量	数组<结构体>		
	映射<地址，数组坐标>		
方法	人员管理	增加	
		删除	
		查询	
	属性操作	获取	
		设定	
	条款执行管理	记录函数	
		查询函数	All
			Some
			This

图 9 当事人合约结构

情况 1：当事人个体生成的 Solidity 合约内容按类别分为三个部分。

1）当事人属性。当事人个体需要记录的内容包括：账户地址、成员属性、条款执行记录。账户地址作为账户的唯一身份标识，对应账户在区块链中的地址[1]。成员属性是在 SPESC 中由用户设定的需要记录的属性，并会在属性操作中生成对应的设定与获取方法。

条款执行记录 Record 生成规则如下：对由该当事人限定的条款集合 Φ 中的每个条款 t，

$$Record ::= \left\{ < \_is\_t\_Done, \_t\_Time >: \atop \forall t, t \in \Phi \right\},$$

其中，_t_ 表示取条款 t 的动作名，变量名以下划线起始是为了与用户定义变量区分。例如，对于条款 1 对应的动作 StartBidding，按上述规则将生成两个变量：*_isStartBiddingDone* 记录条款 t 是否执行完成与 *_StartBiddingTime* 记录条款 t 执行时间。

```
//attributes of action StartBidding
bool _isStartBiddingDone;
uint _StartBiddingTime;

//attributes of action CollectPayment
bool _isCollectPaymentDone;
uint _CollectPaymentTime;

address _auctioneerAddress;
```

图 10 拍卖人合约变量

[1] 账户在区块链中的地址通常是账户公钥的哈希。

如图 10 所示，竞买例子中拍卖人未在 SPESC 中定义属性，但有两个所属条款 StartBidding 和 CollectPayment，每个条款包括前述的两个属性，例如：*_isStartBiddingDone* 和*_StartBiddingTime*。此外，变量*_auctioneerAddress* 变量用于记录拍卖人地址。

2）人员管理。根据当事人是群体还是个体，生成对应管理方法。个体管理较为简单，包括注册、注销与查询三种方法。在竞买例子中拍卖人只涉及注册与查询拍卖人地址两种方法，如图 11 所示。

```
function regist(address a) public {
    _auctioneerAddress = a;
}

function getAddress() public view returns (address a){
    return _auctioneerAddress;
}
```

图 11 拍卖人的人员管理方法

3）条款执行管理。对于当事人的每条条款，生成条款执行记录方法，在条款执行完毕后记录完成时间；生成查询方法，返回条款完成时间。拍卖人的开始竞拍条款生成方法如图 12 所示。

```
function StartBiddingDone() public{
    _StartBiddingTime = now;
    _isStartBiddingDone = true;
}

function StartBiddingTime() public view returns (uint result){
    if(_isStartBiddingDone){
        return _StartBiddingTime;
    }
    return _max;
}
```

图 12 拍卖人条款执行管理方法

情况 2，当事人群体除当事人个体的内容外，还包含更丰富的当事人管理方法，以及多种记录的查询方式。具体生成规则如下：

1）当事人属性。对于当事人群体，为了使当事人可以可以遍历，采用结构体数组记录个体内容；同时为了方便用户通过地址查询用户个体信息，添加账户地址到数组坐标的映射表（Mapping table），如图 13 所示。其中数组中记录的用户个体变量与当事人个体的变量相同。

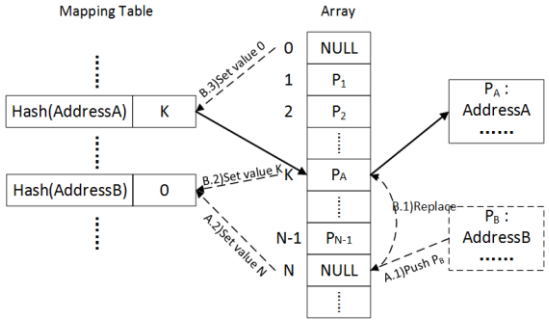

图 13 当事人群体人员管理

2）人员管理。当事人群体人员管理包括添加、删除、查询。

添加操作如图 13 虚线中 A.1 与 A.2 所示，例如，为了添加新的当事人个体 P_B，A.1）首先将个体信息插入数组最后一位；A.2）然后在映射表中记录该个体地址 AddressB 对应的数组坐标 N。

删除操作如图 13 虚线中 B.1 到 B.3 所示，例如，为了删除个体 PA，根据地址 AddressA 从映射中查询到其在数组的坐标 K 后，B.1）将数组最后一位 N 的个体 PB 替换掉 K 位的信息，并将第 N 位清空；B.2）将映射中记录的 PB 坐标替换为新坐标 K；B.3）将 PA 的坐标替换为初始值 0。如果删除的个体是数组中最后一位，则直接将数组和映射表对应位置设置为空即可。

在竞买合约中，竞买人涉及添加与查询操作，如图 14 所示。

```
function add(address a) public {
    _biddersEntity.push(bidderstype({_biddersaddress:a,amount:0}));
    _userlist[a] = _sum;
    _sum ++;
}

function contains(address a) public view returns (bool b){
    return _userlist[a] != 0;
}
```

图 14 竞买人的人员管理方法

3）条款执行管理。当事人群体生成的合约中，除当事人个体包含的记录方法外，在合约中记录第一个人与最后一个人的完成条款的时间，提供以下三种查询方式：

① All 查询：最后一个个体完成时间；

② First 查询：第一个个体完成时间；

③ This 查询：这个当事人个体完成时间。

如在投票合约中，主持人在所有人都投票以后才能统计，用 SPESC 表示如下：

term no1 : chairman **can** count,
　　when after all voters **did** vote.

则通过第一种方式查询时间。

7.3 主体合约生成

生成的主体合约主要分两部分。

第一部分是合约属性和当事人的定义与初始化，合约的属性就是用户在 SPESC 中定义的合约信息，当事人通过上一节定义的当事人合约类定义。其中，生成的变量默认访问权限为公开（public）类型，Solidity 中公开权限变量可以通过合约直接访问，从而获取合约状态。例如，竞买者可以查询合约中 *highestBidder*，获知最高出价人地址，而确定自己是否中标。

第二部分是条款的处理。SPESC 合约中的每个条款包含一个动作，在 Solidity 中对应生成一个方法。方法中包含执行条件检测、方法主体和执行结果检测三个部分，如果条件检测失败，程序会抛出异常并做相应处理。

在 Solidity 中抛出异常分为 require、assert 和 revert 三种。其中，require 关键词用于检测执行条件，如方法的输入或合约的状态等，如果检测不通过，以太坊平台会返还剩下的费用（gas）；而 assert 关键词用于检测程序的意外情况，在正确运行的程序中 assert 检测永远不会失败，一旦检测失败，意味着程序中存在错误，应该修改代码；而 revert 关键词与 require 类似，但可以通过与其它语句结合表达更复杂的情况。

条款的执行条件有三种限制：

1）当事人限制：条款规定有哪些当事人可以执行；

2）条件限制：SPESC 条款中的前置条件；

3）金额限制：交易操作中的 deposit 语句，调用该方法需要向合约存入的金额限制。

执行条件检测失败属于程序正常状态，应返还用户剩余费用，并回滚状态。因此，使用关键词 require 或 revert。

在条款所生成的方法最后会根据后置条件生成相应的执行结果检测，辅助用户编写方法主体逻辑，检验程序中的错误。一旦后置条件检测失败，意味着程序中存在错误，因此，使用 assert 关键词进行检测。

SPESC 编译规则将依据条款的后置条件推测生成方法所包含的主体内容，该内容作为编程人员的参考。同时，编程人员也需依照业务逻辑及其它细节检查、补充完成主体逻辑的设计。在这个过程中，已知方法输入和结果，用户只需要关注单个方法的实现，通过后置条件验证执行正确性即可。

在本例子当中，程序逻辑相对比较简单。因此，在自动生成后，只需添加竞买人注册代码后，检查代码逻辑没有问题，就可以直接运行。

如图 15 所示，SPESC 语言代码记为 A，所生成的 Solidity 代码记为 B。

1）根据 A 第 1 行中动作名 Bid 生成相同方法名（B 子图第 1 行，记为 B1）。如果检测到 A 中含有资产转移语句，如第 4 行的 while 关键词，则为方法添加 pAyABle 关键词，表明方法可以接收或发送以太币；

2）根据 A 第 1 行中 Bidders 的限定生成当事人检测。然而在本例中根据合约实际意图，出价的人即为竞买人，因此，需要手动去掉当事人限制，并添加当事人注册代码（B2-3）：如果出价者不是竞买人则注册成为竞买人；

3）根据 SPESC 中在拍卖人开始竞买后竞买时间结束前的前置条件（A2-3）与出价大于最高价的要求（A4）生成了两条执行要求（B4-5）；

4）根据 SPESC 中的后置条件（A5-7），自动生成了三条执行代码（B7-9）：记录了最高价、最高出价人和最高出价人的资金池，供编程人员参考；

4）根据后置条件（A5-7）生成了断言（B10-11）用于结果检测，其中 A 第 7 行中使用了 Ori 关键词，表示该变量在方法执行前的值，因此在方法主体前记录该变量（B6），以便于在检测时使用。

上述例子中，所生成的执行代码不需要修改，就可以正确运行，但如果将第二条条款的后置条件替换为如下语句：

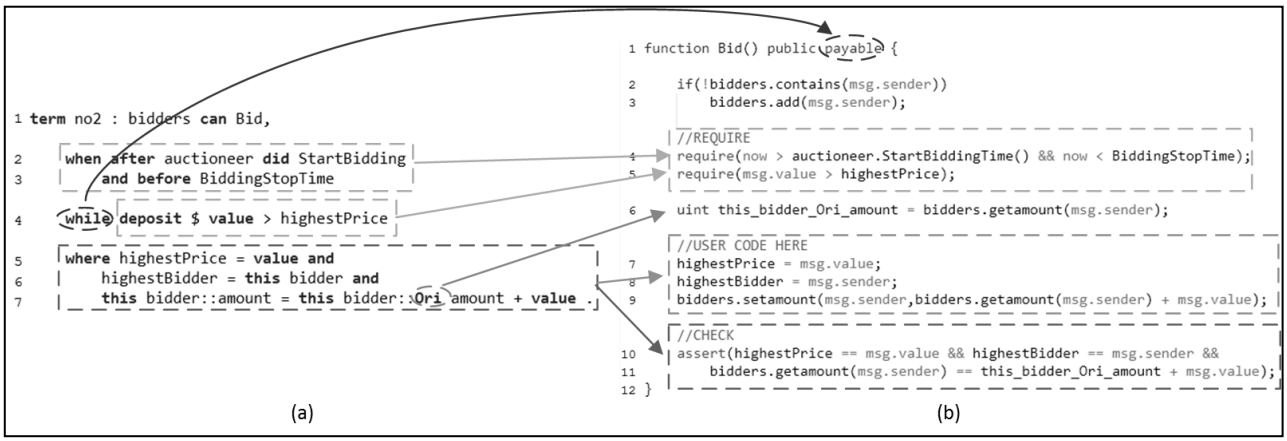

图 15 条款 2 对应关系

where this bidder::amount =
this bidder::**Ori** amount + highestPrice **and**
highestPrice = **value and** highestBidder = **this** bidder.

条款表达意思相同，但生成的方法主体会错误地将上一个最高价加入本次出价者的资金池中（断言可以成功检测出错误），需要手动调整程序所生成的语句顺序。

7.4 表达式实现

SPESC 当中共含有 5 类表达式：逻辑、关系、运算、常量和时间表达式。在 SPESC 语言的转化模型中，所有表达式都继承 Express 抽象类，增强了互操作性。

表 1 SPESC 表达式对应运算符及优先级

优先级	生成运算符	SPESC 表达式
1	.	ActionEnforcedTimeQuery
		ThisExpression
2	!	NotExpression
3	*、/	MultiplicativeExpression
4	+、-	AdditiveExpression
		TimeLine
5	>、>=、<、<=	RelationalExpression
		TimePredicate
6	==、!=	RelationalExpression
7	&&	AndExpression
		TimePredicate
8	‖	OrExpression
9	?:	ConditionalExpression
		ImplyExpression

如表 1 所示，表中展示了 SPESC 表达式对应的编程语言运算符，以及生成后的优先级。

SPESC 中的时间表达式类型分为时间点与时间段，其中类型为日期（Date）的常量与变量、动作完成时间表达式（ActionEnforcedTimeQuery）、全局时间表达式（GlobalTimeQuery）属于时间段；而类型为时间（Time）的常量与变量属于时间点。

时间线表达式（TimeLine）中，时间点与时间点不能进行加减运算，时间点与时间段运算结果为时间点，时间段与时间段运算结果为时间段。而时间谓词表达式（TimePredicate）的返回结果为布尔值，其生成规则如下：

after a	=>	now > a
a after b	=>	a > b
c after b	=>	now > c + b
within c after b	=>	(now > b) && (now < a + b)

其中，a 与 b 属于时间点，c 属于时间段，now 为当前时间。

8 实验及结果

针对 SPESC 语言及其转化规则，本节将通过两个实验分别测试 SPESC 语言所生成合约的易用性和 SPESC 合约转化可执行合约的有效性。

8.1 SPESC语言易用性实验

针对 SPESC 语言的特性，我们设计了一个问卷式对照实验，从阅读时间与准确性两方面评估 SPESC 合约是否比现有的智能合约更易于理解。

1）参加者。本次实验邀请了 15 名参与者，包括 9 名计算机科学系（CS）学生以及 6 名法学系（L）学生。15 名参与者被随机分为两组（即 GA 和 GB）。GA 由四名 CS 学生（即 GA-CS）和三名法律学生（即 GA-L）组成。GB 由 5 名 CS 学生（即 GB-CS）和 3 名法律学生（即 GB-L）组成。

2）智能合约。本次实验所用智能合约包含“借贷合约”与“竞买合约”两种，其中，借贷合约（Lending-SO）改编自 Github 上的开源 Solidity 程序[2]。竞买合约（Auction-SO）改编自 Solidity 文档中描述的官方示例[3]。作为对照，将上述两种 Solidity 合约编写为 SPESC 合约形式，被称为 Lending-SP 和 Auction-SP。

3）问卷与数据收集。实验采用针对不同合约的问卷回答形式，每个人得到 2 份问卷，分别是借贷问卷、竞买问卷。试卷包括 10 个问题，采用标识 Qx.y 表示问卷 x 中第 y 个问题，具体问题如下：

$Q_{1.1}$ 谁能执行 confirm 函数？ (SS)
$Q_{1.2}$ 借款的条件是什么？(SS)
$Q_{1.3}$ 借款会造成什么影响？(SS)
$Q_{1.4}$ 还款会造成什么影响？ (SS)
$Q_{1.5}$ 借贷者的义务有哪些？ (MS)
$Q_{2.1}$ 什么条件下可以参与竞拍？ (SS)
$Q_{2.2}$ 如何取回自己失败的竞价？ (SS)
$Q_{2.3}$ 目前最高出价者可取回之前失败竞价吗？(SS)
$Q_{2.4}$ 结束竞买会造成什么影响？ (MS)
$Q_{2.5}$ 竞买者有哪些权利？ (MS)

其中 SS，MS 分别表示单选和多选。

4）实验过程。首先，我们对所有参与者在前一个小时讲解了 SPESC 和 Solidity 的基本语法和语义；其次，通过购买合约的两种语言示例回答参与者的问题；然后，要求 GA 和 GB 分别阅读

[2] https://github.com/terzim/dapp-bin
[3] https://solidity.readthedocs.io/en/develop/solidity-by-example.html#simple-open-auction

Lending-SP 和 Lending-SO，并填写相关借贷合约的问卷，我们对每个参与者答题时间进行记录；最后，要求 GA 和 GB 分别阅读 Auction-SO 和 Auction-SP，并填写有关竞买合约的问卷并记录时间。

5）实验结果。在上述实验过程中，我们从回答过程与回答结果中统计了作答平均时长 T^G_Q 与平均准确率 P^G_Q，其中，G 表示组别，Q 表示问题。实验结果如表 2 和表 3 所示。

表 2 问卷 1 统计（借贷合约）

	GA-CS	GA-L	GA	GB-CS	GB-L	GB
T^G_{Q1}	762.8	722.3	745.4	1551.2	1193.7	1439.6
$P^G_{Q1.1}$	100%	100%	100%	20.0%	66.7%	37.5%
$P^G_{Q1.2}$	100%	100%	100%	80.0%	0.0%	50.0%
$P^G_{Q1.3}$	75.0%	100%	85.7%	60.0%	66.7%	62.5%
$P^G_{Q1.4}$	75.0%	100%	85.7%	0%	0%	0%
$P^G_{Q1.5}$	83.5%	89.0%	85.9%	63.4%	66.7%	64.6%
$\overline{P^G_{Q1}}$	83.9%	88.7%	86.0%	39.6%	39.2%	39.4%

表 3 问卷 2 统计（竞买合约）

	GA-CS	GA-L	GA	GB-CS	GB-L	GB
T^G_{Q2}	1264.0	1437.3	1338.3	664.2	756.0	698.6
$P^G_{Q2.1}$	50.0%	33.3%	42.9%	60.0%	0.0%	37.5%
$P^G_{Q2.2}$	50.0%	33.3%	42.9%	100%	33.3%	75.0%
$P^G_{Q2.3}$	25.0%	66.7%	42.9%	100%	66.7%	87.5%
$P^G_{Q2.4}$	75.0%	55.6%	66.7%	100%	100%	100%
$P^G_{Q2.5}$	95.0%	58.3%	79.3%	100%	100%	100%
$\overline{P^G_{Q2}}$	64.5%	43.0%	55.3%	91.7%	50.0%	76.0%

从表 2 和表 3 结果可知：对于借贷合约，阅读 SPESC 合约的 GA 完成速度比 Solidity 的 GB 快（前者用时是后者 0.52 倍），答题准确率更高（提高 1 倍多）；对于竞买合约，阅读 SPESC 合约的 GB 完成速度同样比 Solidity 的 GA 快（前者用时是后者 0.52 倍），且答题准确率更高（提高约 0.4 倍）。

8.2 SPESC语言转化实验

针对 SPESC 合约转化有效性，我们将通过实验的方式验证第 7 节方法生成的 Solidity 智能合约。下面介绍从编写 SPESC 到执行完成流程中的实验环境、实验步骤及预测结果、以及实验结果与分析。

1）SPESC 语言模型与生成器

SPESC 包含 70 条语言模型与 64 条语法规则，以及一千多行的生成器代码，最终形成一个 Eclipse 插件，通过插件编写 SPESC 代码并生成目标代码。SPESC 的语法和生成器通过 EMF 与 Xtext 实现，其中，EMF 是一个建模框架和代码生成工具，Xtext 是一个开发程序语言和特定领域语言的框架。

2）区块链测试平台

实验系统运行在三个 windows7 系统的虚拟机下，区块链采用以太坊平台，以太坊 Geth 客户端版本为 1.7.0-stable。创世区块的参数如下：

```
{
  "config": {
      "chainId": 15,
      "homesteadBlock": 0,
      "eip155Block": 0,
      "eip158Block": 0
  },
  "coinbase" : "0x0000000000000000000000000000000000000000",
  "difficulty" : "0x10000",
  "extraData" : "",
  "gasLimit" : "0xffffffff",
  "nonce" : "0x0000000000000042",
  "mixhash" : "0x0000000000000000000000000000000000000000000000000000000000000000",
  "parentHash" : "0x0000000000000000000000000000000000000000000000000000000000000000",
  "timestamp" : "0x00",
  "alloc": { }
}
```

图 16 创世块参数

实验中三个虚拟机共部署 3 个节点，分别是一个拍卖人节点 A 和两个竞买人节点 B1 与 B2。每个节点的主要账户初始状态如表 4 所示：

表 4 账户状态

账户	A	B1	B2
地址	0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c	0x14723A09ACff6D2A60DcdF7aA4AFf308FDDC160C	0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB
余额	100eth	100eth	100eth

3）编译环境

Solidity 合约需要编译为机器代码才能在以太坊虚拟机中运行，本文使用以太坊的 Remix 编译器，Remix 是一个开源工具，包含编写合约、测试、调试和部署的功能。Remix 用 JavaScript 编写，支持在浏览器和本地使用。

4）合约测试

合约的测试流程如下，其中查询步骤由于不影响合约状态、不造成任何开销且随时可以执行而被省略：

步骤 1，由 A 部署合约，此时合约中拍卖人由 A 注册，只有 A 可以执行方法 StartBidding，A 无法执行其它方法，其它账户也无法执行任何方法。

步骤 2，A 执行 StartBidding 方法开始竞拍，并设置底价为 2eth，设置结束时间为执行后 5 分钟（以太坊中的时间每生成一个区块更新一次，执行时间为当次区块时间，以太坊平均 15s 生成一个区块，

因此可能存在少量误差）。此时，StartBidding、WithdrawBid、CollectPayment 方法不能被执行。任何账户可以执行 Bid 方法进行出价，但出价若少于或等于 2eth，执行失败。

步骤 3，B1 执行 Bid 方法出价参与竞拍，出价 3eth。此时，最高出价者为 B1，最高价为 3eth，B1 的资金池里有 3eth，方法可执行情况与步骤 2 相同。

步骤 4，B2 执行 Bid 方法出价竞拍，出价 4eth。此时，最高出价者为 B2，最高价为 4eth，B2 的资金池里有 4eth。所有账户可以执行 Bid 方法，B1 可以执行 WithdrawBid。

步骤 5，B1 执行 WithdrawBid 收回押金，然后再次执行 Bid 出价竞拍，出价 5eth。此时，最高出价者为 B1，最高价为 5eth，B1 的资金池有 5eth，B2 资金池有 4eth。

步骤 6，等到竞拍时间结束。此时，只有 A 可以执行 CollectPayment，B2 可以执行 WithdrawBid。

步骤 7，A 执行 CollectPayment 收取货款，B2 执行 WithdrawBid 收回押金（不分先后顺序）。合约结束，合约中没有资金，没有方法可以执行。

5）实验结果与分析

表 5 运行情况

当事人	操作	参数	账户余额/eth	总消耗 gas	执行消耗 gas	存储消耗 gas
拍卖人 A	部署（初始化）		A:99.9; B1:100.0; B2:100:0	2 124 040	1 577 980	546 060
拍卖人 A	StartBidding	底价:2eth; 时间:300s	A:99.9; B1:100.0; B2:100:0	110 737	89 017	21 720
竞买人 B1	Bid	存入:3eth	A:99.9; B1:96.9; B2:100:0	156 552	135 280	21 272
竞买人 B2	Bid	存入:4eth	A:99.9; B1: 96.9; B2:95.9	141 552	120 280	21 272
竞买人 B1	WithdrawBid		A:99.9; B1:99.9; B2: 95.9	44 361	38 089	6 272
竞买人 B1	Bid	存入:5eth	A:99.9; B1:94.9; B2:95:9	84 480	63 208	21 272
拍卖人 A	CollectPayment		A:104.9; B1:94.9; B2:95:9	81 861	60 589	21 272
竞买人 B2	WithdrawBid		A:104.9; B1:94.9; B2:99:9	44 361	38 089	6 272

经过实际运行测试，测试结果与上述流程中预测的方法可执行情况、合约状态相符。账户余额与执行消耗 gas 情况如表 5 所示，其中执行消耗 gas 指的是智能合约程序在以太坊虚拟机中执行所需消耗的 gas，即程序实际执行的步骤。存储消耗 gas 指的是更改区块链中数据所需消耗的 gas，即将上传到区块链的变量的更改。Gas 所消耗的以太币为 $cost = gasUsed \times gasPrice$ ，其中 gasPrice 为 gas 单价，由合约用户在执行合约时设置，单价的高低影响矿工处理的优先级。

自动生成的 Solidity 智能合约中，程序员仅对方法内的代码检查与修改，而合约的接口、变量、修饰器、合约间的关系等程序结构由生成器自动生成。因此，SPESC 规范化了 Solidity 合约的结构。

上述实验以竞买合约为例展示了通过 SPESC 编写、生成、运行的完整流程。上述实例表明，合约既可以通过时序逻辑表示描述执行流程，也可以通过变量记录的合约状态描述，因此，该流程对于其它合约同样适用，如买卖、竞买、借贷、投票等。但对于合约中出现的除以太币外其它资产，如自定义代币、买卖的货物等，只能通过变量记录，未来可以增加对资产的定义及操作。

9 总结与展望

本文提出了 SPESC 的生成规则，通过 SPESC 编写智能合约，可以自动生成人员管理，提供便捷的操作与查询接口，用户不需要考虑当事人存储结构，但增加了适用性的同时也会稍微增加运算开销；可以根据条件生成时序控制，记录条款执行情况；SPESC 无法保证生成正确的函数体，但可以根据后置条件生成方法结果检测，辅助编写与检测程序。

为了进一步改善 SPESC 语言，后续工作将包括以下方面：首先，为验证 SPESC 编写合约的正确性，目前已有很多通过形式化验证合约正确性的研究，如前文提到的文献[3][4][5][6][7]等，未来可以在已有的 SPESC 语言模型基础上，建立形式化表示，通过形式化的方法，验证合约条款的前置、后置条件，以及条款间时序的正确性，为用户提供

形式化分析工具。其次，对于生成的 Solidity 目标代码正确性，目前已有一些分析或检测漏洞的研究，如文献[17][18][19][20]等，未来研究可以根据这些研究继续改进生成的目标代码，优化程序结构和规范，增强合约安全性和正确性。

致 谢 感谢何啸老师在可读性实验中提供的帮助，感谢实验室所有同学的帮助。

参考文献

[1] Linnhoff-Popien C, Schneider R, Zaddach M. Digital Marketplaces Unleashed. Berlin,Germany:Springer, 2018.

[2] Szabo N. Smart contracts: building blocks for digital markets. The Journal of Transhumanist Thought, 1996,(16): 18-20.

[3] Schrans F, Eisenbach S, Drossopoulou S. Writing safe smart contracts in flint. Proceedings of Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming. New York, USA, 2018: 218-219.

[4] Coblenz M. Obsidian: A Safer Blockchain Programming Language. Proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C). Buenos Aires, Argentina, 2017: 97-99.

[5] Idelberger F, Governatori G, RIVERET, Regis, et al. Evaluation of logic-based smart contracts for blockchain systems. 2016, 9718: 167-183.

[6] Sergey I, Kumar A, Hobor A. Scilla: a smart contract intermediate-level language. ArXiv preprint arXiv: 1801.00687, 2018.

[7] Frantz C K, Nowostawski M. From institutions to code: Towards automated generation of smart contracts. Proceedings of 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W). Tucson, USA, 2016: 210-215.

[8] Kasprzyk K. The Concept of Smart Contracts from the Legal Perspective. Review of Comparative Law, 2018, 34(3).

[9] Goldenfein J, Leiter A. Legal Engineering on the Blockchain: ‘Smart Contracts’ as Legal Conduct. Law and Critique, 2018, 29(2): 141-149.

[10] Gomes S S. Smart Contracts: legal frontiers and insertion into the Creative Economy. Brazilian Journal of Operations & Production Management, 2018, 15(3): 376-385.

[11] Allen J G. Wrapped and Stacked: ‘Smart Contracts’ and the Interaction of Natural and Formal Language. European Review of Contract Law, 2018, 14(4): 307-343.

[12] O’Connor R. Simplicity: A New Language for Blockchainsthe. 2017 Workshop. Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security - PLAS’17. New York, USA, 2017: 107-120.

[13] Regnath E, Steinhorst S. SmaCoNat: Smart Contracts in Natural Language. Proceedings of the 2018 Forum on Specification & Design Languages (FDL). Garching, Germany 2018: 5-16.

[14] Choudhury O, Rudolph N, Sylla I, et al. Auto-Generation of Smart Contracts from Domain-specific Ontologies and Semantic Rules. Pproceedings of the 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). Nova Scotia, Canada 2018: 963-970.

[15] Biryukov A, Khovratovich D, Tikhomirov S. Findel: Secure Derivative Contracts for Ethereump. Financial Cryptography and Data Security. FC 2017: 453-467.

[16] Xiao He, Bohan Qin, Yan Zhu, et al. SPESC: A Specification Language for Smart Contracts. Proceedings of the 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC). Tokyo, Japan 2018, 132-137.

[17] Luu L, Chu D H, Olickel H, et al. Making Smart Contracts Smarter. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna, Austria, 2016: 254 - 269.

[18] Atzei N, Bartoletti M, Cimoli T. A Survey of Attacks on Ethereum Smart Contracts (SoK). International Conference on Principles of Security and Trust. Berlin,Germany: Springer, 2017, 10204: 164-186.

[19] Bhargavan K, Swamy N, Santiago Zanella-Béguelin, et al. Formal Verification of Smart Contracts: Short Paper. Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. 2016: 91-96.

[20] Parizi R M, Amritraj, Dehghantanha A. Smart Contract Programming Languages on Blockchains: An Empirical Evaluation of Usability and Security. Proceedings of 2018 International Conference on Blockchain (ICBC 2018). Springer, Cham, 2018.

附录 A

```
contract SimpleAuction{

    party group bidders{
        amount : Money
        Bid()
        WithdrawOverbidMoney()
    }

    party auctioneer{
        StartBidding(reservePrice : Money, biddingTime : Date)
        StopBidding()
    }

    highestPrice : Money
    highestBidder : bidders
    BiddingStopTime : Date

    term no1 : auctioneer can StartBidding,
        when before auctioneer did StartBidding
        where highestPrice = reservePrice and BiddingStopTime = biddingTime + now.

    term no2 : bidders can Bid,
        when after auctioneer did StartBidding and before BiddingStopTime
        while deposit $ value > highestPrice
        where highestPrice = value and highestBidder = this bidder and
            this bidder::amount = this bidder::Ori amount + value .

    term no3_1 : bidders can WithdrawOverbidMoney,
        when this bidder isn't highestBidder and this bidder::amount > 0
        while withdraw $this bidder::amount
        where this bidder::amount = 0.

    term no3_2 : bidders can WithdrawOverbidMoney,
        when this bidder is highestBidder and this bidder::amount > highestPrice
        while withdraw $this bidder::amount - highestPrice
        where this bidder::amount = highestPrice.

    term no4 : auctioneer can StopBidding,
        when after BiddingStopTime and before auctioneer did StopBidding
        while withdraw $highestPrice.
}
```

图 17 SPESC 编写的竞买合约

ZHU Yan, 1974, PH.D. Professor. His research intersts include Information security and cryptography, secure computing, blockchain, mobile cloud computing.

QIN Bo-Han, master student. His research interests include blockchain, smart contract.

CHEN E, Ph.D. candidate. Her research interests include cryptography and network security.

LIU Guo-Wei, Senior Engineer. His research intersts include big data, security system, security management and standard.

Background

Blockchain is a hot topic in recent years, and smart contracts, as the core of the second-generation blockchain, are widely deployed in decentralized application, and are connected to the fields of computer, finance, law and so on. At present, there are some researches on how smart contracts are expressed from a natural language-like perspective, some researches provides a new framework of automatic generating, and some researches provide formal methods to verify the correctness of smart contracts, but there is no complete solution from syntax representation, code generation to execution.

This paper focuses on the lack of conversion methods between advanced smart contract languages and executable smart contract languages. Based on the previous SPESC research, this paper designs a three-layer structure of the smart contract system, including advanced smart-contract layer, basic smart-contract layer, and executable machine-code layer. Then we provide the conversion rules of the advanced smart contract language to the smart contract language, and demonstrates it through examples. Finally we veifie the legibility of SPESC and the correctness of the conversion process through two case studies.

This work was supported by the National Key Technologies R&D Programs of China (2018YFB1402702), the aim of which is to design and develop smart contract service based on blockchain for researching on theory and technology of modern service trust transaction. This work was also partly supported by the National Natural Science Foundation of China (61972032).