中　国　电　子　学　会　标　准

JH/CIE 194—2021

区块链 智能合约 形式化设计与验证方法

Formal design and verification method of blockchain smart contract

（征求意见稿）

（本稿完成时间：2022 年 1 月 26 日）

在提交反馈意见时，请将您知道的相关专利连同支持性文件一并附上。

XXXX-XX-XX 发布　　XXXX-XX-XX 实施

中国电子学会 发布

目 次

前　言

本标准依据 GB/T 1.1-2020《标准化工业导则 第 1 部分：标准化文件的结构和起草规则》的规则起草。

本标准由北京航空航天大学牵头提出。

本标准由中国电子学会区块链分会技术归口。

本标准起草单位：北京航空航天大学，工信部区块链技术与数据安全重点实验室，云南省区块链应用技术重点实验室，北京科技大学，北京物资学院，北航云南研究院，常州唯实智能物联创新中心有限公司，苏州链约科技有限公司，中数南粤（广州）信息技术有限公司等（公开征集中）

本标准主要起草人：胡凯，李洁，朱健，李洋等。

区块链 智能合约 形式化设计与验证方法

1 范围

本文件规定了使用形式化设计区块链智能合约要求，描述了对应的智能合约验证方法。

本文件适用于采用形式化方法设计、开发可信通用区块链智能合约的应用场景。

2 规范性引用文件

下列文件对于本文件的应用是必不可少的。凡是注日期的引用文件，仅注日期的版本适用于本文件。凡是未注日期的引用文件，其最新版本（包括所有的修改单）适用于本文件。

GB/T 22032-2021 系统与软件工程 系统生存周期过程

GB/T 11457-2006 信息技术 软件工程术语

中国电子学会标准 T/CIE 095—2020 区块链 智能合约 形式化表达

3 术语和定义

下列术语和定义适用于本文件。

3.1

需求分析 demand analysis

创建一个新的或改变一个现存的系统或产品时，确定新系统的目的、范围、定义和功能时所要做的所有工作。

3.2

区块链智能合约 blockchain smart contract

部署在区块链上并可在满足预定合约条款和履行条件时自动执行的计算机代码。

3.3

形式化设计方法 formal design methods

包括智能合约的形式化建模、形式模型的一致性转换、形式模型的验证、可执行代码的自动生成以及一致性测试，使用形式化方法涵盖了智能合约全生命周期的设计与验证过程，以保证智能合约需求设计与描述的正确性、智能合约代码的安全性、智能合约功能属性和非功能属性的可靠性，并通过代码的自动生成提高开发效率，通过设计迭代提高可维护性。

3.4

领域特定语言 domain-specific language

某个应用程序领域的计算机语言，面向某些智能合约应用场景的计算机描述语言。

3.5

智能合约语言 smart contract language

一种为实现智能合约而创建的编程语言，包含规范智能合约撰写的语法规则和语义，具有图灵完备性。

3.6

形式化方法 formal methods

基于数学的特种技术，适合于软件和硬件系统的描述、开发和验证。

3.7

形式化建模 formal modeling

借助数学的方法通过形式化的语言，对现实世界的需求进行描述，建立抽象而精确的模型的过程。

3.8

形式化模型转换 formal model transformation

采用不同的形式化语言对应用场景需求建立不同的形式化模型进行相互转换的技术。

3.9

模型精化 model refinement

基于规约，通过可推理证明的步骤，从抽象到具体逐步实现模型的细节和属性一致性的方法。

3.10

形式化验证 formal verification

根据某个或某些形式规范或属性，使用数学的方法证明其正确性或非正确性，如不正确，有时可以给出反例。

3.11

定理证明 theory proving

将“系统满足其规约”这一论断作为逻辑命题，通过一组推理规则，以演绎推理的方式对该命题开展证明。

3.12

模型检测 model checking

通过显式状态搜索或隐式不动点计算来验证有穷状态并发系统的命题性质。

3.13

自动代码生成 automatic code generation

生成程序的程序，是指通过读取工程中设计的各种文档或者模型，按照一定的领域规则，生成规范的计算机能理解的源代码的过程。

3.14

一致性测试 conformance testing

检验所实现的协议实体（或系统）与协议规范的符合程度（而验证则是检查形式化规范的内部一致性），即测试一个协议给定实现的外部行为是否符合协议的规范。

4 符号

下列符号适用于本文件。

@@	面向法律合同的意思表示
?	前置关键词任选
{ }	包含语句集合
.	语句结束符
+	零条或多条语句
//	注释符
‘’, “”	字符串类型
=, is	相等
!=, <>,isn’t	不相等
Not	否定
all, for all	全部
When	当，表示前置条件
Then	执行
some, exits，∃	存在
any，∀	任意
and, or，|	用于连接两种情况，表示和（或）的关系
≜，:=	定义，变化为
，空格	用于表示同级元素的并列与分割

5 概述

需求分析帮助开发者明确业务目标。领域特定语言是面向具体应用场景，且面向用户友好设计的一种描述性语言。领域特定语言需要向形式化模型转换，用于验证模型的正确性，同时在单个模型不能满足表达能力的时候，需要使用模型转换的方式，使用多个形式化模型互补验证。使用对应的形式化验证工具验证模型，通过反馈修正形式化模型。自动代码生成可以自动或半自动生成智能合约语言。智能合约语言是一种可实现在区块链智能合约环境中执行代码的编程语言。从需求描述出发，通过建模、模型转换和自动代码生成方式，可以最终生成对应智能合约语言的可执行代码。

5.1 架构

本文件适合区块链智能合约的形式化设计与验证一体化过程阶段见图 1。自顶向下的过程是设计与验证的一体化方法，针对已有智能合约代码，也可以反向从智能合约代码建立形式化模型进行验证过程。

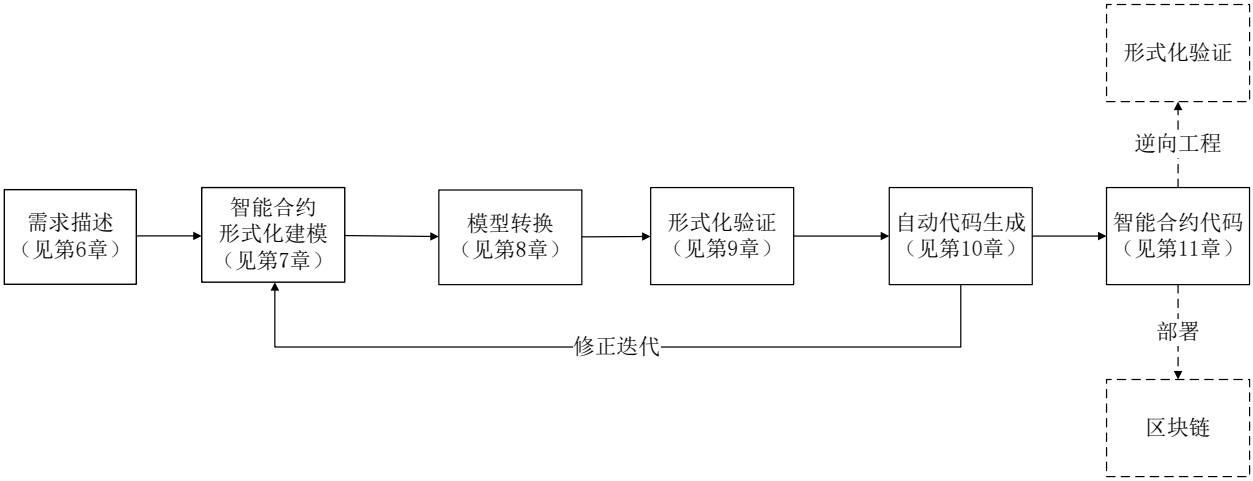

图 1 智能合约形式化设计与验证架构

5.2 关联

图 1 各环节之间的关联如下：

a) 智能合约需求描述是对智能合约的需求进行规约和形式化描述。领域专家可以使用 DSL（Domain-Specific Language）对智能合约的需求进行规范描述。需求设计可以包括合约对可靠性、安全性、可监管性、法律规制等行业特殊属性的描述。使用 DSL 也是需求描述的一种推荐形式，可以通过模型转换技术，转为可执行智能合约代码。DSL 可采用以下几种方式：
——基于一阶逻辑的领域特定语言：描述基本模型中的事件对应于公平离散系统模型中的状态之间的转移，其定义包括变量、证明义务和定理、事件、事件组、运算符。参见 6.4.2。
——面向法律合同的智能合约语言：宜符合 T/CIE 095-2020，由现实法律合同转化为区块链智能合约平台程序代码的过渡性语言，符合现实合同的法律特征和易理解性，也有计算机程序代码的规范性。法律合约语言包括合约头（Heading）、 账户（AccountSection）、资产（AssetSection）、 协议（AgreementSection）、事件（EventSection）等部分，语法推荐参见 6.4.3。

b) 智能合约形式化建模：将智能合约的开发需求按照一定方式、特殊语言等形式表现，并采用形式化语言对该需求进行建模，模型具有可验证性。参见第 7 章。
c) 模型转换：通常单个形式化的方法或工具无法验证复杂模型所有的属性，因此存在针对不同验证属性的形式化语言、模型和验证工具，常常因为需要从一种形式化模型转移到另外一种形式化模型，然后利用对应的不同形式化验证工具对特定属性加以验证，模型转换是形式化验证方法必要的和常用的方法。参见第 8 章。
d) 形式化模型验证：通过定理证明、模型检测、符号证明等形式化方法进行模型验证，根据验证结果，可对模型进行优化循环迭代。参见第 9 章。
e) 自动代码生成与一致性测试：基于建立的形式化模型，通过自动代码转换工具即可生成可执行合约代码，目标生成语言如 Solidity、Java、Golang 等。首先应定义实现保持语义一致的转换规则，然后基于转换规则开发自动转换器，用于辅助生成可执行合约代码，从而建立形式化模型与可执行合约之间的桥梁。自动代码生成参见 10.1。一致性测试方法可以决定测试集的产生、测试执行系统的结构和描述方法，参见 10.3。其中，ISO/IEC 9646-1:1994 定义了四种抽象测试方法，参见 10.3.4。第 10 章用到的缩写如下：
——上测试器（UT，Upper Tester）
——下测试器（LT，Lower Tester）
——控制观察点（PCO，Point of Control and Observation）
——抽象服务原语（ASP，Abstract Service Primitives）
注：在一致性测试过程中，UT 或者 LT 通过 PCO 和 IUT 交换 ASP，以观测系统行为。
f) 智能合约代码：通过以上步骤，最终生成可执行智能合约代码，参见第 11 章。

6 智能合约需求描述

6.1 统一建模语言

在需求模型中，通过用例图从外部用户的角度来捕获系统、子系统或类的行为。在对象结构模型中通过包图、类图和对象图定义系统对象及对象间的静态关系。在行为模型中，通过顺序图、合作图和状态图描述对象间的交互关系、对象的生命周期以及生命周期中对象可能存在的状态以及状态间的转换约束。在体系结构模型中，通过组件图和配置图描述软件体系结构、硬件体系结构以及通信机制。

6.2 智能合约需求描述的分类

6.2.1 公理性描述

定义系统应满足性质的集合，通过说明程序或软件系统应有的性质来描述一个程序或软件系统。其中每个性质都可以看作一个公理，系统或程序规约就是这些公式或公理的合取，规约之间的求精关系表示为较低层的规约的性质蕴含较高层规约中相对应的性质。

示例：时序逻辑方法。

6.2.2 操作性描述

以转移系统为模型来描述软件系统或程序，这样在程序和规约之间的演化就可以形成无缝连接。系统由以下组成：一组系统变量的集合，一组状态之间的转移集合，一组初时条件集合，一组公平描述集

合。

6.3 需求工程的形式化

在程序的开发方法尤其是面向对象的开发方法中，把一些公用的系统或子系统的需求模型模板化，把与业务逻辑无关的数据从需求模型中分离出去。用到时给模板赋以具体的参数即可实例化不同业务系统的需求模型。模板模型在构造时已经进行了形式化验证，引用时不需对其本身验证，可以节约一部分构建系统需求模型的时间。

6.3.1 自然语言与形式化

自然语言或图表描述软件需求，再翻译成一种形式化方法描述的软件需求模型，对需求模型进行一致性和完备性的验证，对不一致的地方进行修正，直到需求模型验证通过为止。

6.3.2 与其他建模工具结合

利用需求建模工具对需求进行建模，在文本描述的基础上构建软件的需求模型。

示例：UML。

6.3.3 轻量级形式化

针对资源有限、工期紧张、迭代频繁、可靠性要求高的系统，对原有的全生命周期形式化方法进行剪裁，将形式化方法应用于软件开发的某一阶段或某一部分，平衡软件质量和生产率之间的矛盾。

6.4 领域特定语言描述

6.4.1 DSL 设计

DSL 语言作为描述特定属性的自然语言，属于规约语言。对 DSL 的设计宜符合以下要求：

a) 限制自定义命名：限制别名的使用，使命名建立在共同理解基础上；

b) 限制嵌套：限制逻辑结构的嵌套；

c) 对代码结构进行划分：分离数据声明和操作声明，每个部分允许某些特定语言结构；

d) 预定义类型系统：预定义数据结构或限制数据类型；

示例：使用变量时，“温度”与“速度”并不相同，即使两者都可以用实数表示，应在出现时对其数据结构进行限制。

e) 选取特殊的关键字和语法规则：选用与上下文不敏感的关键字，支持建立与编程语言的映射。

6.4.2 基于一阶逻辑的领域特定语言

6.4.2.1 变量（variable）

变量的值可以发生改变，由变量约束和变量行为组成：

$$\text{Variable} = \{\text{Invariant}, \text{Axiom}\}$$

式中：

——Invariant 表示变量的类型等约束条件，又称不变式，在程序执行过程中应总能成立，不变式是基于谓词逻辑的命题，可以形成证明义务；

——Axiom定义了变量行为，也可描述由公理导出的性质。

示例：变量的表述方式可有以下几种：

a) 下式表述 x 的值变为 e：

$$x \coloneqq e$$

b) 下式表示 x 应当满足谓词 P：

$$x: |P$$

c) 下式表示 x 是集合 s 中的元素：

$$x: \in s$$

6.4.2.2 **证明义务和定理**（proof obligation and theorem）

证明义务包括常规属性和自定义属性，定义变量或事件时，通常证明工具会生成部分证明义务，如果需要更多的自定义属性，可以在模型中以如下的方式添加：

$$(\exists S, C \cdot A)$$

式中 S 表示定义的集合，C 表示定义的常数。A 表示公理 $a1$,$a2$,…an，则上式也可以改写为：

$$(\exists S, C \cdot a1 \wedge a2 \wedge \ldots \wedge an)$$

定理表示证明义务满足时，可以推导出的结果，如下式表示：

$$(\forall S, C \cdot A \Rightarrow Tc)$$

上式也可表示为如下方式：

$$\left(\forall S, C \cdot A \Rightarrow t1\right), \ldots (\forall S, C \cdot A \Rightarrow tn)$$

6.4.2.3 **事件**（event）

同一模型内事件的名称应唯一，由前置条件和动作组成：

$$\text{Event} = \{\text{Guards}, \text{Actions}\}$$

前置条件（Guards）由一组谓词组成，表示事件发生所必需的条件，用$G(v)$表示，式中 v 代表变量；动作（Actions）表示事件发生后所必需完成的状态变量的修改方式，用$S(v)$表示。具体 DSL 语言中事件可表示如下：

$$E \triangleq \textbf{当}\ G(v)\ \textbf{执行}\ S(v)\ \textbf{结束}$$

初始化事件不含有前置条件：

$$Init \triangleq \textbf{执行}\ S(v)\ \textbf{结束}$$

6.4.2.4 **事件组**（event group）

当有多个事件时，可以写为事件组的形式：

$$E \triangleq \textbf{任意}\ i\ \textbf{当}\ C(i)\ \{\textbf{当} G(i,v)\ \textbf{执行}\ S(i,v)\ \textbf{结束}\}\ \textbf{结束}$$

其中 $C(i)$为条件数组，条件成立时执行对应的事件。

用事件模型描述系统的需求模型，并对模型中描述的软件系统的相关性质进行验证，以确保系统的

需求模型的可靠性和完备性。

6.4.2.5 **运算符**（operator）

表示变量之间的大小关系：T $[e1]$ (‘ < ’|’ > ’|’ <= ’|’ >= ’)$T[e2]$
表示变量之间是否相等：T$[e]$ (‘ = ’ |’ ≠ ’) T$[e2]$
对变量进行定义：T$[e1]$ ‘: = ’ T$[e2]$
命题表达式之间的关系：T$[e1]$(′ ∧′|′ ∨ ′)T$[e2]$
布尔表达式取反：¬T$[e1]$
变量自增：T$[e1]$:= T$[e1]$ + 1
变量自减：T$[e1]$:= T$[e1]$ − 1
替换：x, y := $[e1, e2]R$，表示将 R 中的 x、y 的值替换为 $e1$、$e2$。

6.4.3 面向法律合同的智能合约语言

合约包括：合同名称及合同内容，合同内容包括：当事人描述、资产描述、合约条款、附加信息、合约订立。

***Contracts* ::= *Title*{*Parties*+ *Assets*+ *Terms*+ *Additions*+ *Signs*+}**

合约名称包含合约标题和合约序号构成。语法如下：

@@合同名 名称（：合同序号 文档哈希值）

Title ::= **contract** Cname (: **serial number** Chash)?

当事人描述包含当事人的名称或者姓名和住所，以及当事人所拥有的属性及特征描述（包含常量和变量）和在合同中约定的行为。语法如下：

@@当事人 群体？名称 {属性+ }

***Parties* ::= party group**? Pname {*field*+}

合约标的指合同当事人之间存在的权利义务关系，一般分为物、行为、智力成果等，是合同成立的必要条件。在合约中标的用资产来表示，在区块链中应存在该资产的描述，表示如下：

@@资产 资产名称 {资产属性{} 资产权利{}}

***Assets* ::= asset** Aname {**info**{*field*+} **right**{*field*+}}

一般条款包含条款名、条款当事人、当事人必须、可能或禁止履行的行为、条款执行条件、条款执行过程中资产的转移、以及条款执行后应满足的结果。语法如下：

@@**条款** 条款名：当事人 (**必须**|**可能**|**禁止**) 行为(属性+)
(执行所需的前置条件)?
(伴随的资产转移+)?
(执行后需满足的后置条件)?

***GeneralTerms* ::= term** Tname: Pname **(must|can|cannot)** action(*field*+)

(**when** preCondition)?

(**while** transactions+)?

(**where** postCondition)?.

式中：

——preCondition 由表示前置条件的表达式构成，在条款执行前进行检测，如果满足前置条件，则可执行条款；如果不满足，不能执行条款。

——transactions 表示条款执行过程中伴随的资产操作。

——postCondition 由表示后置条件的表达式构成，在条款执行完后进行检测，如果满足后置条件，则本条款执行成功；如果不满足，则执行失败。

违约条款指双方约定的当事人不履行智能法律合约中规定的义务或履行义务不符合约定时，应承担的法律责任。即在指定条款的后置条件未得到满足且此违约条件的前置条件得以满足时，当事人必须或可以执行违约处理，可伴随相应的资产转移，执行后应满足违约条款的后置条件。语法列举如下：

@@违约条款 条款名 （针对 条款名+）？：当事人 （必须|可以） 违约处理（属性域+）

（执行所需的前置条件）？

（伴随的资产操作+）？

（执行后需满足的后置条件）？

***BreachTerms* ::= breach term** Bname (**against** Tname+)? : Pname **(must|can)** action(*field*+)

(**when** preCondition)?

(**while** transactions+)?

(**where** postCondition)?.

智能法律合约中以仲裁条款形式规定解决争议的方法，具体争议可由自然语言陈述，并可指定仲裁机构，语法如下：

@@（所声明之争议）？由某仲裁机构进行裁决。

***ArbitrationTerms* ::= arbitration term** : (The statement of any controversy)?

administered by **institution** : instName.

7 智能合约形式化建模

7.1 智能合约可建模性

智能合约的运行可以看作从一个状态转移到另一个状态，是一个状态机模型，智能合约编写语言通常是一种面向对象的高级语言，具有图灵完备性且应用广泛，有计算机类语言类似语法、结构、构件和语义，运行代码存储在区块快上，运行环境是特定虚拟机。

智能合约中函数的设计逻辑和执行循序是保证合约状态变量符合预期的重要因素。合约代码包含了各种类型的合约状态变量、构造函数以及普通函数，其类型有整型、字符串、映射、地址、枚举、数组等。针对合约的每一个构件，建立和其语义一致的形式化模型。针对不同的建模语言，通过建模语言定义的构件、结构、关系、集合和公理即可构造对应的表达方式。因此，采用选择的形式化语言对智能合约需求进行建模，或者依据智能合约执行代码，反向建立对应形式化模型是可行的，但需要保持语义的一致性。

7.2 智能合约建模方法

可采用分层建模方法，可分为系统层、行为层、进程执行层三个层次。建立形式化模型，宜建立系统、功能块和进程（执行层单位根据具体建模语言可不同）：

a) 系统层

系统层主要解决与外界环境的交互，如各种信号传递，还有功能块的划分。还要考虑到具体需求定义新的数据类型。可以根据需求定义具体功能点状态机，作为转换依据。

b) 行为层

行为层解决功能描述表达，解决形式化描述行为能力的差异。发送信号、接收信号、子程序调用等行为用状态迁移的动作表示，状态迁移可以是并行的，也可以是串行的，还可以指定迁移条件。可把需求分为 1 个或多个进程，各个进程之间通过信号传递信息。要定义进程和各种信道、信号。

c) 进程层

进程执行单位，模型的大部分功能都在该层实现，包括各种变量、数据结构、过程、状态、任务的定义和处理，主要包括状态及迁移，以及过程设计，可以通过设计过程来简化模型，而过程的划分需要依靠前面需求来设计实现，在过程设计中要设计进程的参数传递及返回值。

8 模型转换

构建一种模型转换为另一种模型的转换规则、转换方法和转换工具。图 1 给出的一种基于 ATL 的转换架构，用于实现两个模型之间的映射。

ATL(Atlas Transformation Language)是一种专门面向模型转换的语言，实现基于 ATL 框架的模型转换工具，首先需要分别定义源语言与目标语言的元模型，元模型相当于一个语言的语法，然后使用 ATL 描述转换规则，ATL 框架提供引擎，根据转换规则，参照元模型，自动实现转换。通过编写一个描述性语句，可以清楚地表达源模型元素和目标模型元素之间的关系。与传统的手动转换相比，它可以轻松实现自动转换，并且具有很高的可重用性。例如，当生成相应对象的 PROMELA 模型时，可以使用模型检查器工具（例如 SPIN）来验证各种模型属性。

模型转换是模型驱动体系结构（MDA）的核心，也是模型演化和重建的关键技术之一。模型转换需要遵循某些约束规则来维护模型的某些属性，例如外部行为，接口的属性等。这种转换约束通常称为属性保留约束，主要用于确保模型转换不会破坏模型的某些属性。合约可以用形式语言来描述，也可以给出形式上的语义和属性。此外，定理证明还可以证明转换是否保持了合约的语义或属性。

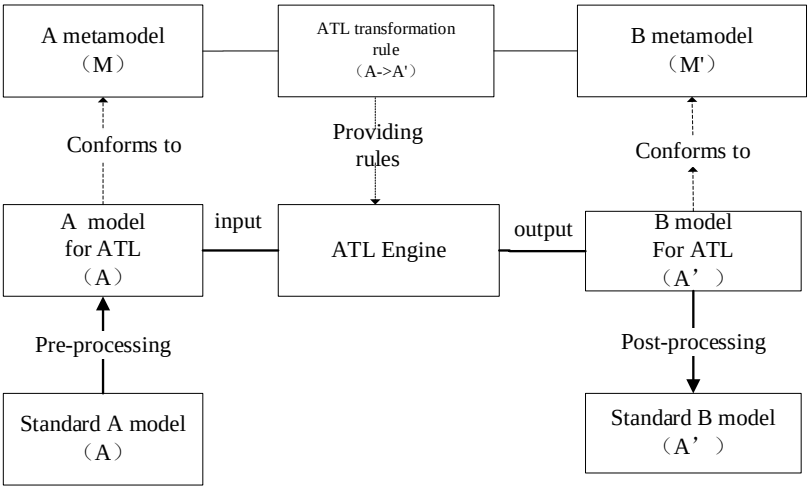

图 2 模型转换

转换需要 A 模型和 B 模型的元模型以及 ATL 转换。元模型描述语言的抽象语法，而 ATL 转换（通常作为 ATL 文件）定义源模型和目标模型之间的映射规则。预处理和后处理部分是转换的补充，包括：

a) 制定 A 元模型和 B 元模型之间的 ATL 转换规则；

b) 从标准 A 模型中获得符合 A 元模型的 ATL 模型；
c) 通过 ATL 引擎输出符合 B 元模型的 ATL 目标 B 模型；
d) 最后，从之前的 ATL 的 B 模型中获得标准的 B 模型。

9 智能合约形式化模型验证

9.1 概述

智能合约最基本的属性分为常规属性和自定义属性。自定义性质是用户期望合约具备的属性，例如所有账户的余额必须大于 C（C 为具体的金额数目）。智能合约本质上一段链上代码，因此它具有和普通程序相似的安全属性，如可达性、死锁、无状态二义性等。形式化方法主要用于解决智能合约产生与执行的可信性问题。形式化验证方法可以检查智能合约模型的很多属性，例如，合约的公平性、可达性、有界性、活锁、死锁，以及无状态二义性等，也可以验证用户自己根据需要制定的自定义属性。形式化可验证包括（不限于）以下属性：

a) 合约不变量保持
智能合约的任何动作都不能违背合约期望属性。
b) 合约状态不发散
合约状态不发散，最终会收敛到确定值。即某个函数的前置条件最终不被满足。
c) 合约状态类型检查
智能合约中所有的状态变量都建立一个类型不变量。
d) 模型精化过程一致性
精化建模过程中，不同模型之间必须满足精化关系，保持模型的一致性。
e) 可达性
验证智能合约的各种可能状态之间的可达关系。如果某个状态从初态不可达，则表明模型有错误。如果从状态 A 到状态 B 的变迁不可能发生(直接或间接)，则从状态 A 到状态 B 是不可达的。
f) 无死锁
合约中至少有一个函数的前置条件可以被满足。最典型的死锁是智能合约中各实体都处于这样的一种等待状态，即只有在“某一事件”发生后才能做进一步的动作，但在该状态下，这个“某一事件”却不可能发生。
g) 无活锁
活锁是指智能合约的运行会处于无限的死循环中，而没有别的事件可使合约状态从这一循环中解脱出来。
h) 弱活锁
指智能合约执行处于死循环中，只有当交换命令的相对速度达到某一状态时，才能退出死循环。
i) 有界性
检验合约的某些成分或参数是否有界。
j) 可恢复性或自同步性
这是当出现差错后，能否在有限的步骤内返回到正常状态(包括初始态)执行。
k) 有界性
检验合约的某些成员或参数是否有界。
l) 无状态二义性
一个进程在某一时刻只允许具有一个稳定状态。若在某一时刻进程可以有多个稳定状态，则称

该进程的状态为二义状态。

本文件将智能合约模型的属性分为安全性、活性两种，如 9.1.1 和 9.1.2 所示。

9.1.1 智能合约模型安全性

智能合约看作是运行在区块链上的状态机。其安全性包括有界性，可达性和无状态二义性等。有界性是指检验智能合约中的参数或变量是否越界（如超出数组索引）。

可达性是指智能合约各个状态的可达关系。如果合约中某个状态从初始状态开始便不可达，表示合约的设计存在缺陷。

无状态二义性是指合约在任意时刻只允许一个稳定状态，如果合约在某个时刻可以存在多个稳定状态，便称这个合约的二义状态。当上述性质不被满足时，容易引发安全漏洞，例如重入，整数溢出，变量未初始化等。

9.1.2 需求模型的活性验证

活性指的是好事情在程序的执行中一定会发生。活性包括终止性、无活锁性和保证服务性。终止性表现为程序最终会停止执行。

无活锁性表现为程序的正常执行。

保证服务性表现为所有请求服务的进程都可以得到服务。

9.2 智能合约模型的形式化验证方法

形式化验证与形式化规约之间具有紧密的联系，形式化验证就是验证已有的程序（系统）P 是否满足其规约（φ,ψ）的要求（即 $P(\phi,\psi)$），它也是形式化方法所要解决的核心问题。

推荐使用表 1 的智能合约形式化验证方法。

表 1 形式化方法推荐

形式化方法	优势	劣势或不足	使用难易程度	可验证合约性质
定理证明	使用数学的方法，通过公理或前提进行推导，保证验证的严谨性	无法保证在源代码与验证代码之间的转换一致性，实现成本高，自动化水平低	高	隐私性，安全性，功能性，语义一致性
符号执行	以符号值作为输入，借助相应工具，可得到具体测试用例，具有很高的代码覆盖率	本质上属于测试，不能100%证明智能合约无误	高	安全性，功能性
模型检测	使用市面上现有的模型检测工具，进行自动化验证	无法保证所使用的模型检测工具的完备性与正确性，合约复杂度过高会导致状态空间爆炸	低	安全性，功能性

规约语言	使用数学的思想，提出新的规约语言，保证验证的严谨性	无法保证新规约语言的正确性且实现成本高	高	安全性，功能性

表 1（续）

形式化方法	优势	劣势或不足	使用难易程度	可验证合约性质
形式化建模	使用精确的数学语句或模型组件来设计系统，其仿真结果可以复现	此方法大多使用市面上建模框架，其框架的完备性与正确性无法保证	中等	隐私性，安全性，功能性
有限状态机	思维导向简单，将智能合约抽象转换为状态机的形式，易于操作，且具有图形界面	状态定义的好坏，对智能合约的验证难易程度有很强相关性，合约复杂度过高会导致状态空间爆炸	中等	安全性，语义一致性
着色 Petri 网	基于已有的 Petri 网模型，进行形式化验证，具有良好的语义描述且具有图形界面	当智能合约逻辑较为复杂时，可能会导致可达图生成难度增加，状态空间爆炸	低	安全性，功能性
运行时验证	轻量级的新型验证技术，具有自适应和自我调整	由于运行时验证的特点，要求必须在智能合约运行时检测漏洞，无法在生产环境中使用	高	安全性，功能性

9.2.1 演绎验证

演绎验证主要基于定理证明（Theorem Proving）的基本思想，采用逻辑公式描述系统及其性质，通过一些公理或推理规则来证明系统具有某些性质。例如，针对智能合约监管规则可以抽象为合约模型中的属性，用于验证智能合约是否满足，实现了对智能合约监管的功能。

典型的建模验证方法如精化技术，可以通过从抽象到具体的方法逐步实现模型的细节和属性，并且所有的建模步骤是可推理证明的。精化技术是基于规约来开发形式化模型的一种建模和验证方法，其核心思路是：首先定义一个抽象模型并赋予它一些属性，然后向模型中不断添加细节和性质来丰富模型直到符合预期。

演绎验证的优点是可以使用归纳的方法来处理无限状态的问题,并且证明的中间步骤使用户 对系统和被证明性质有更多的了解。缺点是现有的方法不能做到完全自动化，还需与用户交互，要求用户能提供验证中创造性最强部分的工作。因而演绎证明方法的效率较低，很难用于大系统的验证。

推荐使用表 2 的定理证明技术。

表 2 定理证明技术推荐

语言/工具	形式化规约类型	优势	劣势或不足
F*语言	代码规约	支持对智能合约和虚拟机规约，可验	自动化程度低，建模复杂，不支持智能合

		证复杂属性	约的循环语句结构
Event-B 语言	代码规约	建立智能合约的 Event-B 模型并给出转换规则，可验证合约的功能性	不支持循环语句建模，自动化程度低，验证复杂的性质需要手动插入

表 2（续）

语言/工具	形式化规约类型	优势	劣势或不足
Coq 证明助手	代码规约	对 The DAO 合约规约并找到漏洞，优化了代码	应用范围局限，需要手动建模实现，比较复杂
F*证明助手	属性规约	完整地定义了 EVM 的操作语义并给出智能合约的一些安全属性	实现比较复杂，不能验证具体的智能合约属性
Coq 证明助手	属性规约	针对智能合约的五种安全问题，提出验证智能合约的框架	可验证智能合约属性有限
K 框架	运行环境规约	语义完整，支持同步更新，可扩展性强，可用于衍生分析或验证虚拟机性质的工具	KEVM 验证器处于起步阶段，缺少具体验证智能合约工具
KEVM 验证器	运行环境规约	该验证器基于完整的 EVM 语义，自动化程度较高，可验证一些合约的功能性和安全性	应用范围局限
Isabelle/HOL 工具	运行环境规约	用 Isabelle/HOL 建立 EVM 规约，涵盖智能合约属性，包括 gas 执行情况，可自动生成验证条件	基于 EVM 的部分语义并不完整，可能出现 EVM 字节码临界情况，不利于验证合约的可靠性
Lem 语言 Isabelle/HOL 工具	运行环境规约	定义了 EVM 的形式语义，面向多种证明助手 Coq，Isabelle/HOL	建模复杂，建立 EVM 模型远小于真实环境，没有考虑虚拟机网络环境，对智能合约的验证受限，不支持合约间的调用

9.2.2 模型检测

模型检测（Model Checking）方法的基本思想是通过状态空间搜索来确认合约是否具有某些性质。即给定一个合约（程序）P 和规约 ψ，生成对应于合约模型 M，然后证明 M ⊨ψ，即规约公式 ψ 在合约模型 M 中成立，这样就证明了合约（程序）P 满足规约 ψ。

模型检测方法通常采用 Dolev-Yao 模型、模态逻辑、有限状态机和进程代数等理论作为合约 分析的理论基础，其基本思想是用状态迁移系统 S 表示系统的行为，用模态/时序逻辑公式 F 描述系统的性质。一个模型检测方法主要由特定的形式模型、形式逻辑和相应的模型检测算法 3 个方面构成，不同的模型检测方法具有不同的应用领域。

将模型检测应用于智能合约以解决合约的可信问题，一般应包括以下步骤：

a) 建模。通过选择合适的建模语言和建模工具，使用模型检测工具能够接受的形式语言来描述合约。
b) 描述。阐明所要验证的合约性质，包括合约的状态可达性、死锁、活锁、有界性等。
c) 验证。对合约的状态空间进行搜索,发现合约存在的问题并及时修改,对合约进行迭代验证。

目前形式化描述技术主要分为 2 种类型：形式化描述模型和形式化描述语言。

通过形式化描述模型，可以获得抽象的合约模型。形式化描述语言总是基于一种或多种形式化描述模型。形式化描述技术已经有几十年的发展，目前有多种形式化描述模型和形式化描述语言，推荐使用表 3 的形式化描述。

表 3 形式化描述推荐

形式化描述方法	分类	内容
形式化描述模型	状态变迁模型	FSM,EFSM,Petri 网模型
	进程代数	通信系统演算(CCS) 通信顺序进程(CSP)
	其他	时序逻辑(或时态逻辑)(TL)
形式化描述语言	CCITT 组织	SDL
	ISO 组织	LOTOS, ESTELLE
	其他	Promela 语言:SPIN（著名模型检测工具）的输入语言

推荐采用的模型验证工具：

a) ProB
b) SMV（Symbolic Model Verifier）
c) SPIN（Simple Promela Interpreter）
d) SDL（Specification and Description Language）
e) UPPAAL

10 自动代码生成与一致性测试

10.1 智能合约自动生成

在使用自动生成技术之前，首先由系统工程师定制软件的需求和功能，他们更懂业务或领域知识，但并不擅长编程。然后设计工程师提供具体需求和设计说明书，软件工程师再实现为具体的可执行代码，最后系统工程师或测试工程师再进行测试。

而有了自动代码生成技术之后，系统工程师（或者领域专家）可以直接使用工具进行编码及测试，并对代码不断优化迭代，直到符合自己的需求。根据自动生成工具的输入和输出不同，可以采用不同种类的代码生成工具。

自动代码转换过程见图 3，首先使用形式化描述语言和建模方法来构建模型，模型是后续形式化验证和代码生成的基础；其次，根据描述语言的特点，设计模型与目标平台代码之间的转换规则。再次，根据转换规则和元模型设计来设计代码模板；最后，通过分析模型，提取并打包模型信息，模型信息和模板将被转换引擎用来生成目标平台代码。

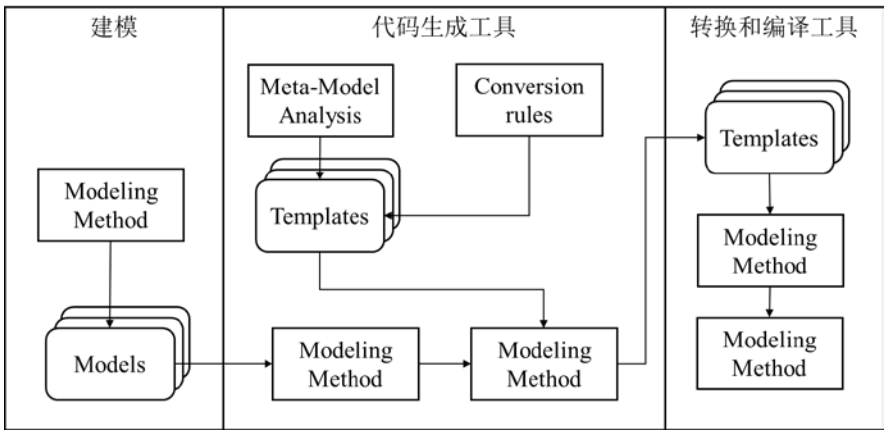

图 3 自动代码生成过程

10.1.1 结构模型和行为模型的代码生成

根据建模语言表述系统的不同层次，分为结构模型的代码生成和行为模型的代码生成。

结构模型和行为模型分别表述系统的静态结构方面和动态行为方面。静态模型生成代码的方式是相对固定的，只需进行相应翻译即可。行为模型描述了系统内部的控制流、状态迁移、对象间的交互等等动态逻辑，这些动态逻辑使得模型具有真正的应用和执行特性，但也增加了实现的复杂度。

10.1.2 基于规则引擎和基于模板的代码生成

根据代码生成过程中模型的处理方式，可以分为基于规则引擎的代码生成和基于模板的代码生成。

模型文件：利用建模语言建立的模型文件，一般是经过转换后得到的 XML 文件。

模板文件：利用模板语言建立的模板文件。

模板引擎：一般由文件解析、应用转换规则和输出代码三部分组成。文件解析是解析模型文件，然后根据预先制定的模型到代码的转换规则设计模板，将模型文件和模板文件输出为代码文件。

代码文件：模板引擎输出的代码文件。

目前已经出现多种流行的模板语言和模板引擎技术，如 JSP，Velocity 等。模板引擎主要用于信息系统分离视图层和业务逻辑层，但也可用于读取模型的信息，实现模型的代码生成。大部分模型的代码生成工具应用此技术，优点是使用具体语法定义代码片段，相比于抽象语法定义更加容易并可以灵活地运用于不同目标语言，同时容易实现代码定制，控制生成的代码格式。这一过程可分为以下几个主要步骤：

a) 利用正则表达式分解出普通字符串和模板标识符；
b) 将模板标识符转换成普通的语言表达式；
c) 生成待执行语句；
d) 将数据填入执行，生成最终的字符串，并组成新的目标语言。

10.2 自动代码生成工具

形式化语言（模型）到可执行智能合约代码的转换工具。开源的模型到代码的生成工具包括但不限于 Eclipse 旗下的 JET，Acceleo，Xpand，FUJABA，OpenExpressWeb 等，工业上也使用 Matlab 中的 Simulink 搭配 Targetlink 或 Embedded coder 等工具进行代码生成。以下列出部分自动代码生成中推荐的

技术：

a) Eclipse EMF

Eclipse Modeling Framework (EMF)是 Eclipse 提供的一套建模框架，可以用 EMF 建立自己的 UML 模型，设计模型的 XML 格式或编写模型的 Java 代码。EMF 提供了一套方便的机制，实现了功能的相互转换。

b) Xtext

Xtext 是一个开发编程语言和领域特定语言的框架，包括一个完整的语言开发流程，在 Eclipse IDE 里集成。使用 Xtext 可以定义领域语言，按照要求的格式输入语法规则，Xtext 会自动生成包括词法分析、语法分析模块，并在 Eclipse 得到完美的支持。

c) Xtend

Xtend 是 Eclipse 推出的一门静态类型的编程语言，它会编译成 Java 代码。Xtend 实现了与 Eclipse Java 开发工具的紧密集成，可以实现与 Java 的互操作。它可以帮助开发人员减少样板代码，有效提高模板可读性和可维护性，同时支持模板文件的调试。对于自动代码生成技术而言，Xtend 提供模板表达式和模板解析自动流转引擎。用户易通过模板表达式定制代码模板和 Java 中的类型系统，实现无缝连接。

d) ANTLR

ANTLR 全称为 Another Tool for Language Recognition，其前身是 PCCTS，是一种可以根据输入自动生成语法树并可视化显示出来的开源语法分析器，为 Java，C++，C#等多种语言提供了通过语法描述来自动构造自定义语言的识别器、编译器、和解释器的框架。通过 ANTLR 自动生成的语法树，可以方便的进行语言的转换。

10.3 一致性测试

10.3.1 一致性测试目的

一致性测试的主要目的是验证系统实体与系统规范的符合程度。

一致性测试的重点在于理论与现实的符合程度。

通常称系统实现为 IUT（Implementation Under Test），称一组测试实现一致性的测试序列为一致性测试集（conformance test suite）。测试集由系统规范生成。

10.3.2 一致性测试流程

一致性测试可遵循以下的流程：

a) 根据合约规范，明确测试目标；
b) 设计并实现测试套件：测试序列生成、测试数据生成、测试例实现；
c) 执行测试；
d) 根据测试执行的记录，进行测试的评估，写出测试报告。

10.3.3 一致性测试方法

10.3.3.1 本地测试法

本地测试法是最常用的一种，见图 4。

这种方法假定 IUT 的上下界都有 PCO 和 LT 及 UT 的联系。测试系统一方面通过 PCO 对 IUT 发送输入激励，另一方面接收 IUT 的输出相应，并根据系统规范描述做出判断测试是否通过。LT 和 UT 在 IUT 的上下接口处通过与 IUT 交换测试事件来观察 IUT 的行为。测试协调过程（TCP，Test Coordinate Procedure）用来协调 UT 和 LT 的运作。

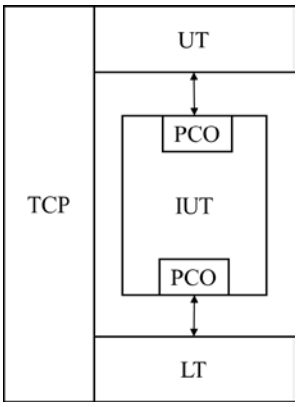

图 4 本地测试法

10.3.3.2 分布式测试法

在分布式测试法中，IUT 和 UT 处于同一台机器中，而 LT 则分布在其它机器中，如图 5 所示。

LT 和 IUT 之间利用(n-1)层服务交换报文(可以在线测试)。

与本地方法相比，LT 和 IUT 之间的接口 PCO 从 IUT 中转换到 LT 中，LT 相当于(n-1)层服务的使用者。

测试协同过程 TCP 隐含在测试例中，测试同步问题由 UT 和 LT 的操作者来实现。

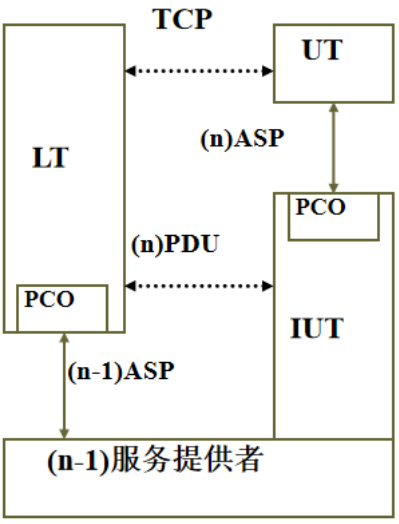

图 5 分布式测试法

10.3.3.3 协同测试法

协同测试法和分布式测试法的根本区别在于协同测试法引入测试管理协议 TMP(Test Management

Protocol)，见图 6。

有了 TMP，UT 和 LT 就通过交换 TM-PDU 实现测试协同过程。交换 TM-PDU 有两种方法：

——带内传送法，即将 TM-PDU 作为(n)ASP 的用户数据传送给 IUT，IUT 再将它传送给 LT；

——带外传送法，即将 TM-PDU 直接利用(n-1)层服务来传送。

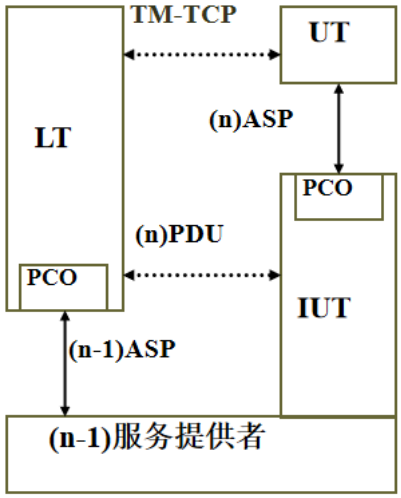

图 6 协同测试法

10.3.3.4 远程测试法

远程测试方法中没有 UT，因此也不存在 UT 和 LT 之间的协同问题，见图 7。

在这种方法中，测试例完全用(n-1)ASP 描述。

远程方法比较适用于被动式协议实现或服务型协议实体的测试。

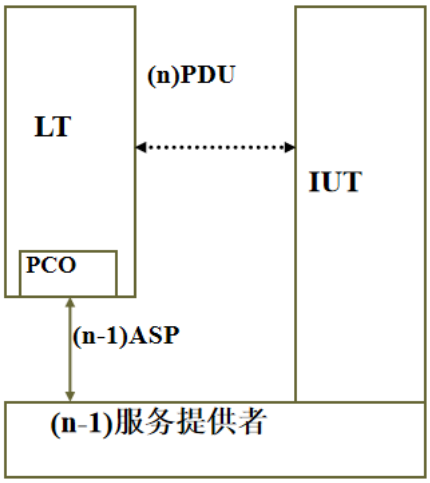

图 7 远程测试法

11 智能合约代码

智能合约语言分为两类：一类使用传统语言作为智能合约语言，具有易于编程，降低学习成本的优点；还有一类使用专用语言作为智能合约语言，通常针对区块链特点定制了安全的语法策略。目前常用智能合约语言可使用以下几种：

a) Solidity：一些区块链项目如以太坊、FISCO BCOS 等使用 Solidity 为智能合约语言，应用广泛。除此之外，以太坊也使用过 Serpent、Mutan、LLL 等作为智能合约语言。
b) Move：Facebook 的区块链项目 Diem 在使用，是一种灵活安全的函数式语言。
c) ink！：是由 Parity 开发的智能合约语言，并与 Rust 语言进行了整合，波卡（PolkaWorld）在维护并使用。
d) 通用语言：EOS 使用 C++等语言作为合约语言，Hyperledger Fabric、京东链等支持 Java、Golang 语言编写智能合约。

附录 A
（资料性）
一种形式化语言 Event-B 简介

A.1 Event-B 介绍

Event-B 是一种基于精化技术的一种离散系统建模的语言，可以描述系统的状态以及状态的转移，已经应用于航天安全关键实时系统等领域。可以用来建立与 Solidity 的语法子集语义等价的映射关系，支持从抽象到具体地方式逐步建立和需求文档保持一致的形式化模型，并通过验证生成的证明义务来保证模型的正确性。

Event-B 可以将程序或系统抽象成基于类型集合论的离散系统，并生成证明义务（proof obligations），然后验证系统的属性（properties）。离散系统主要由一系列的状态（states）和状态之间的转移（transitions）组成，这种转移也称为事件（events）。证明一个离散系统的主要任务就是验证所有的状态在转移中必须遵守给定的属性，这些属性也称为不变量（invariants）。

在 Event-B 中，状态由一些变量来定义，和命令式程序一样。不同之处在于，这些变量可能是整数（integer）、配对（pairs），集合（sets）、关系（relations）或函数（functions）等，但不支持浮点数、数组、文件这种数据结构。对于不变量的表示，可以是任何一阶逻辑或集合论中的谓词表达式。

Event-B 模型主要由代表系统静态属性的文本（contexts）和代表系统动态行为属性的机器（machines）组成。机器和文本的关系见图 A.1。

文本包含以下几个组件：集合、常量（constants）、公理（axioms）和定理（theorems）。其中，集合和常量用于定义数据结构和初始化机器中的变量（variables），公理和定理用于规约文本中集合或常量的属性。定理是可以由公理推导出来的。

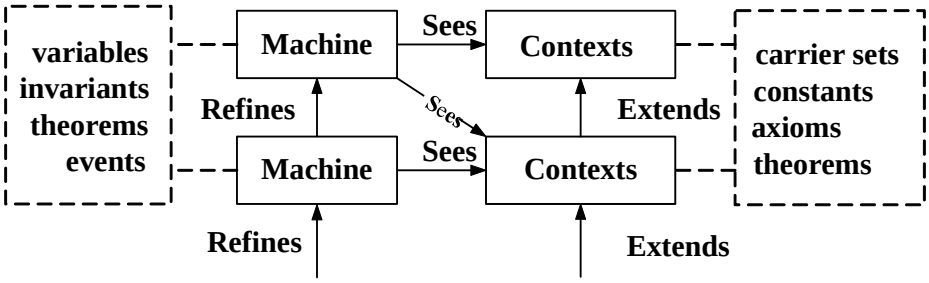

图 A.1 机器和文本关系图

机器包含变量、不变量和事件三个组件。变量的值由初始化事件赋予，并可以被事件改变。不变量定义了可达状态必须满足的属性。事件包含动作、守卫条件和参数，动作通过修改变量的值来改变系统的状态，守卫条件作为判断事件能否执行的前置条件，通常是对参数和变量的一些规约。所有事件的执行都满足原子性，即当几个事件的守卫条件同时成立时，只有其中一个事件可以执行，这种选择是非确定且随机的。

机器可以参考（sees）文本的内容，使用文本中定义的数据结构，并且文本中定义的公理和定理可以作为机器中证明的前提条件。Event-B 支持精化操作，在抽象机器的基础上，进行精化得到更加具体的机器。

A.2 Event-B 主要语法规则

Event-B 语言属于命题逻辑语言，可以细分为四个部分组成：谓词逻辑语言、等式逻辑语言、集合论语言以及布尔和算术表达式语言。下面给出 Event-B 基础的语法：

a) 谓词逻辑语言。该部分除了包含合取、析取等操作，还引入了表达式与变量。变量是一个简单的标志符，基于变量还定义了全称量词与存在量词语法。表达式由变量或两个表达式的配对组成。见表 A.1。

b) 等式逻辑语言。该部分语法引入了比较表达式的操作“=”。比较配对（pairs）的真假值如 E ↦ F = G ↦ H，可以等价为 E=G ∧ F=H。见表 A.2。

表 A.1　谓词逻辑语法范式

predicate ::= ⊥
¬predicate
predicate ∧ predicate
predicate ∨ predicate
predicate ⇒ predicate
predicate ⇔ predicate
∀ var_list · predicate
∃ var_list · predicate
expression ::= variable
expression ↦ expression
var_list ::= variable
variable, var_list

表 A.2　等式逻辑语法范式

predicate ::= ⊥
¬predicate
predicate ∧ predicate
predicate ∨ predicate
expression = expression
expression ::= variable
expression ↦ expression

c) 集合论语言。该部分语法引入了集合（set）的概念和属于（∈）的集合操作。expression × expression 表示笛卡尔积，P(expression)表示 expression 的幂集。{var_list· predicate| expression}表示递推式构造集合（set comprehension），其语义为：在满足 predicate 成立的条件下，expression 的所有取值组成的集合。见表 A.3。

表 A.3　集合论语法范式

Predicate ::= …
expression ∈ expression
expression ::= …
expression × expression
P(expression)
{var_list· predicate| expression}

d) 布尔和算术表达式语言。主要定义布尔值、整型值、自然数值、正整数值、succ 函数、pred 函数与算术表达式运算，其中 succ 函数是让整型值加 1，pred 是让整型值减 1，二者互为逆函数。见表 A.4。

表 A.4 布尔和算术表达式范式

expression ::= …	
	BOOL
	TRUE
	FALSE
	Z
	N
	N1
	succ
	pred
	0
	1
	…
	expression + expression
	expression * expression
	expression ^ expression

附录 B
（资料性）
安全性合约的验证案例

智能合约采用 Event-B 形式化方法验证安全性漏洞案例：蜜罐合约，来源于 GitHub:（https://github.com/thec00n/smart-contract-honeypots/blob/master/Gift_1_ETH.sol）

蜜罐本质上是一种对攻击方式进行欺骗的技术，通过布置一些作为诱饵的主机、网络服务等，诱使攻击方对它们实施攻击。智能合约蜜罐是一种假装以不安全的方式持有以太币的智能合约，给黑客造成假象以为他们可以从这些合约中窃取以太币。设计的初衷是引诱黑客来入侵系统，借机搜集证据，并隐藏真实环境。智能合约蜜罐不同于一般的蜜罐，它针对的是智能合约的开发者、代码审计人员以及黑客，钓鱼的门槛更高。

由于 Gift_1_eth 衍生出的很多版本的合约存在于区块链中，一些具有相同的源代码，而另一些则进行了一些细微的更改。表 B.1 展示了其中的一种版本，该合约主要有三个函数来确保交易顺利进行：

a) SetPass()：当发送方的交易价值大于 1 个 ether，变量 passHasBeenSet 设置为 false 时，可以设置新密码（即 hashPass）。
b) GetGift()：当输入的密码等于设定值（即 hashPass）时，发送方可以拿走合约中的所有 ethers。
c) PassHasBeenSet()：如果输入的密码等于设定值（即 hashPass），则变量 passHasBeenSet 会被设置为 true。

表 B.1 智能合约蜜罐代码

```
pragma solidity ^0.4.17;
contract Gift_1_ETH
{
    bool passHasBeenSet = false;
    constructor() payable{}
    function GetHash(bytes pass) constant returns (bytes32)        {return sha3(pass);}
    bytes32 public hashPass;
    function SetPass(bytes32 hash) payable
    {
        if(!passHasBeenSet&&(msg.value >= 1 ether))
        {
            hashPass = hash;
        }
    }
    function GetGift(bytes pass) returns (bytes32)
    {
        if( hashPass == sha3(pass))
        {
            msg.sender.transfer(this.balance);
        }
        return sha3(pass);
    }
    function PassHasBeenSet(bytes32 hash)
    {
        if(hash==hashPass)
        {
            passHasBeenSet=true;
```

```
        }
    }
}
```

遵循以下步骤进行模型转换：

a) 类型声明转换为 Event-B 的集合和公理。

b) 变量属性转换为 Event-B 的变量和不变量。

c) Constructor 函数转换为 Event-B 的初始化事件。

d) 普通函数转换为 Event-B 的事件、常量和公理。

以 SetPass 函数转换为例见表 B.2，得到的 Event-B 模型包含 3 个参数和 5 个守卫条件，其中 @grd4 要求转账值不能超过合约账户余额，@grd5 要求转账要高于指定的默认值。这两个守卫条件保证了 Solidity 合约的安全属性。同时定义了两个动作：@act1 是应用了 if 语句的转换规则得到的，表示变量 hashPass 的赋值取决于变量 passHasBeenSet 和 msg_value。@act2 应用了关键字 payable 的转换规则得到，它表示账户余额变化过程，更新了 this 和 msg_sender 的账户余额。

表 B.2 抽象模型的 SetPass 事件

```
event SetPass
any hash msg_sender msg_value
where
  @grd1 hash∈Z
  @grd2 msg_sender∈address_tem\{this}
  @grd3 msg_value∈N1
  @grd4 msg_value≤balanceof(msg_sender)
  @grd5 msg_value≥TRANSFER_VALUE
then
    @act1 hashPass:={TRUE↦hash,FALSE↦hashPass}
            (bool(passHasBeenSet=FALSE
            ∧msg_value≥TRANSFER_VALUE))
    @act2 balanceof:=balanceof<+{
              this↦balanceof(this)+msg_value
             ,msg_sender ↦ balanceof(msg_sender)−msg_value}
end
```

表示在可以修改合约密码的情况下，调用 SetPass 函数前后的账户余额不应该发生变化，即不应当出现调用 SetPass 后导致账户余额减少的行为。

$$\forall i \cdot i \in address\_tem \wedge passHasBeenSet = TRUE \Rightarrow$$
$$balanceof(i)[Before\ calling\ SetPass] =$$
$$balanceof(i)[After\ calling\ SetPass]$$

但是该属性中是用自然语言描述的，无法直接在 Event-B 的抽象模型中表示，因此本文对抽象模型精化并使用反证法的思路进行建模。将 SetPass 事件的守卫条件强化见表 B.3，当 passHasBeenSet 为 true 的时候该事件才能调用，假设属性二是成立的，即用户在调用 SetPass 函数后，其账户余额不发生改变，如@act2 定义的行为。

表 B.3 精化后的 SetPass 事件

```
event SetPass refines SetPass
 any hash msg_sender msg_value
  where
    @grd1 hash∈ℤ
    @grd2 msg_sender∈address_tem\{this}
    @grd3 msg_value∈ℕ1
```

```
        @grd4 msg_value≤balanceof(msg_sender)
        @grd5 msg_value≥TRANSFER_VALUE
        @grd6 passHasBeenSet=TRUE
    then
        @act1 hashPass≔hashPass
        @act2 balanceof≔balanceof
end
```

采用验证平台 Rodin 生成的证明义务出现了不通过的情况，见图 B.1。表示精化后的事件动作和抽象模型行为不一致，即在 SetPass 事件后，用户的账户余额会发生变化。从而发现了该智能合约代码中存在的逻辑漏洞。攻击该智能合约的流程如下：首先合约创建者设置了一个只有他自己知道的密码，并将变量 passHasBeenSet 设置为 true，这样只有合约创建者才能取出合约中的以太币。其他尝试调用 SetPass 函数的人来修改密码时，账户会默认转出一定金额的以太币，这就是该智能合约蜜罐的漏洞。

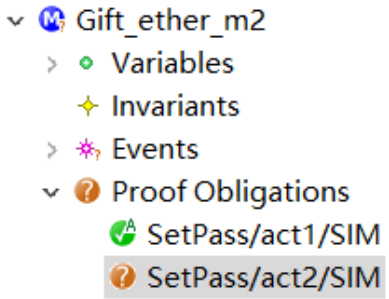

图 B.1　验证不通过的证明义务

附录 C
（资料性）
交易一般性质的验证案例

构建一个远程购物智能合约交易模型案例，采用 Event-B 的来进行正确性验证见表 C.1。该合约通过三个状态变量和三个状态转移函数实现了网上购物的功能。三个函数分别是：

a) abort（）函数：当商家觉得商品的价格需要修改时候，调用该函数可以撤回投入在合约汇总的担保金，然后重新部署合约。
b) ConfirmPurchase（）函数：商家调用该函数用于确认订单，并将合约状态锁定，只有当顾客收到货物后才能解锁。
c) ConfirmReceived（）函数：顾客调用该函数用于确认收到货物，解锁合约账户，商家可以收到合约账户的余额，即卖出商品的费用。

表 C.1　远程购物 Solidity 合约

```
contract RemotePurchase{
    uint256 public value;
    address payable public buyer;
    address payable public seller;
    enum State {Created,Inactive,Locked}
    State public state;
    constructor() public payable{
        value = msg.value/2;
        seller = msg.sender;
        require((2*value) == msg.value,"Value has to be even.");
    }
    function abort() public {
        require(state = State.Created,"Invalid state.");
        require(msg.sender == seller, "Only seller can call this");
        state = State.Inactive;
        seller.transfer(address(this).balance);
    }
    function confirmPurchase() public payable{
        require(state == State.Created,"Invalid state.");
        require(msg.value == (2*value),"Value has to be even.");
        buyer = msg.sender;
        state = State.Locked;
    }
    function confirmReceived() public {
        require(msg.sender == buyer,"Only buyer can call this.");
        require(state == State.Locked,"Invalid state.");
        buyer.transfer(value);
        seller.transfer(address(this).balance);
    }
}
```

基于对合约的行为和功能分析，为其建立 Event-B 模型，文本（context），定义一些静态数据结构。定义名为“ADDRESS”的抽象集，表示其元素是所有的合约地址，并且没有相同元素。

表 C.2 远程购物 Solidity 合约的文本模型

```
context purchase_ctx1
sets State ADDRESS
constants Created Locked Inactive seller1
value this buyer1 seller1account buyer1account
axioms
  @axm1 partition(State,{Created},{Locked},{Inactive})
  @axm2 seller1∈ADDRESS
  @axm3 value∈N1
  @axm4 this∈ADDRESS
  @axm5 buyer1∈ADDRESS
  @axm6 seller1account∈N
  @axm7 seller1account > value
  @axm8 buyer1account ∈ N1
  @axm9 (this ≠ seller1)∧(buyer1 ≠ seller1) ∧ (buyer1≠ this)
```

在机器中，需要分别为三个函数建立三个语义一致的事件（events）模型。以事件 abort 为例，首先定义一个参数“msg_sender”，表示任何调用该函数的合约地址。该合约地址不能和合约地址重复，因此需要在守卫条件中说明。其实，只有卖家才可以调用这个函数来撤销交易，因此可以直接令 msg_sender 等于 seller1。然后，这个函数主要有两个动作：将合约地址中的金额转移到调用地址中和将合约的状态改成“Inactive”。使用 Event-B 语言建模见表 C.3。

表 C.3 abort 事件模型

```
event abort
any msg_sender
where
  @grd1 state = Created
  @grd2 msg_sender∈address_tem\{this}
  @grd3 msg_sender = seller1
then
  @act1 balanceof:=balanceof<+{msg_sender↦balanceof(msg_sender)+
  balanceof(this), this↦ 0}
  @act2 state:= Inactive
end
```

当智能合约的形式化模型建立后，需要对模型的自定义属性和常规属性进行规约。下面是模型的属性规约和解释：

自定义属性 1：当合约状态为锁定的时候，合约账户余额应该是 3 倍的 value。

$$state = Locked \Rightarrow balanceof(this) = 3 * value$$

自定义属性 2：当合约状态为创建的时候，合约账户余额应该是 value。

$$state = Created \Rightarrow balanceof(this) = value$$

自定义属性 3：当合约状态为非活跃的时候，合约账户余额应该是 0。

$$state = Inactive \Rightarrow balanceof(this) = 0$$

常规属性 4：任何账户的余额都必须大于或等于 0，如果账户的余额可以为负数，那么将黑客将拥有无限多的资金，这对智能合约来说将是灾难。

$$balanceof \in address\_tem \rightarrow N$$

这些性质以不变量的形式定义在模型中，Rodin 会为模型自动生成对应的证明义务，为了验证模型的所有证明义务，需要将其导入证明视图，点击 Event-B 资源管理器中文件里面的绿色对钩符号，可以将模型上生成的所有证明义务展开。其中，

简单的证明义务可以通过定理证明助手 PP、SMT、Z3 等自动验证，有一些复杂的性质需要

人工交互式证明，手动输入一些条件见图 C.1。

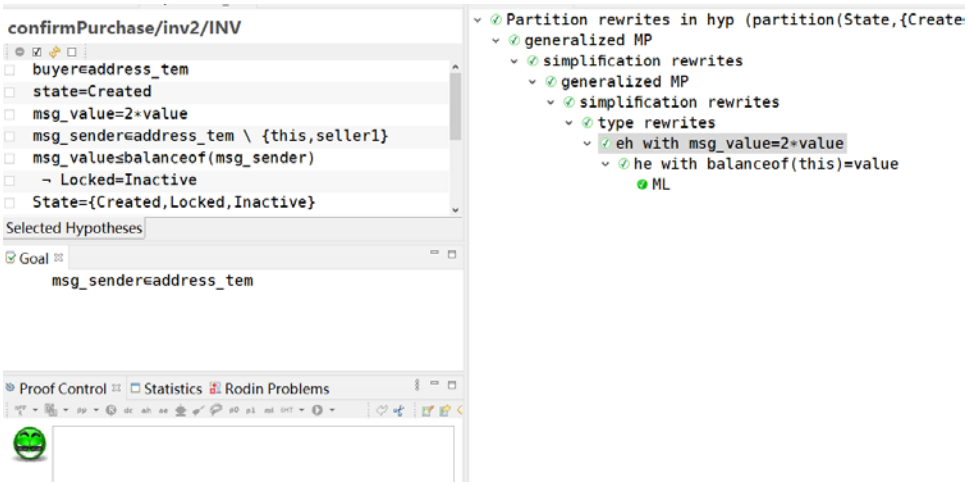

图 C.1　交互式证明视图

通过以上步骤，最终所有证明义务均通过验证的结果见图 C.2。

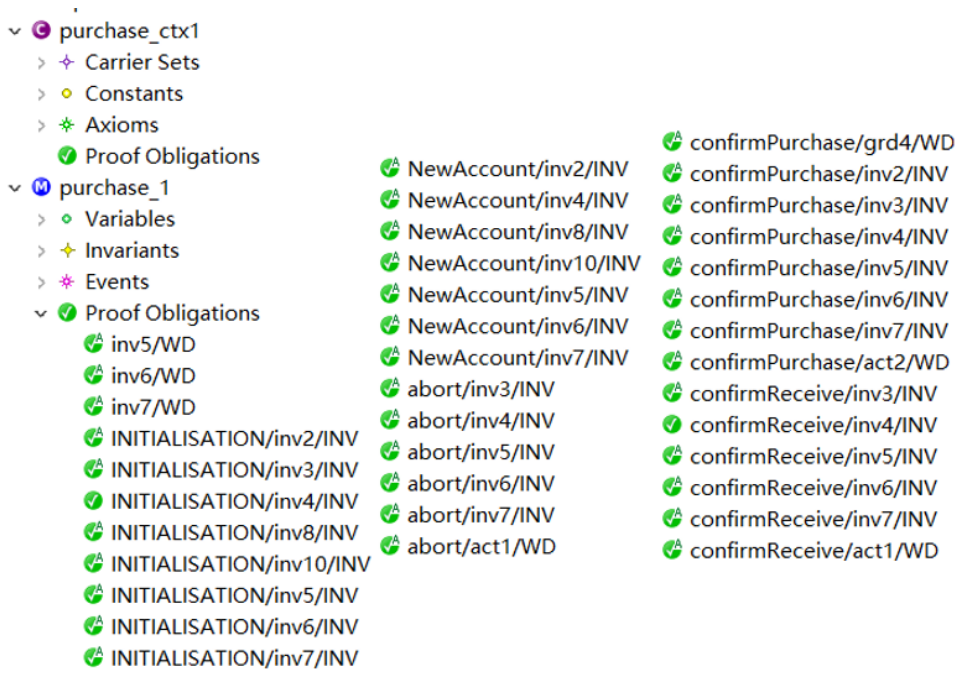

图 C.2　通过验证的证明义务

附录 D
（资料性）
SPESC 的建模与验证

SPESC 是基于 T/CIE 095—2020《区块链智能合约形式化表达》开发的面向法律人士等领域专家的领域特殊语言，下面通过一个交易合约进行开发及转换过程的说明。

根据法律合同，改写为 SPESC 合约。以交易创建和购买方确认为例，该 term 代码可以编写见表 D.1。

表 D.1 SPESC 代码示意

```
term no1 : seller can create
       while deposit $ xxxDescription::price.

   term no2 : buyer can confirmPurchase
       when after seller did create
           while deposit $ xxxDescription::price*2.
```

利用 xtext 等技术开发自动转换工具，自动生成 Event-B 模型代码，并针对报错信息进行交互式调试，根据提示，SPESC 代码没有进行余额的前置条件判断，无法通过 Event-B 全部证明义务的验证，需要进行常规属性的添加。其代码如表 D.2 所示：

表 D.2 Event-B 代码示意

```
event seller_create // no1
    where
      @grd1 seller_balance≥ price∗2
    then
      @act1 seller_balance≔seller_balance−price∗2
      @act2 balance≔balance+price∗2
      @act3 seller_create_bool≔TRUE
      @act4 seller_create_day≔day
  end

  event buyer_confirmPurchase // no2
    where
      @grd1 seller_create_bool=TRUE
      @grd2 buyer_balance≥price∗2
    then
      @act5 buyer_balance≔buyer_balance−price∗2
      @act6 balance≔balance+price∗2
      @act7 buyer_confirmPurchase_bool≔TRUE
      @act8 buyer_confirmPurchase_day≔day
     end
```

全部证明义务通过见图 D.1。

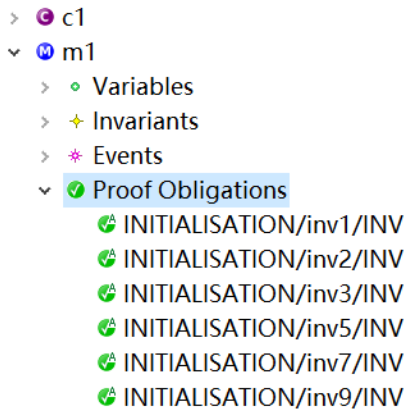

图 D.1 证明义务通过示意图

参 考 文 献

[1] Clarke E M, Grumber J O, Peled D A. Model Checking [M], Cambridge： MIT Press, 1999

[2] Kleppe Anneke, Warmer Jos, Bast Wim. 解析 MDA[M]. 鲍志云. 北京:人民邮电出版社, 2004.

[3] Rice H G. Classes of Recursively Enumerable Sets and Their Decision Problems[J]. Transactions of the American Mathematical Society, 1953, 74(2):358-358.

[4] Xiaomin Bai, Zijing Cheng, Zhangbo Duan, and Kai Hu. 2018. Formal Modeling and Verification of Smart Contracts. In Proceedings of the 2018 7th International Conference on Software and Computer Applications (ICSCA 2018). Association for Computing Machinery, New York, NY, USA, 322–326.

[5] T/CIE 095—2020, 区块链智能合约形式化表达[S]. 北京:中国电子学会, 2020.

[6] JH/CIE 170-2021, 区块链 智能合约 合同文本置标语言（CTML）[S]. 北京:中国电子学会, 2021.

[7] 胡凯,白晓敏,于卓,等. 智能合约工程[J]. 中国计算机学会通讯,2017,13(5):16-22.

[8] 朱健, 胡凯, 张伯钧. 智能合约的形式化验证方法研究综述[J]. 电子学报, 2021, 49(4):13.

[9] 王友. 基于 Event-B 的软件形式化需求获取方法研究[D]. 重庆师范大学, 2007.

[10] 董东. 智能合约设计模式[J]. 河北省科学院学报, 38(1):6.

[11] 欧阳恒一,熊焰,黄文超.一种代币智能合约的形式化建模与验证方法[J].计算机工程,2020,46(10):41-51.

[12] 吴礼发. 网络协议工程[M]. 北京:电子工业出版社, 2011.

[13] ISO/IEC 9646-1:1994, Information technology — Open Systems Interconnection — Conformance testing methodology and framework — Part 1: General concepts[S].

[14] Michael Butler. Decomposition Structures for Event-B[J]. Lecture Notes in Computer Science, 2009, 54(23):20-38.

[15] J. Abrial. Event-B language[J]. RODIN Deliverable, 2005, 3(1):2-254.

[16] 诸葛建伟,唐勇,韩心慧,段海新.蜜罐技术研究与应用进展[J].软件学报, 2013, 24(04):825-842