MODELING AND ANALYSIS OF INFORMATION SYSTEMS, VOL. 27, NO. 4, 2020
journal homepage: www.mais-journal.ru

THEORY OF COMPUTING

# Architecture of the Formally-Verified Distributed Ledger System InnoChain

L. A. Merkin-Janson[1], R. M. Rezin[1], N. K. Vasilyev[1,2]

DOI: 10.18255/1818-1015-2020-4-472-487

[1]Innopolis University, 1 Universitetskaya, Innopolis, 420500, Russia.
[2]National Research University Higher School of Economics, 20 Myasnitskaya St., Moscow 101000, Russia.

MSC2020: 93A30, 68Q60
Research article
Full text in Russian

Received November 17, 2020
After revision December 3, 2020
Accepted December 16, 2020

In this paper we consider the software architecture of InnoChain, a distributed ledger system (DLS) with 5 levels of formal verification, including a formally-verified underlying operating system (OS). The objective of this architecture is to achieve a higher level of DLS dependability compared to more traditional software architectures and quality assurance (QA) methods. The architecture of InnoChain includes (1) a programming language for smart contracts which is a domain-specific language with formal semantics embedded into CakeML, which is a functional language of the ML family; this allows us to carry out formal verification of smart contracts' correctness properties using higher-order logic systems, such as HOL4; (2) trusted compilation of smart contracts into the machine code using the verified compiler available for CakeML, rather than relying on a virtual machine for execution of smart contracts; (3) using CakeML for implementation of InnoChain node functionality which allows for formal verification of code correctness and trusted compilation into the machine code; (4) formal verification of the consensus protocol used InnoChain, namely HotStuff BFT; (5) using seL4, a formally-verified microkernel, as the underlying OS for InnoChain instead of more traditional general-purpose OSes such as Linux. The proposed verified architecture will allow InnoChain to be used in mission-critical applications, such as the decentralized Aircraft Fuelling Control System which is currently under development for JSC Aeroflot, the Russian national air carrier.

**Keywords:** distributed ledger systems; formal verification; HOL4; CakeML; seL4

INFORMATION ABOUT THE AUTHORS

| | |
|---|---|
| Leonid Al'bertovich Merkin-Janson<br>correspondence author | orcid.org/0000-0002-8427-2200. E-mail: l.merkin@innopolis.ru<br>Professor, Doctor of Mathematics, Delft University of Technology, Dr. |
| Ruslan Maratovich Rezin | orcid.org/0000-0002-0604-2645. E-mail: r.rezin@innopolis.ru<br>Head expert in formal verification methods at Innopolis University, Master degree in Computer Science, MSc. |
| Nikolay Konstantinovich Vasilyev | orcid.org/0000-0003-3112-5482. E-mail: n.vasilev@innopolis.ru<br>Lead Developer. |

**Funding:** Ministry of Digital Development, Communications and Mass Media of the Russian Federation and Russian Venture Company (Agreement No. 004/20 dd. 20.03.2020, IGK 0000000007119P190002).

**For citation**: L. A. Merkin-Janson, R. M. Rezin, and N. K. Vasilyev, "Architecture of the Formally-Verified Distributed Ledger System InnoChain", *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 472-487, 2020.

МОДЕЛИРОВАНИЕ И АНАЛИЗ ИНФОРМАЦИОННЫХ СИСТЕМ, ТОМ 27, № 4, 2020
сайт журнала: www.mais-journal.ru

THEORY OF COMPUTING

# Архитектура формально-верифицированной системы распределенного реестра InnoChain

Л. А. Меркин[1], Р. М. Резин[1], Н. К. Васильев[1,2]

DOI: 10.18255/1818-1015-2020-4-472-487

[1]Университет Иннополис, Университетская, д.1, г. Иннополис, 420500 Россия.
[2]Национальный исследовательский университет «Высшая школа экономики», ул. Мясницкая, д. 20, г. Москва, 101000 Россия.

УДК 519.7
Научная статья
Полный текст на русском языке

Получена 17 ноября 2020 г.
После доработки 3 декабря 2020 г.
Принята к публикации 16 декабря 2020 г.

В настоящей работе рассматривается архитектура системы распределенного реестра (CPP) InnoChain. Основной целью этой архитектуры является реализуемость 5-ти уровней формальной верификации программного обеспечения (ПО) системы InnoChain, включая операционное окружение. Методы формальной верификации являются основными методами обеспечения качества ПО с критическими требованиями по надежности, но до сих пор они не находили широкого применения в CPP. Архитектура InnoChain включает (1) предметно-ориентированный язык смарт-контрактов с формальной семантикой, встроенный в функциональный язык CakeML (диалект языка ML), что позволяет осуществлять формальную верификацию свойств корректности смарт-контрактов в системах логики высших порядков (например, HOL4); (2) верифицированную трансляцию смарт-контрактов в машинный код с использованием компилятора CakeML вместо использования виртуальных машин для исполнения смарт-контрактов; (3) реализацию функционала узла CPP также на CakeML с формальной верификацией свойств корректности и с верифицированной трансляцией исходного кода узла в машинный код; (4) формальную верификацию протокола консенсуса CPP (HotStuff BFT); (5) использование формально-верифицированного микроядра seL4 в качестве операционного окружения CPP вместо операционных систем общего назначения. Предлагаемая архитектура открывает возможности для использования CPP InnoChain в критических по надежности приложениях, в частности, в системе управления заправкой воздушных судов ПАО Аэрофлот.

**Ключевые слова:** системы распределенного реестра; формальная верификация; HOL4; CakeML; seL4

## ИНФОРМАЦИЯ ОБ АВТОРАХ

| | |
|---|---|
| Леонид Альбертович Меркин<br>автор для корреспонденции | orcid.org/0000-0002-8427-2200. E-mail: l.merkin@innopolis.ru<br>Профессор, научный руководитель Лидирующего исследовательского центра в области формально-верифицированных систем распределенного реестра. |
| Руслан Маратович Резин | orcid.org/0000-0002-0604-2645. E-mail: r.rezin@innopolis.ru<br>Главный эксперт формальных методов верификации, степень магистра по направлению “Информатика и вычислительная техника”. |
| Николай Константинович Васильев | orcid.org/0000-0003-3112-5482. E-mail: n.vasilev@innopolis.ru<br>Ведущий разработчик. |

**Финансирование:** Министерство цифрового развития, связи и массовых коммуникаций РФ и АО "Российская венчурная компания"(договор №004/20 от 20.03.2020, ИГК 0000000007119P190002).

**Для цитирования**: L. A. Merkin-Janson, R. M. Rezin, and N. K. Vasilyev, “Architecture of the Formally-Verified Distributed Ledger System InnoChain”, *Modeling and analysis of information systems*, vol. 27, no. 4, pp. 472-487, 2020.

## Введение

На сегодняшний день системы распределенного реестра (СРР) находят свое применение в различных направлениях информационных технологий и в различных секторах экономики, начиная с традиционного для СРР финансового сектора и заканчивая специфическими применениями в области робототехники, транспорта, связи и многих других. В Российской Федерации на государственном уровне СРР идентифицированы как одна из приоритетных «сквозных» информационных технологий, то есть таких, которые применимы практически во всех отраслях национальной экономики и в сфере государственного управления.

СРР представляют собой распределенные системы, функционирующие в реальном масштабе времени. К надежности СРР (включая корректность, безопасность, отказоустойчивость и иные качественные параметры, ассоциируемые с общим понятие надежности) предъявляются особые требования. Это связано с тем, что свойства СРР, с одной стороны, по определению обеспечивают доступ к данным для всех участников системы без ограничений, а также согласованность и целостность данных. С другой стороны, любой дефект в исходном коде СРР может обернуться серьезной уязвимостью для системы в целом, ввиду того, что нет единой точки доступа для устранения проблемы.

«Традиционные» методы обеспечения качества программного обеспечения (например, ручное и автоматическое тестирование, аудит программного кода, открытый доступ к коду для сообщества, проектирование по контракту) недостаточно эффективны для СРР. Например:

- Тестирование в принципе не может использоваться как основной метод обеспечения надежности систем реального времени, включая СРР, так как никакой объем тест-кейсов не обеспечит достаточного покрытия, ввиду того, что время является одним из входных параметров системы. Более того, использование тестирования в качестве основного метода обеспечения надежности систем реального времени скорее вредно, чем полезно, так как ведет к возникновению «ложного чувства безопасности» [1].
- Аудит кода и публикация открытого программного кода являются субъективными методами обеспечения качества ПО, надежность которых трудно оценить количественно. Кроме того, в случае осуществления аудита кода путем привлечения сторонних коммерческих организаций существует теоретическая возможность возникновения конфликта интересов, влияющих на исход аудита.
- Методология проектирования по контракту в целом позволяет улучшить раннее обнаружение и изоляцию ошибок ПО. Однако в случае СРР, множественные узлы системы часто обрабатывают одни и те же блоки данных. В случае возникновения ошибки, она потенциально может затронуть все узлы СРР сразу.

На протяжении последних 30 лет, основной методологией обеспечения качества систем реального времени с критическими требованиями по надежности являются различные формальные методы, например, методы формальной спецификации, верифицированной детализации и автоматической генерации программного кода [2].

Однако в области СРР формальные методы пока остаются менее популярными, чем «традиционные» методы, перечисленные выше. Одним из примеров использования формальных методов в СРР является система Tezos с предметно-специфическим языком смарт-контрактов Archetype [3], допускающим ручную и автоматическую верификацию свойств смарт-контрактов.

Тем не менее уязвимости в СРР могут возникать не только на уровне программной логики смарт-контрактов, но и на других архитектурных уровнях системы. Например, широко известны различные уязвимости в системе исполнения смарт-контрактов СРР Ethereum, проистекающие, в частности, из нарушений целостности стека виртуальной машины при некорректных сценариях вызовов рекурсивных функций [4]. Кроме того, уязвимости могут возникать в других функцио-

нальных компонентах, относящихся к узлу СРР, в протоколе консенсуса СРР, а также ввиду ошибок в коде окружения в котором функционирует и с которым взаимодействует узел сети СРР, например, в операционной системе (ОС).

Таким образом, для СРР, предназначенных для использования в критических по надежности индустриальных приложениях, необходимо систематическое применение методов формальной верификации не только на уровне логики смарт-контрактов, но и на всех вышеперечисленных архитектурных уровнях системы. Данный принцип был положен в основу архитектуры СРР InnoChain, которая в настоящий момент находится в стадии реализации. Одной из областей индустриального применения системы СРР InnoChain является распределенная система управления заправкой воздушных судов ПАО «Аэрофлот».

Настоящая статья представляет общую архитектуру СРР InnoChain и методы формальной верификации, применяемые на каждом из 5-ти уровней. Статья организована следующим образом. В разделе 1 приводятся общие сведения об уровнях архитектурной организации и формальной верификации СРР InnoChain. В разделе 2 рассматривается функционирование СРР InnoChain как распределенной системы реального времени и выбор протокола консенсуса. В разделе 3 рассматриваются вопросы выбора языка программирования для реализации СРР InnoChain, в частности, для реализации предметно-ориентированного языка смарт-контрактов. В разделе 4 рассматриваются вопросы выбора высоконадежного операционного окружения для СРР InnoChain. Раздел 5 представляет выводы и будущие задачи.

## 1. Обзор архитектуры СРР InnoChain и уровней формальной верификации

### 1.1. Уровень языка смарт-контрактов

С точки зрения пользователя, СРР InnoChain представляет собой распределенную децентрализованную базу данных со встроенным языком программирования — предметно-ориентированным языком смарт-контрактов, которые выполняются на всех узлах СРР и изменяют (дополняют) ее состояние.

Смарт-контракты разрабатываются на предметно-ориентированном языке, встроенном в язык CakeML [5]. В текущей версии СРР InnoChan, находящейся в процессе разработки, в качестве языка смарт-контрактов выступает сам CakeML. В перспективе предполагается возможность трансляции других языков программирования в абстрактное синтаксическое дерево, реализованное в терминах конструкций языка CakeML.

Для языка смарт-контрактов требуется наличие формальной семантики, позволяющей формулировать и доказывать свойства функциональной корректности смарт-контрактов. В частности, формальная операционная семантика существует для языка CakeML.

### 1.2. Уровень исполнения смарт-контрактов

В большинстве современных СРР, смарт-контракты транслируются в байт-код, исполняемый на виртуальной машине, входящей в состав каждого узла СРР. Однако реализация такой виртуальной машины для InnoChain потребовала бы, в частности, разработки формальной семантики байт-кода и формально-верифицированного компилятора из языка смарт-контрактов в байт-код.

Вместо этого, в InnoChain используется прямая трансляция кода смарт-контрактов (реализованных на языке CakeML или на языке, встроенном в CakeML) в машинный код процессора x86_64 (возможна также трансляция в машинный код ARM) с использованием верифицированного компилятора CakeML. Это гарантирует функциональную корректность генерируемого кода.

### 1.3. Уровень функционала узла

Узел СРР InnoChain осуществляет такие функции, как сетевые взаимодействия, хранение текущего состояния распределенного реестра и исполнение смарт-контрактов. Функционал узла также

реализуется на CakeML, что делает возможным формальную верификацию корректности алгоритмов узла и генерацию машинного кода узла с помощью формально-верифицированного компилятора CakeML.

### 1.4. Уровень протокола консенсуса СРР

Целостность состояния данных в СРР обеспечивается с помощью протокола консенсуса, исполняемого узлами системы. В качестве протокола консенсуса в InnoChain выбран HotStuff BFT [6], в отношении которого осуществлена формальная верификация свойства *Safety* (см. Раздел 2).

### 1.5. Уровень операционной системы

Для высоконадежных приложений СРР желательно использование операционного окружения, которое также является формально-верифицированным. С этой целью, наряду с ОС Linux, в СРР InnoChain может использоваться формально-верифицированное микроядро seL4 [7]. Более подробно этот вопрос рассмотрен в Разделе 4.

## 2. Функционирование системы распределенного реестра и выбор протокола консенсуса

Под системой распределенного реестра (СРР) обычно понимают множество вычислительных станций (далее узлов), объединенных в общую сеть с целью изменения единого состояния на основе команд от пользователей системы — транзакций. Структуры данных, описывающие состояние и транзакции СРР, варьируются в зависимости от целей, для которых используется система.

В случае СРР InnoChain, данные структуры определяются как изображено на Рис. 1. В структуре данных аккаунта хранится: идентификатор (`accountId`), множество открытых ключей (`publicKeys`, для отправки транзакций от имени аккаунта ее необходимо подписать каждым соответствующим секретным ключом), тип аккаунта (`type`), количество транзакций аккаунта (`nonce`), код смарт-контракта (`code`), хранилище дополнительных данных аккаунта (`storage`). В структуре данных транзакции хранится: время создания (`timestamp`), идентификатор аккаунта, от имени которого отправлена транзакция (`from`), идентификатор аккаунта для развертывания смарт-контракта (`to`), порядковый номер транзакции (`nonce`, который должен соответствовать полю nonce аккаунта отправителя), начальное состояние смарт-контракта (`data`).

Как можно заметить, *состояние* СРР определяется как частичная функция $s$, ставящая в соответствие идентификатору в формате GUID некоторое значение структуры данных *Account*:

$$s : GUID \longrightarrow Account \tag{1}$$

где $GUID = \{0,1\}^{128}$

Функция $s$ является частичной ввиду того, что не для каждого идентификатора могут существовать данные аккаунта.

Состояние СРР изменяется при исполнении транзакций, которые для оптимизации скорости их обработки группируются в `блоки`. Корректность изменения состояния СРР контролируется `протоколом консенсуса` — набором правил обмена сообщениями, которым должен следовать узел для поддержания правильного состояния системы. На основе проведенного анализа, для реализации в InnoChain была выбрана интерпретация протокола консенсуса HotStuff BFT [6] (далее протокол консенсуса) от проекта Libra [8].

Будем называть узлы, которые следуют протоколу консенсуса, `правильными`, а которые отклоняются от него — `неисправными`. Предполагается, что $N \geq 3f + 1$ узлов, где $f$ - максимальное число неисправных узлов. Узлы, объединенные в общую сеть, согласуют между собой изменение состояния распределенного реестра на основе блоков транзакций. Также предполагается, что верно

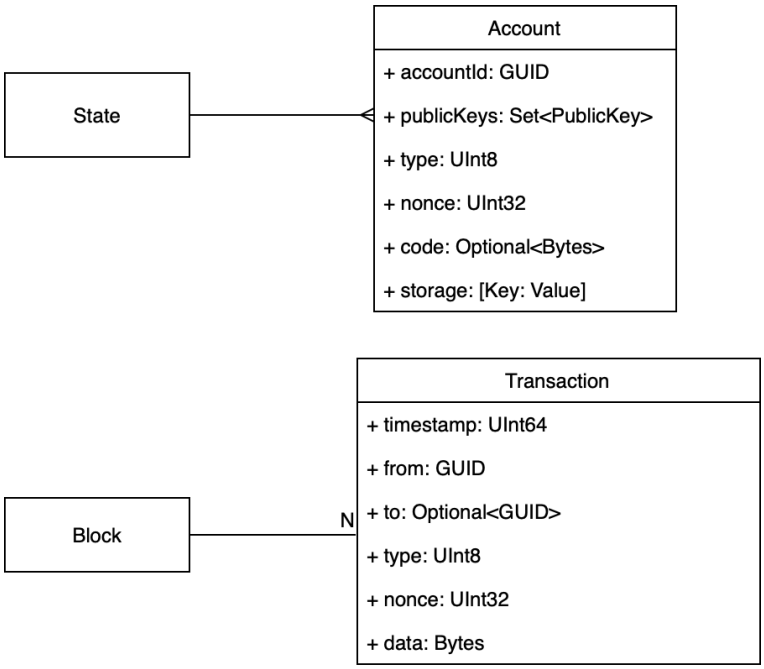


**Fig. 1.** Principal data structures of the InnoChain distributed ledger system

**Рис. 1.** Основные структуры данных CPP InnoChain

условие частичной синхронности: время доставки сообщения от одного узла другому не превосходит $\Delta$ после некоторого момента времени стабилизации сети (*GST, Global Stabilization Time*). Наконец, считаем, что вычислительной мощности узла недостаточно для того, чтобы «взломать» электронно-цифровую подпись или найти коллизию хэш-функции. Если описанные предположения верны, то должно гарантироваться, что протокол консенсуса удовлетворяет свойствам *Safety* и *Liveness*. *Safety* говорит о том, что для двух различных правильных узлов существует момент времени $t \geq GST$, когда последовательности принятых ими блоков совпадут, а до этого момента одна последовательность является префиксом другой. Свойство *Liveness* означает, что если правильный узел получил транзакцию, которая не противоречит состоянию сети, то обязательно узлами будет принят новый блок, который изменит состояние сети.

Обозначим через $S$ множество всевозможных состояний CPP. Тогда под исполнением транзакции можно понимать функцию E, которая преобразует транзакцию и предыдущее состояние в новое состояние CPP:

$$E : Transaction \times S \longrightarrow S \tag{2}$$

Под смарт-контрактом обычно понимают программу, написанную на специальном языке программирования для децентрализованного исполнения узлами CPP. Сторонние разработчики имеют возможность развернуть новые смарт-контракты в уже запущенной сети CPP. При этом машинный код, полученный после компиляции программы, и начальное состояние смарт-контракта сохраняются в структуре данных соответствующего аккаунта CPP. Данный машинный код загружается в память и выполняется узлом при исполнении специальных транзакций, содержащих сигнатуру функции смарт-контракта и параметры для ее вызова.

Важным процессом при обработке вызовов смарт-контракта является загрузка и исполнение кода смарт-контракта. Ввиду того, что узлы могут работать в разном окружении (например, использовать разные операционные системы), создание отдельного процесса смарт-контракта и передача ему управления для выполнения функции автоматически устанавливает зависимость корректно-

сти работы программы от ее окружения. По этой причине наиболее популярным подходом для исполнения смарт-контрактов является их интерпретация специально разработанной виртуальной машиной. Более того, существуют CPP, для которых была верифицирована функциональная корректность виртуальной машины. Тем не менее, данный подход не гарантирует корректность взаимодействия между узлом CPP и операционной системой, например, для передачи данных по сети. В разделе 4 рассматривается возможность использования формально верифицированной операционной системы, что делает возможным использование первого подхода для исполнения смарт-контрактов, а также увеличивает надежность взаимодействия между процессом узла и операционной системой.

Однако помимо использования надежной операционной системы, необходимо также обеспечить функциональную корректность работы узла и смарт-контрактов. Для достижения данной цели необходимо выбрать набор инструментов формальной верификации и подходящий для этого язык программирования. Как будет показано в разделе 3, наиболее подходящим на момент написания работы является язык программирования CakeML [9] в связке с программным инструментом дедуктивных методов верификации HOL4 [10].

На Рис. 2 изображена верхнеуровневая схема формальной верификации CPP. Описание протокола консенсуса HotStuff BFT, являющееся отдельной частью общей спецификации логики работы узла, преобразуется в модель для верификации свойств безопасности методом Model Checking [11]. В то же время алгоритмы функционирования узла разрабатываются на языке CakeML. С одной стороны, это позволяет получить гарантии формально верифицированной компиляции, а с другой стороны — иметь возможность формальной верификации свойств исходного кода в HOL4, используя метод характеристических формул [12].

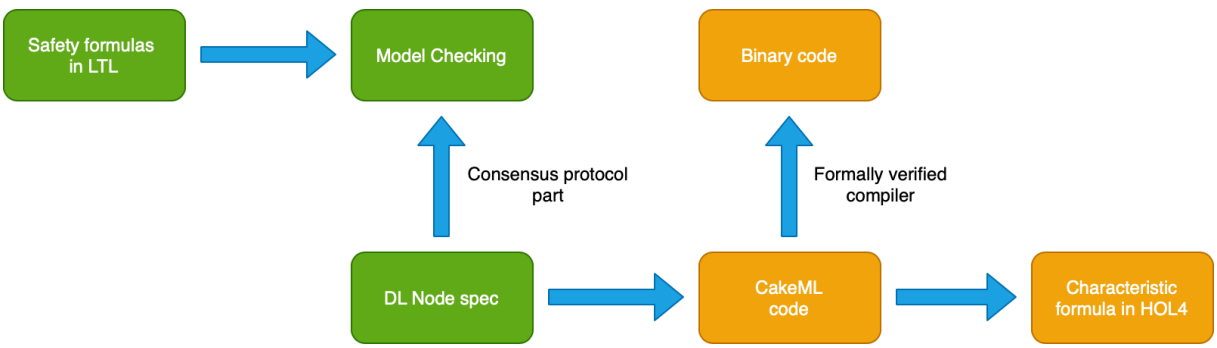


**Fig. 2.** High-level view of the formal verification scheme for the InnoChain distributed ledger system

**Рис. 2.** Верхнеуровневое представление схемы формальной верификации системы распределенного реестра

## 3. Выбор языка программирования

Необходимым условием для возможности формальной верификации программ на некотором языке программирования является наличие формальной семантики и программных инструментов для автоматизации проведения доказательств функциональной корректности. В рамках данной работы необходимо было выбрать язык, который одновременно подходил бы для разработки логики узла CPP и также мог бы служить в качестве языка смарт-контрактов. Таким образом, было исследовано четыре направления, изображенных на Рис. 4.

Три из четырех направлений связаны с языком CakeML. CakeML — это язык программирования, близкий к Standard ML. Формальная семантика CakeML [9] выражена в терминах логики высшего порядка (higher-order logic, HOL) и реализована в программном обеспечении для автоматизации процесса доказательств дедуктивным методом — HOL4 [10]. Для CakeML научным сообществом

разработан набор инструментов и теорий, реализованных в терминах HOL4. Основным инструментом является транслятор термов [13] из системы HOL4 в абстрактные синтаксические деревья CakeML. Поскольку семантика CakeML описана в HOL4, то появляется возможность записывать любые программы на CakeML в HOL4 — о таких программах говорят, что они «глубоко вложены» в HOL. Таким образом, определения, функции и типы данных в логике HOL можно передать на вход транслятору — и получить их CakeML-аналоги, глубоко вложенные в HOL. Важно отметить, что вместе с кодом на CakeML транслятор генерирует сертификат — доказательство того, что функции, записанные в HOL, работают так же как и полученный код на CakeML.

Обычные функции в HOL4 описывают «чистые» математические вычисления, то есть такие, в которых нет ввода-вывода и взаимодействия с внешним миром. В рамках проекта CakeML была разработана do-нотация для HOL4, которая позволяет описывать программы, в которых присутствует ввод-вывод, непосредственно в HOL4. Также создан монадический транслятор [14] — аналог обычного транслятора, но позволяющий транслировать функции HOL4, использующие do-нотацию. Таким образом, можно прямо в HOL4 записать программу, которая получает ввод с клавиатуры, обрабатывает введенные данные и выводит результаты на экран, а затем воспользоваться монадическим транслятором, чтобы автоматически получить аналогичную программу на CakeML (глубоко вложенную в HOL).

Другим важным подходом для анализа программ на CakeML является метод характеристических формул [12]. Данный инструмент позволяет формально верифицировать свойства корректности функций, изначально реализованных на CakeML, а не на HOL4. Свойства корректности предлагается задавать в виде формул сепарационной логики [15].

Авторами CakeML также разработан формально верифицированный компилятор для данного языка. Идея реализации этого компилятора, изображенная на Рис. 3, состоит в том, чтобы сначала записать компилятор в HOL4, а затем использовать транслятор для получения эквивалентного кода на CakeML. Затем этот код на CakeML подается на вход компилятору, реализованному на HOL4, и тем самым на выходе получается машинный код компилятора с гарантией функциональной корректности ввиду доказательств в HOL4.


**Fig. 3.** Sequence of transformations for generation of a formally-verified machine code using the CakeML compiler and HOL4 proofs.

**Рис. 3.** Последовательность преобразований для получения формально верифицированного машинного кода компилятора языка CakeML на основе доказательств в HOL4

Параллельно с HOL4 была рассмотрена популярная в научном сообществе система Isabell/HOL [16]. Идея была все та же: записать программу в Isabelle/HOL, а затем использовать формально верифицированный транслятор [17] для получения эквивалентного кода на CakeML. Однако данный подход оказался еще не готов к практическому применению: получаемое после трансляции абстрактное синтаксическое дерево программы приходится дорабатывать вручную для успешной компиляции компилятором CakeML. В частности, код «с эффектами» (например, с вводом-выводом) не поддерживается этим транслятором. Но более существенной проблемой является то, что структуры данных Isabelle (в частности, списки) не транслируются в структуры данных CakeML.

Наконец, последней опцией была возможность использовать систему Coq. Для данной системы научным сообществом был разработан фреймворк Verified Software Toolchain (VST) [18]. Он

представляет собой набор инструментов и библиотек для Coq, созданный с целью верификации программ, написанных на ограниченном подмножестве языка C (Clight). Проект VST тесно связан с другим проектом верификации на Coq — формально верифицированным компилятором CompCert языка C [19]. Несмотря на то, что большинство исследований, связанных с VST, ориентированы на язык C, изначально этот фреймворк разрабатывался как универсальный инструмент, позволяющий гарантировать, что утверждения статического анализатора относительно программы на некотором языке сохраняются для машинного кода, который исполняется в рамках некоторой операционной системы.

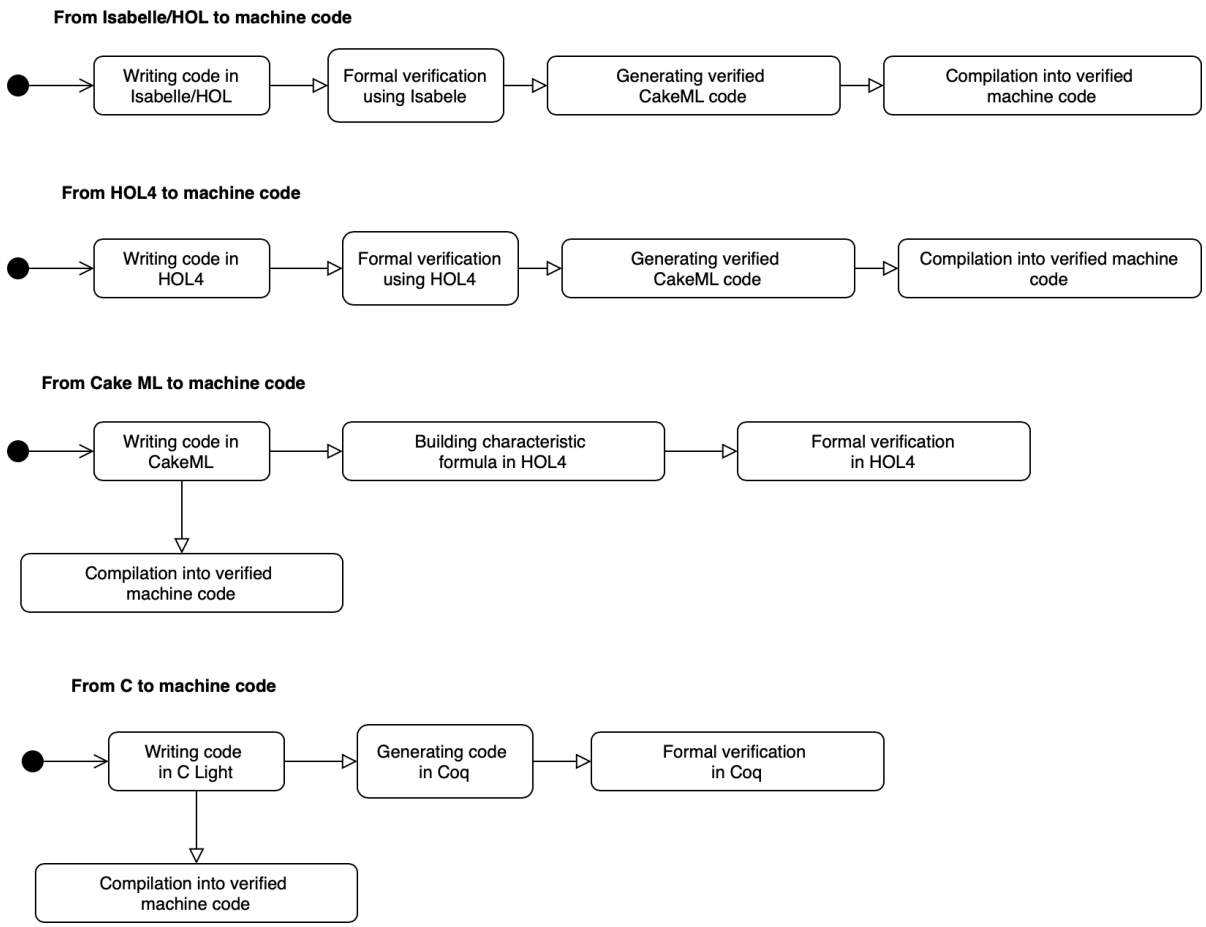


**Fig. 4.** The technologies and tools considered as the basis for implementation of the formally-verified distributed ledger system InnoChain

**Рис. 4.** Рассмотренные варианты технологий и инструментов для разработки формально верифицированной СРР

В виду совместимости с формально верифицированной операционной системой и функциональной парадигмой, которая на практике дает больше гарантий корректности кода, в качестве языка для разработки узла СРР, а также в качестве базового языка смарт-контрактов был выбран CakeML, а не Clight. В качестве системы для проведения формальных доказательств корректности программ на CakeML была выбрана система HOL4 ввиду того, что транслятор из Isabelle в CakeML еще не готов к практическому применению. Более того, для упрощения разработки смарт-контрактов было принято решение реализовать язык смарт-контрактов как предметно-ориентированный язык (DSL), специфичный для каждой отдельной области применения. В частности, свой язык смарт-контрактов разрабатывается для задачи автоматизации управления заправкой воздушных судов в

ПАО «Аэрофлот». Далее, для каждого DSL необходимо разработать транслятор для преобразования смарт-контрактов в эквивалентную программу на языке CakeML. Этот транслятор можно реализовать непосредственно в HOL4 и там же верифицировать его функциональную корректность. Эта идея уже была успешно применена разработчиками CakeML для реализации формально верифицированного компилятора для него. Таким образом, формально верифицированный транслятор из DSL в CakeML позволит использовать существующие инструменты для анализа корректности смарт-контрактов на DSL.

Тем не менее, удобная для разработчиков смарт-контрактов реализация DSL требует глубокого анализа соответствующей отрасли применения. В связи с этим, было решено реализовать первый смарт-контракт для конкретной задачи управления заправкой воздушных судов на самом CakeML, а затем, проанализировав это решение, прийти к понимаю конструкций, которые необходимо включить в DSL. В конечном итоге, разработчики смарт-контрактов получат возможность писать код как на DSL, так и напрямую на CakeML.

## 4. Выбор операционной системы

ОС играет важную роль в работе узла, предоставляя возможность взаимодействовать с другими узлами через сетевой интерфейс и сохранять данные, используя файловую систему. К сожалению, существующие операционные системы, получившие широкое распространение, разрабатывались без использования формальных методов верификации и имеют большую кодовую базу. Наиболее популярная на сегодняшний день операционная система Linux содержит порядка двадцати миллионов строк исходного кода. Несмотря на то, что за последние годы были сделаны попытки формально верифицировать отдельные части ядра данной операционной системы [20], бóльшая часть кода остается без формализации. Данный факт, помимо рисков возникновения дефектов при взаимодействии между узлом CPP и операционной системой, также затрудняет процесс формальной верификации исходного кода узла, ввиду необходимости построения формальной модели объекта (ОС), с которым осуществляется взаимодействие.

С другой стороны, микроядро seL4 [7] имеет кодовую базу порядка десяти тысяч строк, и его функциональная корректность была формально верифицирована для наиболее популярных на сегодняшний день архитектур процессоров и платформ. Данный факт позволяет сформировать гипотезу о возможности функционирования узла CPP с формально верифицированным исходным кодом под управлением операционной системы, код которой также формализован и формально верифицирован. Тем не менее, следует учитывать тот факт, что на данный момент микроядро seL4 еще не получило широкого распространения. По этой причине может потребоваться разработка нового или доработка существующего функционала seL4, необходимого для корректной работы узла CPP, включая сетевое взаимодействие и взаимодействие с базами данных.

В данной работе была исследована возможность использования seL4 в качестве операционной системы узла CPP. Кроме того, был разработан инструментарий для упрощения развертывания приложений для seL4 на реальном оборудовании, который может быть полезен в других исследованиях.

В отличие от полноценных ОС, таких как Linux, seL4 — это операционная система на базе микроядра. Это означает, что в режиме «ядра» исполняется урезанный функционал, а все остальные функции исполняются в «пользовательском» режиме. В случае с seL4, на уровне ядра реализована следующая функциональность: взаимодействие между процессами (IPC), управление потоками и виртуализация памяти. Преимуществом данного подхода является тот факт, что объем доверенного кода, который служит основой для работы приложений, значительно ниже чем у «традиционных» операционных систем и, таким образом, содержит потенциально меньше ошибок. Кроме того, меньший объем кодовой базы проще формализовать и для него формально верифицировать выполнение критических свойств системы, что и было сделано авторами seL4.

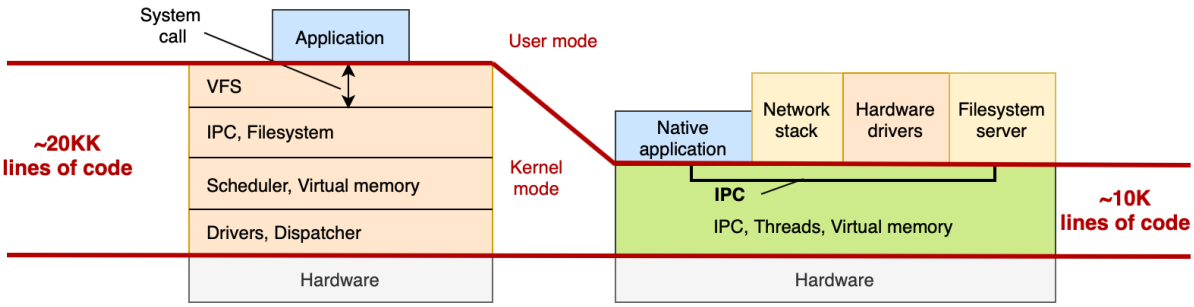


**Fig. 5.** Comparison of the seL4 microkernel with "traditional" monolithic kernel OSes.

**Рис. 5.** Сравнение seL4 с «традиционными» ОС

Как можно заметить из Рис. 5, seL4 содержит доверенную кодовую базу, которая исполняется в режиме «ядра» и наиболее критична с точки зрения безопасности. Вместе с тем, seL4 не содержит сервисов для доступа к оборудованию, а вместо этого предлагается реализовать эти сервисы как приложения для работы в «пользовательском» режиме. Существенным недостатком данного подхода является то, что стандартные компоненты для работы с оборудованием (файловая система, работа с сетью) необходимо либо разрабатывать «с нуля», либо портировать их из других ОС, но в любом случае это требует существенных трудозатрат. Тем не менее, в данном направлении сообществом seL4 уже проделана большая работа. Например, была адаптирована библиотека `picotcp` [21] для использования приложениями сетевого стека TCP/IP. Но все же в случае с узлом CPP остается открытым вопрос об интеграции приложений на базе seL4 с базами данных.

Взаимодействие между приложениями осуществляется одним из следующих способов: защищенные процедурные вызовы (RPC), общая память (Shared Memory) или сигналы (Notifications). По историческим соображениям данные способы взаимодействия объединяются под общим названием IPC (Inter-Process Communication). При взаимодействии между приложениями, seL4 выступает в качестве медиатора, отвечая за контроль доступа и корректность передачи параметров. Таким образом, работу системы под управлением seL4 можно рассматривать как работу отдельного приложения в «песочнице», при этом взаимодействие с остальными приложениями (включая драйверы оборудования) осуществляется через описанные механизмы IPC микроядра.

Формальное доказательство корректности seL4 выполнено его авторами с использованием системы верификации Isabelle/HOL [16] и изначально насчитывало 200 000 строк кода. Для формализации подмножества языка C, на котором написано микроядро, в Isabelle был записан синтаксический анализатор, который трансформирует исходный код в семантически эквивалентную запись на языке математической логики HOL. После формализации и преобразования требований к seL4 в формулы спецификации, формальная верификация проводилась с использованием инструментов Isabelle. Однако доказательство корректности кода на C еще не означает, что компилятор не привнесет ошибок в машинный код, так как для компиляции seL4 использовался стандартный компилятор gcc, а не формально-верифицированный компилятор CompCert.

Для доказательства корректности машинного кода в Isabelle были записаны трансляторы для преобразования исходного кода на языке C и машинного кода в графовое представление (control flow) с сохранением семантики. Таким образом, для доказательства корректности работы микроядра остается доказать семантическую эквивалентность полученных графовых моделей. Для этой цели использовался набор SMT решателей [22]. На Рис. 7 приведены основные этапы доказательства эквивалентности машинного кода исходному коду. На Рис. 6 приведены основные этапы проведения общего доказательства корректности seL4, а также свойства корректности, формализованные в спецификации:

- конфиденциальность: приложение не сможет прочитать данные или извлечь информацию другим образом, если у него нет доступа на чтение;
- целостность: приложение не сможет изменить данные, если у него нет доступа для этой операции;
- доступность: одно приложение не может помешать другому приложению использовать ресурс, если у второго есть на это доступ.

Конечно, формальное доказательство требует некоторых предположений о контексте, в котором оно проводится, а также о контексте, в котором работает микроядро. Данные предположения должны быть явно закреплены. Важно отметить, что такой подход может помочь выявить проблемы корректности системы на ранней стадии, ввиду четкого понимания ограничений, исходящих от закрепленных предположений о работе системы.

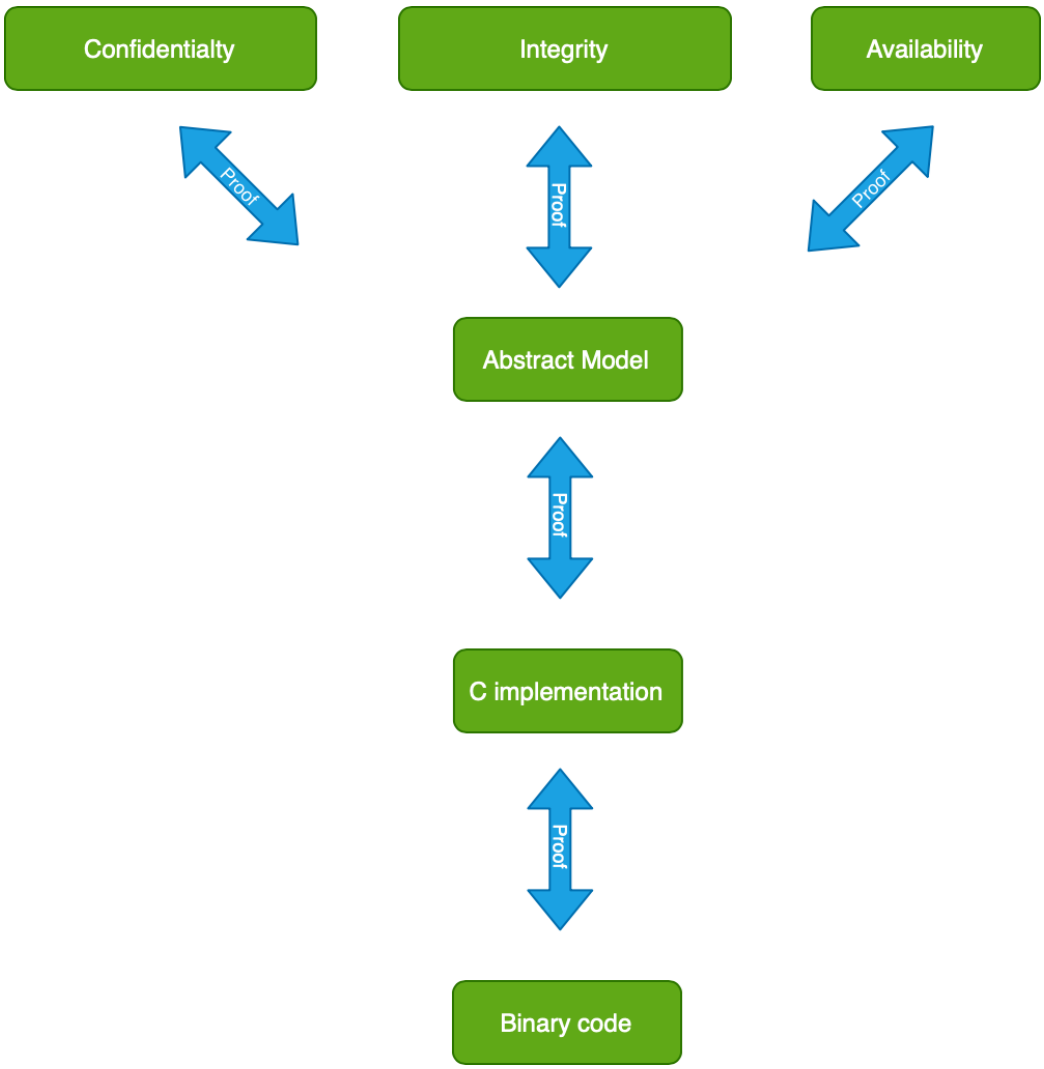


**Fig. 6.** Main stages of seL4 correctness verification

**Рис. 6.** Основные этапы проведения доказательства корректности seL4

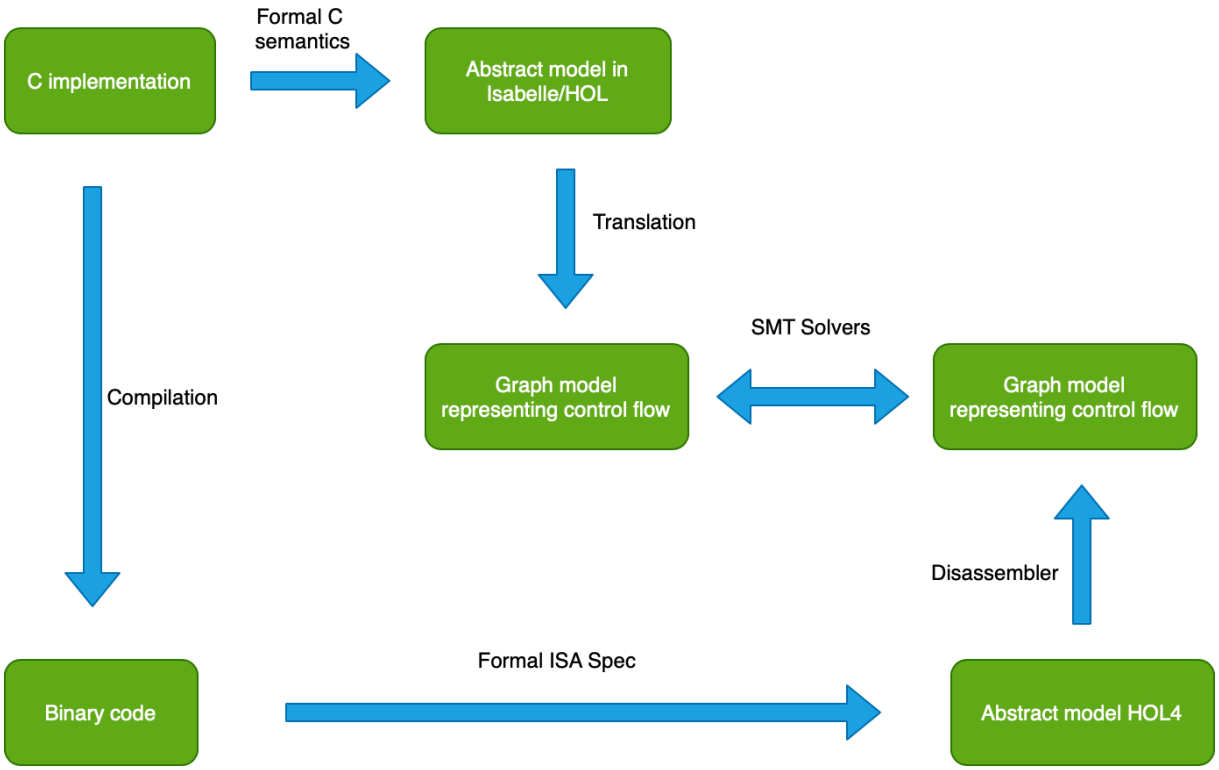


**Fig. 7.** Main stages of proving the functional equivalence of seL4 machine code and its source C code

**Рис. 7.** Этапы проведения доказательства эквивалентности машинного кода seL4 исходному коду на языке C

Формальная верификация seL4 его авторами была проведена с учетом следующих предположений:

- оборудование функционирует корректно;
- спецификация удовлетворяет ожиданиям — это самое сложное, так как не всегда легко удостовериться в том, что математическая формула действительно соответствует тому, что написано в требованиях к системе;
- сама система верификации функционирует корректно;
- код загрузки seL4 корректен: 1200 строк кода отвечающего за старт/загрузку seL4 не верифицировано;
- виртуальная память: механизм разделения памяти между ядром и приложениями, а также между приложениями, верифицирован в предположении о стандартном поведении виртуальной памяти при исполнении кода ядра;
- ассемблер: около 340 строк кода для доступа к оборудованию на ассемблере не верифицировано;
- прямой доступ к памяти (DMA): при формальной верификации было сделано предположение о том, что только процессор и блок управления памятью (MMU) имеют доступ к памяти; таким образом, DMA либо должен отсутствовать вовсе, либо нужно сделать предположение о корректной работе драйвера для этого контроллера;
- модель оборудования учитывает все каналы передачи информации для доказательства конфиденциальности;
- в микроядре имеется статический набор процессов, не изменяющийся после развертывания и запуска системы.

Однако стоит заметить, что машинный код может отличаться в зависимости от архитектуры процессора, поэтому необходимо провести формальную верификацию для каждой интересующей конфигурации в отдельности. Кроме того, seL4 может функционировать и в режиме гипервизора, поэтому желательно осуществить верификацию также и этого режима. В настоящее время, полный процесс формальной верификации, описанный выше, был проведен только для архитектуры ARMv7 (Sabre Light), а для x86-64 формально верифицирована функциональная корректность.

На момент написания работы существуют два способа развертывания приложений на базе seL4. Первый способ заключается в использовании очень гибкого фреймворка Genode [23]. Он позволяет запустить образ микроядра, а затем динамически компилировать и исполнять приложения, аналогично тому, как это делается в «традиционных» ОС. Тем не менее, ввиду того, что Genode поддерживает и другие микроядра, только общая для всех ядер функциональность доступна пользователям этого фреймворка. Например, при развертывании приложений seL4 на основе Genode не будет доступен режим гипервизора. Более того, использование Genode исключает гарантии формальной верификации seL4, так как эти гарантии основаны на предположении о статическом наборе процессов в системе.

Второй способ заключается в сборке и запуске статической конечной системы на базе seL4 с использованием фреймворка CAmkES [24]. Этот фреймворк позволяет описать компоненты системы и взаимодействия между ними на специальном языке CAmkES ADL (Architecture Description Language). Базовые элементы описания следующие:

- **Компоненты** — приложения, в общем случае, запускаемые в виде отдельных процессов (но есть возможность объединить несколько компонентов в одно приложение) и взаимодействующие между собой через механизмы IPC.
- **Интерфейсы** — набор сигнатур функций для возможности указания того, как один компонент может обратиться к другому; на данный момент поддерживаются следующие типы интерфейсов: функция (RPC), общая память (Shared Memory), сигналы (Notifications).
- **Коннекторы**, которые связывают взаимодействующие компоненты друг с другом, например, импортируемый интерфейс одного компонента с экспортируемым интерфейсом другого.

В рамках процесса приложения создаются отдельные потоки для основного сценария исполнения и для каждого обработчика взаимодействия с другими компонентами. Для синхронизации потоков поддерживаются стандартные примитивы: семафоры и мьютексы.

## 5. Заключение и дальнейшие задачи

В данной работе предложена 5-уровневая архитектура CPP InnoChain, допускающая применение методов формальной верификации на каждом из уровней, с целью достижения более высокого уровня надежности CPP, чем было бы возможно при использовании «традиционных» методов обеспечения качества ПО, ранее применявшихся к CPP.

В настоящее время CPP InnoChain находится в стадии реализации. В частности, разработан набор инструментов для автоматического развертывания приложений для seL4 на реальном оборудовании [25].

Для завершения реализации необходимо будет решить следующие дополнительные задачи.

Несмотря на то что формальная верификация seL4 гарантирует свойство функциональной корректности, для микроядра не реализована библиотека для интеграции с базами данных. Одним из возможных путей решения этой проблемы предлагается использовать seL4 с включенным режимом гипервизора для взаимодействия с приложением базы данных, запущенном на виртуальной машине под управлением ОС Linux. Для реализации сетевых взаимодействий между узлами CPP предлагается использовать адаптированную под seL4 библиотеку `picotcp`[21].

Более существенной проблемой является требование по исполнению машинного кода смарт-контракта при обработке транзакций. Ввиду ограничений CAmkES ADL, связанных с формальной

верификацией контроля доступа приложений, на базе seL4 разрешается развертывать исключительно приложения со статической конфигурацией, что означает невозможность порождения дополнительных процессов после запуска приложения. В частности, это означает невозможность динамического развертывания новых смарт-контрактов в системе на основе seL4. В качестве начального решения в данном направлении предлагается линковать смарт-контракты на CakeML с основным приложением узла на этапе развертывания. В качестве альтернативного решения предполагается отдельно исследовать фреймворк Genode, который позволяет динамически запускать новые приложения для seL4, ценой отказа от некоторых гарантий формальной верификации микроядра.

## References


[1] A. Burns and A. Wellings, *Concurrent and Real-Time Programming in Ada 2005*. Cambridge University Press, 2007.

[2] A. Burns and A. Wellings, *Real-Time Systems and Programming Languages*. Addison–Wesley, 2009.

[3] *Smart Contracts Under Control*, 2020. [Online]. Available: https://archetype-lang.org.

[4] *More Ethereum Attacks: Race-To-Empty is the Real Deal*, 2016. [Online]. Available: https://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal.

[5] *CakeML: A verified implementation of ML*, 2020. [Online]. Available: https://cakeml.org.

[6] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, «HotStuff: BFT Consensus with Linearity and Responsiveness», in *PODC*, ACM, 2019, pp. 347–356.

[7] G. Heiser, *The seL4 Microkernel: An Introduction*, 2020. [Online]. Available: https://sel4.systems/About/seL4-whitepaper.pdf.

[8] *Libra by Facebook*. [Online]. Available: https://github.com/libra/libra.

[9] S. Owens, M. O. Myreen, R. Kumar, and Y. K. Tan, «Functional Big-step Semantics», in *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016*, P. Thiemann, Ed., ser. Lecture Notes in Computer Science, vol. 9632, Springer, Apr. 2016, pp. 589–615. DOI: 10.1007/978-3-662-49498-1"_23.

[10] T. Gauthier, C. Kaliszyk, and J. Urban, «TacticToe: Learning to Reason with HOL4 Tactics», in *LPAR*, ser. EPiC Series in Computing, vol. 46, EasyChair, 2017, pp. 125–143.

[11] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[12] J. A. Pohjola, H. Rostedt, and M. O. Myreen, «Characteristic Formulae for Liveness Properties of Non-terminating CakeML Programs», in *Interactive Theorem Proving (ITP)*, To appear, LIPICS, 2019. [Online]. Available: https://cakeml.org/itp19.pdf.

[13] M. O. Myreen and S. Owens, «Proof-producing Translation of Higher-order logic into Pure and Stateful ML», *Journal of Functional Programming*, vol. 24, no. 2-3, pp. 284–315, 2014. DOI: 10.1017/S0956796813000282.

[14] O. Abrahamsson, S. Ho, H. Kanabar, R. Kumar, M. O. Myreen, M. Norrish, and Y. K. Tan, «Proof-Producing Synthesis of CakeML from Monadic HOL Functions», Springer, 2020. [Online]. Available: https://rdcu.be/b4FrU.

[15] P. O'Hearn, «Separation Logic», *Communications of the ACM*, vol. 62, no. 2, pp. 86–95, 2019. DOI: 10.1145/3211968.

[16] L. C. Paulson, *Isabelle - A Generic Theorem Prover (with a contribution by T. Nipkow)*, ser. Lecture Notes in Computer Science. Springer, 1994, vol. 828.

[17] L. Hupel and T. Nipkow, «A Verified Compiler from Isabelle/HOL to CakeML», in *European Symposium on Programming (ESOP)*, ser. Lecture Notes in Computer Science, vol. 10801, Springer, 2018, pp. 999–1026. DOI: 10.1007/978-3-319-89884-1_35. [Online]. Available: https://lars.hupel.info/pub/isabelle-cakeml.pdf.

[18] A. W. Appel, «Verified Software Toolchain», in *European Symposium on Programming*, Springer, 2011, pp. 1–17.

[19] *CompCert project*. [Online]. Available: https://compcert.org.

[20] *Verification center of the operating system Linux*. [Online]. Available: http://linuxtesting.org/publications.

[21] *Implementation of the network layer for seL4*. [Online]. Available: https://github.com/SEL4PROJ/picotcp.

[22] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, «Handbook of Satisfiability», in, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, vol. 185, ch. Satisfiability Modulo Theories, pp. 825–885.

[23] *seL4 deployment using Genode*. [Online]. Available: https://genode.org/documentation/articles/sel4_parts_1.

[24] *Introduction to CAmkES*. [Online]. Available: https://docs.sel4.systems/Tutorials/hello-camkes-0.html.

[25] R. Rezin, *seL4 deploy scripts*, 2020. [Online]. Available: https://github.com/RZRussel/seL4-deploy-public.git.