Санкт–Петербургский государственный университет
Математико-Механический факультет

Русаков Егор Евгеньевич

Выпускная квалификационная работа

Тема работы: использование модальных логик для формализации и верификации сценариев смарт-контрактов

Направление 01.03.02 «Прикладная математика и информатика»
Учебная группа 18.Б05-мм

Научный руководитель:
профессор, д.т.н. А. Л. Фрадков

Рецензент:
генеральный директор ООО «Современные компьютерные инновации»
А. М. Ляшин

Санкт-Петербург
2022 г.

Saint Petersburg State University
Faculty of Mathematics and Mechanics

Rusakov Egor Evgenievich

Graduation Project

Topic: Using modal logic for smart contract scenarios formalization and verification

Direction of study 01.03.02 «Applied Mathematics and Computer Science»
Study group 18.Б05-мм

Scientific Supervisor:
Professor at the Department of Theoretical Cybernetics, D. Sc. in Engineering, Professor, A. L. Fradkov

Reviewer:
CEO of LLC «SCI»
A. M. Lyashin

Saint Petersburg
2022 г.

Содержание

Введение

С 2008 года блокчейны [1] появляются все чаще и чаще, на их разработку тратится большое количество денег, и все больше обычных людей используют их для инвестиций и денежных операций. Сейчас блокчейны дают возможность полностью анонимизировать свои денежные переводы внутри сети, устраивать аукционы, получать инвестиции для компании. Большую часть этих возможностей в настоящее время предоставляют смарт-контракты [2, 3]. Это некоторые программы, которые работают на децентрализованных системах. К сожалению, если в системе находится уязвимость, то злоумышленник может похитить довольно крупную сумму денег у множества пользователей. В отличие от централизованных регуляторов денежных потоков, в блокчейне отсутствует возможность легко остановить такую кражу. В связи с этим возникает естественная проблема в верификации всех программ, которые участвуют в системе, в частности, смарт-контрактов.

Стандартные методы проверки безопасности, такие как аудит и тестирование, не способны гарантировать достаточный уровень безопасности системы, поэтому все больший интерес для поиска уязвимостей представляют методы формальной верификации [4]. Формальная верификация – это способ программного доказательства соответствия некоторого объекта верификации его описанию (спецификации). Сама верификация, в ряде методов, при этом может представлять собой формальное доказательство соответствия программного кода спецификации и проводится на абстрактной математической модели системы, в предположении о том, что соответствие между математической моделью и природой системы является изначально заданным. Например, по построению модели либо математического анализа и доказательства правильности алгоритмов и программ. Несмотря на свои фундаментальные свойства, к сожалению, широкого распространения эти системы пока не получили.

Одним из продуктов, позволяющих верифицировать программы, является интерактивная система доказательств Coq [5], которая является программным комплексом, предназначенным для формализации и проверки правильности математических рассуждений. Она представляет собой логическую среду, позволяющую описывать математические теории и в интерактивном полуав-

томатическом режиме строить доказательства [6, 7]. В системе Coq можно доказывать программы, написанные исключительно внутри этой же системы. Это, с одной стороны, может показаться неудобным, так как эта система построена на формальной логике и может быть непривычна для программистов, привыкших писать на операторных императивных языках типа Java, C/C++ или Python. Но Coq позволяет осуществлять трансляцию со своего внутреннего языка на компилируемые языки, такие как Haskell, O'Caml, и кроме того разработка транслятора на уровне исходного кода и на другие, в том числе императивные языки, представляется вполне реализуемой задачей. В этом случае авторам программного обеспечения приходится рассчитывать на корректность построенного транслятора, а также и на корректность применяемого впоследствии компилятора и/или виртуальной машины (в частности интерпретатора) финального языка программирования. Надо заметить, что несмотря на указанные ограничения, доказательство свойств построенных программ, по-прежнему дает существенный вклад в верификацию надежности программ.

В данной работе мы сфокусируемся на построении модели сценариев смарт-контрактов для блокчейна EverScale [8] при помощи программного средства Coq с последующей верификацией этих сценариев. Под сценарием будем понимать некоторую пользовательскую инструкцию. Например, если пользователь хочет купить на аукционе NFT, то он должен сделать ряд действий, которые приведут к ставке и получении им NFT, в случае если его ставка окажется наибольшей за конечное время. Абсолютно недопустимо поведение контракта, которое приводит к потере денег или несостоявшейся отправки NFT конечному пользователю. В таких сценариях мы не рассматриваем возможности неспецифицированного поведения пользователя, хотя впоследствии достаточно естественно потребовать и более сложные условия, однако, в данной работе мы не будем обращать на них внимания. В результате такой деятельности мы сможем гарантировать клиенту правильность работы смарт-контрактов при его корректном поведении.

Основным языком написания смарт-контрактов для EverScale является язык программирования Solidity. В данной работе будет описан DSL (Domain Specific Language) под названием URSUS, в который транслируются про-

граммы, написанные на Solidity в Coq для их последующей верификации.

Постановка задачи

Целью данной работы является разработка методики и программного обеспечения для верификации сценариев смарт-контрактов блокчейна EverScale при помощи системы Coq. Для достижения этой цели были поставлены следующие задачи.

- Разработать систему для построения математической модели сценариев смарт-контрактов.
- Сформулировать и верифицировать спецификацию такой системы.
- При помощи этой системы научиться формулировать сценарии и проверять необходимые свойства спецификации смарт-контрактов.
- Провести верификацию сценариев реального смарт-контракта.

1. Обзорный раздел предметной области

В данной главе представлены основные определения и термины, используемые в работе. Также описаны блокчейн, смарт-контракты и их сценарии. Кроме того, описана уже существующая система для верификации смарт-контрактов и система, которая будет использоваться в данной работе.

1.1. Понятие блокчейна

Блокчейн — это база данных, которая одновременно хранится во множестве узлов компьютерной сети, которые вычисляют каждое следующее её состояние с помощью алгоритмов консенсуса. Блокчейны наиболее известны своей решающей ролью в системах криптовалют, таких как Биткойн, для обеспечения безопасного и децентрализованного учета транзакций. Инновация блокчейна заключается в том, что он гарантирует обработку транзакций, в соответствии со строгим публично известным алгоритмом, и невозможность остановки работы системы или её удаления. Таким образом, создаётся доверие без необходимости в доверенной третьей стороне. За счёт механизмов proof-of-work и proof-of-stake [9] система гарантирует валидаторам (участникам сети, которые вычисляют состояние блокчейна) как вознаграждение за корректную работу, так и наказание за некорректную. Что и является основным источником доверия к системе и ключевым отличием от не-блокчейн распределённых баз данных.

1.2. Смарт-контракты

Смарт-контракт — это компьютерная программа, которая может быть встроена в блокчейн для облегчения проверки или согласования контрактного соглашения. Смарт-контракты работают в соответствии с набором условий, на которые соглашаются пользователи. При выполнении этих условий автоматически выполняются условия соглашения. Для лучшего понимания темы давайте рассмотрим пример. NFT — контракт отражающий "факт владения". Это значит, что он хранит или публичный ключ человека-владельца, или адрес контракта-владельца. Владелец может передать контракт другому вла-

дельцу, изменив значение поля. Метод, который это делает, проверяет, что сообщение пришло или с адреса контракта-владельца, или подписано ключом человека-владельца. Таким образом, мы можем передать право владения NFT контракту, который будет осуществлять его продажу. Самый простой сценарий - продажа за нативную валюту по фиксированной цене. Контракт-покупатель вызывает метод "купить"контракта-продавца, указывает какому адресу (или публичному ключу) нужно передать во владение NFT и прикладывает к переводу требуемую сумму. Контракт-продавец проверяет, что пришло достаточно денег, меняет владельца у контракта-NFT и переводит средства дальше туда, куда захотел его владелец. Таким образом, "посредником"в этой сделке становится контракт.

Для понимания данной работы важно знать некоторые тонкости функционирования смарт-контрактов в блокчейне EverScale. Во-первых, люди с контрактами и контракты между собой общаются сообщениями. Сообщение по сути является вызовом некоторой функции другого контракта. В сообщении передается некоторое количество денег, логический флаг bounce и специальный флаг (в системе называется просто flag), который является битовой маской и описывает: стратегию оплаты комиссии, стратегию точного вычисления суммы, поведение в случае нехватки денег на транзакцию и некоторое другое специальное поведение. Если функция, вызванная сообщением завершается с ошибкой, то в зависимости от флага bounce нужно или вернуть оставшиеся средства обратным сообщением, или ничего не делать, тогда, в случае ошибки, средства останутся на контракте-получателе.

Леджер — совокупность всех параметров состояния контракта и системных значенийы. В нем же находится очередь сообщений, которые контракт должен отправить другим контрактам. Чтобы контракт появился в блокчейне, он проходит процедуру деплоя. Контракты могут сообщениями деплоить новые контракты. Если контракт больше не нужен, то он может уничтожиться при помощи определенной функции.

Адрес контракта — функция от публичного ключа, статических переменных и кода. Таким образом, один контракт легко может убедить другой в том, что он (первый) запрограммирован определённым образом. С помощь этого механизма можно очень изящно решать вопрос доверия (авторизации)

контрактов.

1.3. Описание существующих систем для верификации

На данный момент есть очень небольшое количество проектов, которые позволяют верифицировать смарт-контракты, например, ConCert [10]. ConCert — это система, которая из абстрактно-синтаксического дерева (AST) контракта написанного на функциональном языке, переводит в глубоко вложенное AST на лямда-абстракциях с последующей обработкой MetaCoq [11] и переводом непосредственно в программы, написанные на Coq. После этого происходит процедура доказательства свойств безопасности и темпоральных свойств. ConCert также позволяет не использовать сложную верификацию, а проверить свойства при помощи средства для рандомизированного тестирования QuickChick [12]. У приведенной системы верификации есть большой минус — итоговый код свойств невозможно прочитать без специальной подготовки. Таким образом, мы должны или верить на слово верификатору, или разбираться в сложном коде, что делает эту систему плохо интегрируемой в реальные бизнес задачи.

1.4. Описания языка URSUS

Ursus — это встроенный предметно-ориентированный язык, реализованный поверх Gallina, языка декларативных спецификаций для Coq. Его можно использовать для моделирования императивного поведения в декларативной среде. Представление императивного кода на декларативном, осуществляется посредством механизма монад [13], в частности монады состояния (state-monad) . Синтаксис Ursus похож на синтаксис Solidity. URSUS можно использовать двумя способами. В качестве промежуточного шага для формальной проверки смарт-контрактов, написанных на Solidity, а именно: оригинальный смарт-контракт в Solidity переводится на Ursus и выполняется формальная проверка свойств при помощи Coq. Также можно использовать Ursus в качестве языка для разработки смарт-контрактов, то есть смарт-контракт пишется на URSUS и сразу можно проверить любые свойства безопасности. Далее этот контракт транслируется в Solidity и может быть ис-

пользован в блокчейне. Последний подход упрощает формальную проверку смарт-контракта, снижая затраты и усилия, необходимые для этого.

1.5. Исчисление конструкций

В работе используется язык Coq, который построен на исчислении конструкций (Calculus of constructions) [14] и изоморфизме Карри-Говарда. Это дает нам право считать доказательства написанные на Coq реальными математическими доказательствами. Исчисление конструкций — это формализм высокого порядка для конструктивных доказательств в стиле естественного дедуктивного вывода. Каждое доказательство представляет собой λ-выражение, полученное с помощью утверждений лежащей в основе логики. Стирание типов дает нам чистое λ-выражение, выражающее связанный с ним алгоритм. Вычисление λ-выражения примерно соответствует устранению сечений (cut-elmination). В данном случае мы получаем функциональный язык программирования высокого уровня со сложным полиморфизмом, хорошо подходящий для спецификации модулей. Понятие типа охватывает обычное понятие типа данных, но допускает также произвольно сложные алгоритмические спецификации. Теория исчисления конструкций удовлетворяет теореме о сильной нормализации, которая показывает, что все вычисления завершаются.
Исчисление конструкций находится в высшей точке лямбда-куба Барендрегта [16]. Лямбда-куб — это наглядная классификация восьми типизированных лямбда-исчислений с явным приписыванием типов. Куб организован в соответствии с зависимостями между типами и термами этого исчисления и формирует естественную структуру для исчисления конструкций. Каждое

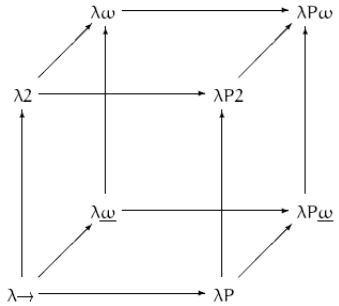

Рис. 1: λ - куб.

ребро куба представляет отношение включения $\subseteq$. Система $\lambda \rightarrow$ — это просто типизированное лямбда исчисление. $\lambda 2$ представляет собой полиморфное или типизированное лямбда-исчисление второго порядка и, по сути, является системой F [17]. $\lambda\omega$ и $\lambda\underline{\omega}$ являются исчислением высказываний высокого порядка и слабым исчислением высказываний высокого порядка. Системы λP, $\lambda P2$ и $\lambda P\underline{\omega}$ — это логика предикатов, исчисление предикатов второго порядка и слабое исчисление предикатов высшего порядка. $\lambda P\omega$ или λC — это, уже описанное выше, исчисление конструкций. Все логики являются импликативными (т.е. единственными связками являются $\rightarrow$ и $\forall$). Однако, в логиках второго и более высокого порядка можно определить другие связки, такие как $\wedge$ $\neg$ в импредикативной форме. В слабых логиках высокого порядка существуют переменные для предикатов высокого порядка, но их квантификация невозможна.

2. Математическая модель

В данной главе приведено описание математической модели для верификации сценариев смарт-контрактов. Верификация смарт-контрактов проводится в несколько этапов. Сначала контракт аудируют и пишут для него функциональную спецификацию и сценарии. После этого, для каждой функции проверяют соответствие спецификации. Самой сложной частью является проверка корректности работы сценариев, о которой в этой работе пойдет речь.

Определение. *Сценарий — это последовательность сообщений, которая соответствует возможному исполнению системы контрактов.*

Определение. *Прямой сценарий — последовательность внешних сообщений, которая соответствует некоторой пользовательской инструкции.*

Верификация таких сценариев позволяет гарантировать пользователю, что если он все делает «правильно», то ничего неожиданного не случится. Однако это не гарантирует невозможность пользователем украсть средства или как-то еще некорректно (с точки зрения спецификации контракта) повлиять на работу системы.

Определение. *Обратный сценарий — произвольная последовательность сообщений, переводящая из одного состояния леджера в другое.*

Доказательство корректности работы обратных сценариев является нерешенной задачей. Для их верификации необходимо рассмотреть все возможные последовательности сообщений, которые из одного состояние приводят к другому и доказать, что других нет. Другими словами мы гарантируем, что переход из одного состояния в другое происходит специфицировано.

2.1. Построение дерева сообщений

Все контракты, написанные на урсусе, имеют свой леджер. Чтобы реализовать взаимодействие между контрактами нужно иметь структуру для хранения и обработки сообщений, уметь по сообщению моделировать его исполнение, в том числе моделировать поведение функции, и менять сразу все

леджеры. Для унификации типов леджеров и типов выражений было написано два рекорда.

```
Context (Interfaces : Type) (Contracts : Type).

Record ledgerWithType := make_ledg {
    tp  : Contracts;
    led : GetLedger tp;
}.
Definition superLedger := XHMap address ledgerWithType.
```

Первый рекорд позволяет нам лаконично объединить все леджеры системы контрактов в один хэш-мап. Все изменения леджеров будут храниться в структуре под названием суперледжер. Так как на данном этапе контракта у нас нет, необходимо внести в контекст Contracts и Interfaces, которые в будущем нужно будет проинстанциировать типами, которые соответствуют контрактам и существующим в них интерфейсам.
Второй рекорд нужен для корректной работы с функциями. Тип каждой функции зависит от трех параметров: типа леджера контракта, возвращаемого значения и флага, отвечающего за возможность функции завершиться досрочно.

```
Record UExpressionWithType l `{lC : LedgerClassT l} :=
make_exp {
    X_of    :  Type;
    _       :  XDefault X_of;
    b_of    :  bool;
    Uexp_of :> UExpression l X_of b_of
}.
```

Для абстрактной работы с контрактами мы написали класс типов ContractsUtils, который, после инстанциации, позволяет работать с разными сообщениями и контрактами как с одним целым. Полную реализацию можно посмотреть в репозитории [18]. Почти все последующие функции будут параметризованы этим классом.

Следующей важной функцией является функция

```
Definition exec_super_state (C : Contracts) (I : Interfaces)
  (pubkey : XUInteger256) (r : GetLedger C) (addrS : address)
  (mess : OutgoingMessage (GetInterface I)) (IsInternal : bool)
  : GetLedger C * errorType.
```

Эта функция позволяет обрабатывать сообщения на уровне леджера, генерировать обратные сообщения и для более информативного вывода возвращать сообщение об ошибке работы функции.
Эти подготовительные работы напрямую не связаны с самими сценариями, чтобы их определить нужно научиться по сообщению строить дерево. В вершинах дерева мы будем хранить интерфейс, сообщение, адреса отправителя и получателя, коды возможных ошибок и суперледжер, на котором это сообщение будет обрабатываться.

```
Structure InterpretNode_ans := mk_IN {
    sl_of    : superLedger;
    tss_of   : list (list nodeType);
    errTp_of : errorType
}.

Definition InterpretNode
    (n : nodeType) (sl : superLedger) : InterpretNode_ans.
```

Функция InterpretNode обрабатывает одну вершину и по ней строит список списков сгенерированных сообщений, которые еще предстоит обработать. Причем сообщения разным контрактам помещаются в разные списки.
Понятно, что руками составлять полное дерево сообщений сложно и легко ошибиться, поэтому система сделана так, чтобы верификатору достаточно было написать только последовательность внешних сообщений некоторого сценария.
Дерево сообщений будет строиться следующим образом. Есть список внешних для системы контрактов сообщений, которые соответствуют некоторому

сценарию подлежащему верификации. Каждая вершина при помощи функции InterpretNode позволяет составить новый список сообщений, который помещается в список еще не обработанных сообщений. Далее по очереди мы достаем по одному сообщению из этого списка, интерпретируем и добавляем его ребенком для прошлого сообщения. Таким образом, почти все сообщения становятся потомками вершины, которая их сгенерировала, или потомками вершины с сообщением, которое было сгенерировано тем же родителем.

```
Fixpoint Builds (Build : nodeType -> StateT' messageTree)
(nss : list (list nodeType)) : StateT' (list messageTree) :=
  match nss with
  | (n :: ns) :: nss =>
    modify (putMessageQueue ns) >>
    do mt  ← Build n;
    do mts ← Builds Build nss;
    $ (mt :: mts)
  | _ :: nss => Builds Build nss
  | _        => $ nil
end.

Fixpoint Build (fuel : nat) (n : nodeType) : StateT' messageTree :=
  if fuel is S fuel then
    do sl ← embed_fun SuperLedger;
    do ms ← embed_fun MessagesQueue;
    let Int := (InterpretNode n sl)                    in
    let sl' := sl_of  Int                              in
    let nss := tss_of Int                              in
    let err := errTp_of Int                            in
    let qss' := if ms is nil then nss else ms :: nss in
      modify (putSuperLedger sl') >>
      do mts ← Builds (Build fuel) qss';
      $ (Node (putError n err) mts)
  else modify runOut >> $ (Node n nil).
```

Функции построения дерева написаны для обработки одной вершины с внешним сообщением, понятно, что для запуска на списке внешних сообщений достаточно несколько раз запустить функцию Build. Так как нам важно, что обработка сообщения меняет суперледжер, функция возвращает монаду состояния, которая хранит не только полученное дерево, но и суперледжер актуальный на данный момент. Из-за того, что система контрактов в теории может генерировать бесконечное количество сообщений, функции Build и Builds имеют в аргументах переменную топливо, которая отвечает за завершаемость функции.

Когда дерево уже построено, можно начать верификацию сценария. Здесь возникает два вопроса, ответа на которые еще нет. Так как порядок обработки некоторых сообщений не специфицирован (и в реальности происходит в произвольном порядке), можно ли утверждать, что никакие другие способы построения такого дерева не дадут другой результат? Существует ли способ задать некоторый класс эквивалентности суперледжеров так, чтобы даже при построении другого дерева мы приходили к аналогичному результату (в случае, если контракт «хороший»). Ответы на данные вопросы выходят за рамки этой работы, однако требуют дополнительного исследования в будущем.

3. Спецификация математической модели

В предыдущих главах был описан принцип построения математической модели сценариев смарт-контрактов. Однако функции, которые участвуют в построении дерева, никак не гарантируют корректность построения дерева. В этой главе будет описана спецификация, соответствие которой было доказано. В данной работе мы никак не проверяем корректность работы более низкоуровневых функций.

3.1. Верификация спецификации обработки сообщений

От функции InterpretNode естественно потребовать следующие свойства.

- Все сообщения, которые генерирует какое-то сообщение должны быть внутренними.
- Адреса отправителя и получателя обработаны корректно.
- Если сообщение завершилось с ошибкой, то флаг bounce обработан корректно.

```
Lemma IN_internal n sl sl' qss err :
  InterpretNode n sl = mk_IN sl' qss err ->
     Forall (Forall (fun n => isInternal n = true)) qss.
```

Первый пункт формулируется довольно просто. Если запустить функцию InterpretNode на некоторой вершине n и суперлежере sl и получить некоторые новые суперледжер sl', список сообщений qss и возможную ошибку err, то любое сообщение из qss является внутренним. Важно, что мы считаем отправление денег на внешние для системы контрактов адреса внутренним сообщением.
Для второго пункта спецификации лемма формулируется похожим образом.

```
Lemma IN_addrS n sl sl' qss err:
  InterpretNode n sl = mk_IN sl' qss err ->
  Forall (Forall (fun n' => getAddrS n' = getAddrR n)) qss.
```

Если запустить функцию InterpretNode на некоторой вершине n и суперлежере sl и получить некоторые новые суперледжер sl', список сообщений qss и возможную ошибку err, то адрес отправителя любого сообщения из qss совпадает с адресом контракта, обработавшего сообщение из n. Для последнего пункта был написан предикат, который описывает слово «корректно».

```
Definition onBounce_spec `{eqb_spec Interfaces}
(addrS : address) (n : nodeType) :=
  isOnBounce n                    /\
  getAddrS   n           = addrS /\
  getBounce  (getMess n) = false.
```

Сообщение bounce считается корректным, если оно соответствует интерфейсу onBounce, отправляется на обратный адрес и имеет флаг bounce равный false.
Доказана лемма о соответствии такой спецификации.

```
Lemma IN_error n sl (Int := (InterpretNode n sl)) :
  isInternal n                = true ->
  isSome (err_of Int)         = true ->
  has_money_for_on_bounce Int = true ->
  getBounce (getMess n)       = true ->
  exists2 l,  In l (tss_of Int) & Exists (onBounce_spec (getAddrR n))
```

Лемма гласит, что если сообщение было внутренним, оно завершилось с ошибкой, для отправки сообщения bounce было достаточно денег и флаг bounce был равен true, то в списке отправленных сообщений есть сообщение, которое удовлетворяет onBounce_spec.

3.2. Верификация спецификации построения дерева сообщений

Для проверки фактов про деревья нужно сформулировать предикат, который по дереву сообщений, сценарию и суперледжеру проверяет соответствие дерева дереву сообщений, построенному для данного сценария и суперледжера.

```
Fixpoint checkTreeT (mt : messageTree) (qss : list (list nodeType))
  StateT bool :=
    let fix Checks mts qss : StateT bool :=
    match mts, qss with
    | [ ]    , [ ]               => $true
    | t :: ts, (q :: qs) :: qss  =>
      do sl ← get;
      let sl'  := sl_of  (InterpretNode (Tree.root t) sl) in
      let tss  := tss_of (InterpretNode (Tree.root t) sl) in
      let err  := errTp_of (InterpretNode (Tree.root t) sl) in
      let qss' := if qs is nil then tss else qs :: tss    in
        (put sl' >>
        do b  ← checkTreeT t qss';
        do bs ← Checks ts qss;
        $ (eqb (Tree.root t) (putError q err) && b && bs))
    | _      , _                 => $ false
end in
Checks (Tree.nodes mt) qss.

Fixpoint checkForestT (l : list messageTree) : (StateT bool) :=
  match l with
  | [ ]     => $ true
  | t :: ts =>
    modify (setTime (getTime (Tree.root t))) >>
    do sl ← get;
    let: mk_IN sl' qss err := InterpretNode (Tree.root t) sl in
    put sl' >>
    do b  ← checkTreeT   t qss;
    do bs ← checkForestT ts;
    $ (b && bs && eqb (getError (Tree.root t)) err)
end.
```

Теперь, когда мы можем проверить, является ли дерево корректным, можно сформулировать спецификацию и доказать соответствие ей.

- Если дерево построено корректно, то у него корень внешнее сообщение, а все его дети внутренние сообщения.
- Все адреса в сообщениях корректно построенного дерева корректны.
- Если дерево построено корректно, то ситуация завершения функции ошибкой обрабатывается корректно.

Во всех формулировках теорем первым условием будет корректность дерева.

```
Theorem checkForest_messages_type trs ems sl :
  checkForest trs ems sl ->
  (forall tr, In tr trs -> isExternal (Tree.root tr)) /\
  (forall tr, In tr trs ->
  Forall (Tree.Forall (fun x => isInternal x))
  (Tree.nodes tr)).
```

В этой теореме проверяется, что если корень дерева является внешним сообщением, то все его дети являются внутренними сообщениями.

Для оставшихся двух пунктов необходимо сначала сформулировать определение слова «корректно».

```
Inductive addr_spec : messageTree -> Prop :=
  | Addr_spec1 n (t : messageTree) ts
    (addrRt       : getAddrS n = getAddrS (Tree.root t))
    (addrSts      : Forall (fun n' => getAddrS (Tree.root n') =
                                      getAddrR n) ts)
    (addr_spec_tr : Forall addr_spec (t :: ts)) :
     addr_spec (Node n (t :: ts))
  | Addr_spec2 n ts
    (addrSts      : Forall (fun n' => getAddrS (Tree.root n') =
                                      getAddrR n) ts)
    (addr_spec_tr : Forall addr_spec ts) :
     addr_spec (Node n ts).
```

В этом пункте спецификации говориться о том, что наше дерево сообщений должно быть построено так, что адрес отправителя всегда верный, с точки зрения того, кто это сообщение отправил.

```
Theorem checkForest_addr_spec trs ems sl :
  checkForest trs ems sl ->
    Forall addr_spec trs.
```

Была доказана теорема, что такая спецификация выполняется на всем дереве. Спецификация последнего пункта использует уже знакомый нам предикат onBounce_spec.

```
Definition error_spec `{eqb_spec Interfaces} (t : messageTree) :=
  forall t',
  let rt' := Tree.root t' in
    Tree.sub t' t ->
    isInternal rt' ->
    isSome (err_of (getError rt'))  ->
    has_money_for_on_bounce (getError rt') ->
```

```
    getBounce (getMess rt') ->
  exists2 n, Tree.In n t' &
             onBounce_spec (getAddrR rt') n.
```

И доказана теорема о том, что на любом корректном дереве выполняется предикат error_spec.

```
Theorem checkForest_error_spec `{eqb_spec Interfaces} trs ems sl :
  checkForest trs ems sl ->
    Forall error_spec trs.
```

Таким образом, можно считать, что наша система работает ровно так, как ее спроектировали создатели.
Естественно, для доказательства вышеуказанных фактов было доказано гораздо больше теорем, чем приведено в данном тексте. Более полный список теорем и доказательств можно найти в репозитории [18].

4. Верификация сценария системы смарт-контрактов

В этой главе мы разберемся, что нужно сделать, чтобы верифицировать сценарии конкретной системы смарт-контрактов. Чтобы начать строить сценарии и проверять свойства системы контрактов, нужно написать инстансы для всех тайпклассов. Самым энергозатратным является инстанс для ContractUtils. Написание инстансов для относительно небольшого контракта занимает значительное количество человеко-часов, мы надеемся, что в скором времени мы сможем автоматизировать эту деятельность. Так как написание инстансов не является интеллектуальной деятельностью, в данной работе мы опустим подробности, но их можно найти в репозитории [18].
В данной работе мы рассматриваем сценарии системы контрактов для инвестиций. Мы не будем делать аудит этого контракта, но для понимания сценария,который будет построен дальше нужно немного знать про этот смарт-контракт. Есть какая-то команда, которая делает проект и хочет его запустить в сети EverScale. Инвестировать в этот проект на ранних стадиях могут люди, которых мы условно будем называть инвесторами и гиверами. Инвестор вкладывает средства, а гивер увеличивает его значения, используя резервы сети, чтобы поддержать инвесторов. Система состоит из трех контрактов: Blank, FromGiver и DPool. Последние два контракта отвечают за первичное хранение денег гиверов и инвесторв, если что-то пошло не так, то с этих контрактов средства возвращаются обратно. Если средства были успешно собраны, то начинается фаза голосования за появление дополнительного кода в контракте Blank, который в последствии будет регламентировать участие средств в инвестиционном фонде. Сценарий будет выглядеть следующим образом.

1. Деплой контракта Blank.
2. Вызов setDPoolCodeHash.
3. Деплой контракта DPool.
4. Вызов setFromGiverCodeHash.
5. Деплой контракта FromGiver.

6. Трансфер денег на DPool.
7. Трансфер денег на FromGiver.
8. Вызов функции finalize у контракта DPool.
9. Вызов функции finalize у контракта FromGiver.
10. Вызов функции startVoting.
11. Вызов функции Vote.

К сожалению, пока не реализована возможность делать деплои автоматически, более сложные сценарии (например, с переменным количеством гиверов) верифицировать невозможно, поэтому в работе мы рассмотрели несколько упрощенные сценарии.
Начнем с первого. Если ровно один человек голосовал, то его голос соответствует результату голосования.

```
Theorem voting_result1 b :
  exists2 sled : LedgerLRecord,
    make_ledg Blank sled = (slf1 b)[addrBlank] &
  voting_result__ (LedgerLGetField Ledger_MainState sled) = Some b.
  Proof.
  destruct b.
  { Definition
   sl' := Eval vm_compute in (slf1 true).
   reflexivity.  }
  Definition
  sl'' := Eval vm_compute in (slf1 false).
  exists (led sl''[addrBlank]); reflexivity.
  Qed.
```

Данное утверждение не представляет какого-то внушительного результата, однако оно выявила ряд проблем. `slf1` – леджер, который получается в результате работы сценария. Первая часть формулировки леммы позволяет Coq правильно вычислить типы. На саму лемму она никак не влияет, но иначе

написать формулировку леммы не удалось. В процессе доказательства леммы стало понятно, что нужно установить флаг `Set Nested Proofs Allowed`, который позволяет писать определение внутри доказательства. Потребность в этом возникла, потому что тактика `vm_compute` зависала при попытке вычисления леджера, но при помощи определений внутри доказательства удалось вычислить и леджер. Также лемма была доказана вычислением суперледжера для двух возможных значений `b` , что не получится сделать, если сценарий будет зависеть от переменной типа `nat`.
Следующий сценарий не получилось доказать, так как не была доделана предыдущая фаза верификации с доказательство функциональной спецификации, которая является последним шагом доказательства этой теоремы.

```
Definition transfer_rest (summa_givers : uint128) : bool :=
let d1      := led sl_after_deploy[addrDPOOL]          in
let farm_rate := arm_rate_ (getD Ledger_MainState d1)  in
let quant     := quant_     (getD Ledger_MainState d1) in
(summa_givers <? quant * farm_rate / 100) &&
(FG_MIN_BALANCE_ <? summa_givers) ==>
let ms :=
  make_inode
    IDef
    (EmptyMessage PhantomType (make_internalParams
    (Build_XUBInteger
    ((quant * (quant * farm_rate / 100 - summa_givers)) /
    ( quant * farm_rate / 100)) )
    true
    (Build_XUBInteger 1)))
    addrGiver
    addrDPOOL
    Build_errorType None true) in
existsb (fun tr => Tree_bIn (inr ms) tr) (forest2 summa_givers).
```

В этой теореме гарантируется, что для произвольной суммы, присланной гиверами, если верны некоторые неравенства, в дереве будет находится сооб-

щение с перевод остаточных денег инвесторам. Несмотря на то, что теорему доказать не получилось, применение QuickChick показало, что факт является верным.

Заключение

В рамках данной дипломной работы были достигнуты следующие результаты.

- Построена математическая модель, позволяющая моделировать сценарии смарт-контрактов для блокчейна EverScale.
- Доказано, что модель соответствует написанной для неё спецификации.
- При помощи данной модели сформулированы и частично проверены некоторые сценарии контракта KW.

Уже сейчас проделанная работа позволяет проверять довольно большой класс сценариев, однако, стоит указать на некоторые допущения модели, которые должны быть исправлены в будущем. Во-первых, нет поддержки газа, использующегося для исполнения сообщений. Периодически случаются ситуации, когда на выполнение сообщения не хватает газа. Программист может забыть написать команду `tvm_accept`, что приведет к трате только кредитного газа и функция не будет работать корректно. Во-вторых, нет поддержки флагов в сообщениях. В-третьих, должны быть реализованы автоматические деплои и генерация инстансов. Отсутствие возможности деплоить контракты автоматически накладывает ограничения на контракт, который можно верифицировать. Например, моделирование множественных деплоев в цикле, на данный момент, почти невозможно. Ровно, как и необходимость писать инстансы руками вынуждает тратить большой объем времени на монотонную работу.
Уже сейчас понятно, как можно исправить указанные пробелы модели. В ближайшем будущем планируется доделать модель до почти полного соответствия реальной работе смарт-контрактов.

Список литературы

[1] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". — 2008

[2] Zibin Zheng and Shaoan Xie and Hong-Ning Dai and Weili Chen and Xiangping Chen and Jian Weng and Muhammad Imran. "An overview on smart contracts: Challenges, advances and platforms". Future Generation Computer Systems — 2020. — Vol. 105. — P. 475-491.

[3] Vitalik Buterin. Ethereum: "A Next-Generation Smart Contract and Decentralized Application Platform". — 2013. URL: `http://ethereum.org/ethereum.html`.

[4] Serna-M., Edgar, David Morales-V. "State of the Art in the Research of Formal Verification". Ingenieria Investigacion y Tecnologia, XV, 04 (2014): 615-623.

[5] Сайт проекта Coq. URL: `coq.inria.fr`.

[6] G. Gonthier, A. Asperti, J. Avigad et al. "A machine-checked proof of the Odd Order Theorem". // 4th International Conference on Interactive Theorem Proving (ITP'13). — 2013.

[7] Gonthier G. "A computer-checked proof of the Four Colour Theorem". — 2005.

[8] Nikolai Durov. "Telegram Open Network". — 2019.

[9] Bentov I., Gabizon A., Mizrahi A. Cryptocurrencies without Proof of Work. // Cryptography and Security. — 2015.

[10] Abhishek Anand et al. CertiCoq: "A verified compiler for Coq". In: CoqPL'2017.

[11] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau and Théo Winterhalter. "The MetaCoq Project". February 2020. Extended version of the ITP 2018 paper.

[12] Leonidas Lampropoulos, Zoe Paraskevopolou, Benjamin C. Pierce. "Generating Good Generators for Inductive Relations". In: POPL'2018

[13] Eugenio Moggi. Notions of computation and monads. Information and Computation Vol. 93, July 1991, P. 55-92

[14] Thierry Coquand, Gérard Huet. "The calculus of constructions". Information and Computation Vol. 76, Issues 2–3, February–March 1988, P. 95-120

[15] P. Martin-Löf Intuitionistic type theory Studies in Proof Theory, Lecture Notes, Bibliopolis, Naples (1984)

[16] Henk Barendregt, Wil Dekkers, Richard Statman. "Lambda calculus with types". Cambridge University Press. — 2013

[17] Girard, Jean-Yves. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur : Ph.D. thesis. — Université Paris 7, 1972.

[18] Репозиторий проекта . URL: `https://github.com/fibletype/scenarios`.