

**Федеральное государственное автономное образовательное
учреждение высшего образования**

Университет ИТМО

Дисциплина: Программирование встраиваемых систем

Курсовая работа

**Работу выполнили,
студенты группы Р4219:**
Ануфриев Илья Владимирович,
Василев Васил Николаев,
Кадырин Вадим Юрьевич,
Суходей Ярослав Юрьевич

Преподаватель:
к.т.н., доцент ПИКТ
Ключев Аркадий Олегович

2025 г.
Санкт-Петербург

Содержание

| | |
|--|-----------|
| Введение | 2 |
| Техническое задание | 3 |
| Типовой сценарий использования системы | 4 |
| Ограничения и допущения | 5 |
| Архитектура | 6 |
| Поток данных от источника к выполнению | 6 |
| Модель параллелизма и синхронизации | 7 |
| Пример смешанной сети (Fibonacci) | 10 |
| Описание реализации | 11 |
| Грамматика языка Solace | 11 |
| Структура программы | 11 |
| Объявление узлов | 11 |
| Операторы в software-узлах | 11 |
| Операторы в hardware-узлах | 12 |
| Выражения и приоритеты операторов | 12 |
| Описание сети network | 12 |
| Лексические элементы | 13 |
| Компилятор (compiler) | 13 |
| Формат выходных файлов компилятора | 14 |
| Сетевая ВМ (vm/network) | 15 |
| Sniff-режим и CSV-логирование | 15 |
| Симулятор (vm/sim) | 15 |
| Стековая ВМ Harv (vm/harv) | 16 |
| Описание тестов | 17 |
| Тесты компилятора | 17 |
| ParserConstructsTest.kt | 17 |
| NetworkAnalysisTest.kt | 17 |
| HardwareVisitorTest.kt | 18 |
| SoftwareVisitorTest.kt | 19 |
| PackagingIntegrationTest.kt | 19 |
| Тесты симулятора (vm/sim) | 20 |
| Тесты стековой ВМ Harv (vm/harv) | 20 |
| Тесты сетевой ВМ (vm/network) | 20 |
| RuntimeTest | 20 |
| SimNodeVmFactoryTest | 20 |
| FibonacciNetworkTest | 21 |
| HarvNodeVmFactoryTest | 21 |
| DotOutTest | 22 |
| Вывод | 23 |

Введение

Современные встраиваемые и распределённые системы требуют не только высокой производительности, но и прозрачной модели взаимодействия программных и аппаратных компонентов. Традиционные подходы либо фиксируют архитектуру (жёстко разделяя hardware и software), либо не дают удобного формального описания сети взаимодействующих вычислительных узлов. В результате усложняется проектирование, тестирование и повторное использование решений.

Проект Solace предлагает экспериментальный язык программирования и семейство виртуальных машин, ориентированных на моделирование вычислений в терминах узлов и каналов сети Кана. Язык позволяет единообразно описывать как аппаратно-ориентированные (hardware) узлы, реализуемые в виде комбинационных схем, так и программные (software) узлы на основе стековой машины. Сетевая виртуальная машина восстанавливает из единого бинарного пакета топологию сети и запускает соответствующие узловые ВМ, связывая их через асинхронные FIFO-каналы.

Цель данной работы — описать архитектуру и реализацию языка Solace, компилятора и виртуальных машин, а также способы их тестирования. В отчёте приводится обзор формата выходных файлов, принципов построения сети Кана, моделей вычислений для аппаратных и программных узлов и методик верификации корректности работы подсистем.

Техническое задание

Основной объект разработки — экосистема языка Solace, включающая:

- язык описания узлов и глобальной сети (*.solace);
- компилятор (compiler), преобразующий исходный код в единый бинарный пакет *.solpkg;
- набор виртуальных машин:
 - симулятор аппаратных узлов (vm/sim);
 - стековая VM Harv для программных узлов (vm/harv);
 - сетевая VM (vm/network), строящая сеть Кана и запускающая узловые VM.

Ключевые функциональные требования к системе:

1. Язык и модель вычислений

- Поддержка двух типов узлов: hardware (комбинационная логика) и software (стековая машина).
- Наличие трёх типов портов: in, out, self.
- Разделение кода узла на секции init {} и run {}, где init выполняется однократно при запуске, а run — в цикле.
- Описание глобальной сети в секции network { ... } с явным перечислением соединений вида A.out -> B.in;.

2. Компилятор

- Синтаксический анализ исходного файла .solace с использованием ANTLR-грамматики Solace.g4.
- Семантический анализ:
 - сбор списка узлов, их типов и портов;
 - проверка корректности соединений и отсутствия ссылок на неизвестные узлы/порты;
 - проверка отсутствия дублирующихся портов и соединений;
 - диагностика неиспользованных выходных и self-портов в сети и коде.
- Генерация байткода:
 - для аппаратных узлов — в формате, исполняемом симулятором комбинационной логики;
 - для программных узлов — в формате байткода стековой машины Harv.
- Формирование единого пакетного файла *.solpkg (формат SOLP), содержащего:
 - заголовок с метаданными о пакете;
 - таблицу строк (имена узлов и портов);
 - поток инструкций NODE_DEF/CONNECT, описывающий сеть;
 - встроенные контейнеры байткода узлов формата solbc (SOLB).

3. Сетевая виртуальная машина (network VM)

- Загрузка файла формата SOLP с диска, в том числе с поиском относительно текущего каталога.
- Разбор таблицы строк и потока инструкций:
 - восстановление списка узлов с их типами и сигнатурами портов;
 - восстановление списка соединений между портами.
- Валидация структуры:
 - проверка magic-чисел, версий и размеров секций;
 - согласованность типов узлов в метаданных и контейнерах solbc;
 - проверка корректности подключений всех портов.
- Построение сети Кана:
 - создание отдельных FIFO-каналов для всех соединений;
 - создание self-каналов для self-портов и их «разветвление» на вход и выход.
- Запуск узловых VM:
 - поддержка различных фабрик узловых VM (stub, симуляторная, Harv, смешанная);
 - управление временем работы сети, корректное завершение и сбор статистики.
- Поддержка режимов наблюдения:
 - текстовый sniff-режим с выводом всех сообщений;

- логирование трафика в CSV-формате с ограничением по числу записей.
- 4. **Симулятор аппаратных узлов (vm/sim)**
 - Реализация модели комбинационной логики:
 - набор базовых элементов (арифметические, логические, регистры, мультиплексоры, FIFO-очереди);
 - представление схемы в виде графа NetlistGraph.
 - Поддержка self-очереди для хранения состояния между тактами.
 - Определение статусов выполнения блока (SUCCESS, BLOCKED, ERROR) в зависимости от доступности данных во входных очередях и корректности вычислений.
- 5. **Стековая ВМ Harv (vm/harv)**
 - Реализация стековой архитектуры с поддержкой:
 - целых чисел и строк;
 - FIFO-очереди как объектов с блокирующим поведением;
 - идентификаторов и таблицы переменных;
 - меток и условных/безусловных переходов.
 - Набор инструкций для арифметики, логики, работы со стеком, управления выполнением, определения переменных, входа-выхода и получения размера FIFO.
 - Разделение байткода на инициализационную и основную фазу исполнения.
- 6. **Нефункциональные требования**
 - Реализация всех компонентов на языке Kotlin.
 - Модульная архитектура и чёткие интерфейсы между компилятором и ВМ.
 - Документированный формат выходных файлов (FILE_FORMATS.md).
 - Наличие автоматических тестов для ключевых подсистем.

Дополнительно ТЗ подразумевает, что язык и инфраструктура Solace должны быть пригодны как для учебных, так и для исследовательских задач: разработчик должен иметь возможность быстро описывать новые конфигурации сети, комбинировать аппаратные и программные узлы, анализировать структуру получившейся сети и воспроизводить эксперименты.

Типовой сценарий использования системы

С точки зрения конечного пользователя (разработчика), работа с системой Solace выглядит следующим образом:

1. На языке Solace описывается набор узлов (hardware и/или software) и глобальная сеть network, задающая соединения между ними.
2. Исходный файл program.solace передаётся компилятору:
 - компилятор выполняет синтаксический и семантический анализ;
 - по результатам анализа формируется пакет program.solpkg с описанием сети и встроенным байткодом всех узлов.
3. Полученный пакет загружается сетевой ВМ:
 - ВМ читает заголовок и мета-секцию, восстанавливая список узлов и соединений;
 - создаются FIFO-каналы и self-каналы для всех портов.
4. Пользователь выбирает режим исполнения:
 - заглушечный (stub) — для быстрой проверки топологии и корректности проводки портов;
 - симуляторный (sim) — для исполнения только аппаратных узлов;
 - программный (harv) — для программных сетей;
 - смешанный (mixed) — для одновременного запуска аппаратных и программных узлов.
5. Сеть запускается на ограниченное время или до завершения всех узлов; при необходимости включается sniff-режим или сбор CSV-логов трафика.

Такой сценарий обеспечивает единый жизненный цикл: от высокоуровневого описания сети до её исполнения и анализа поведения.

Ограничения и допущения

В ТЗ и спецификациях формата выходных файлов зафиксированы следующие важные ограничения:

- Таблица строк в пакете S0LP:
 - максимальное число строк — 65535;
 - максимальная длина одной строки в UTF-8 — 65535 байт.
- Порты узлов:
 - количество портов каждого типа (`in`, `out`, `self`) ограничено значением `u8` (не более 255);
 - имена узлов в пакете уникальны.
- Формат контейнера байткода S0LB:
 - чётко фиксирована структура заголовка (`magic` "S0LB", версии, тип узла, размеры секций `init` и `run`);
 - секции `init` и `run` могут иметь нулевой размер, если соответствующий блок в исходнике отсутствует.
- Типы данных:
 - аппаратные узлы оперируют целыми числами, соответствующими целочисленному типу в симуляторе;
 - программные узлы дополнительно поддерживают строки и абстракции FIFO-очереди.

Эти ограничения позволяют упростить реализацию загрузчиков, симулятора и стековой машины, а также гарантируют совместимость между версиями компилятора и ВМ.

Архитектура

Архитектура Solace опирается на трёхслойную модель:

1. Фронтенд (язык и компилятор)

- Исходный код *.solace описывает:
 - узлы (node) с типом hardware или software, портами in/out/self и секциями init/run;
 - глобальную сеть network, задающую соединения между портами.
- Грамматика ANTLR4 (Solace.g4) определяет синтаксис языка и используется для построения AST.
- Компилятор, работая поверх AST, выполняет:
 - анализ сети и проверку корректности топологии;
 - генерацию байткода для аппаратных и программных узлов;
 - упаковку результата в пакет S0LP с встроенными контейнерами S0LB.

2. Узловые виртуальные машины

- **Симулятор аппаратных узлов (vm/sim):**
 - представляет вычисления в виде графа NetlistGraph, где вершины — примитивные элементы (Adder, Multiplier, LogicAnd и др.), а рёбра — связи между ними;
 - поддерживает специальные элементы Mux2, Register, Fifo для организации потоков данных и хранения состояния;
 - выполняет вычисления, пропуская значения через граф и учитывая блокировки на FIFO.
- **Стековая ВМ Harv (vm/harv):**
 - реализует классическую стековую машину с таблицей переменных, системой типов и набором инструкций;
 - использует FIFO-очереди для взаимодействия с внешним миром и другими узлами;
 - отделяет фазу инициализации от основной фазы выполнения.

3. Сетевая ВМ (vm/network)

- Интерпретирует мета-секцию пакета S0LP как описание сети Кана:
 - узлы с типами, портами и привязками к контейнерам S0LB;
 - соединения между конкретными портами узлов.
- На базе этих данных строит структуру BuiltNetwork, содержащую:
 - каналы для всех соединений и self-портов;
 - набор узловых ВМ, созданных через выбранную фабрику (StubNodeVmFactory, SimNodeVmFactory, HarvNodeVmFactory, MixedNodeVmFactory);
 - вспомогательные компоненты для sniff-режима и записи CSV-логов.
- Запускает каждый узел в отдельной корутине, обеспечивая параллельное исполнение и обмен данными только через FIFO.

На рисунке приведена иллюстрация преобразования кода узлов в различные формы, в зависимости от их типов.

Логически архитектура обеспечивает слабую связанность компонентов:

- компилятор не знает о деталях реализации ВМ и лишь формирует согласованный бинарный формат;
- узловые ВМ не знают о структуре всей сети и работают только с локальными портами и очередями;
- сетевая ВМ отвечает за связывание узлов и организацию коммуникации, абстрагируясь от внутреннего байткода.

На рисунке приведена схематичная иллюстрация процесса компиляции.

Поток данных от источника к выполнению

Архитектурно жизненный цикл программы на Solace можно представить в виде последовательности шагов:

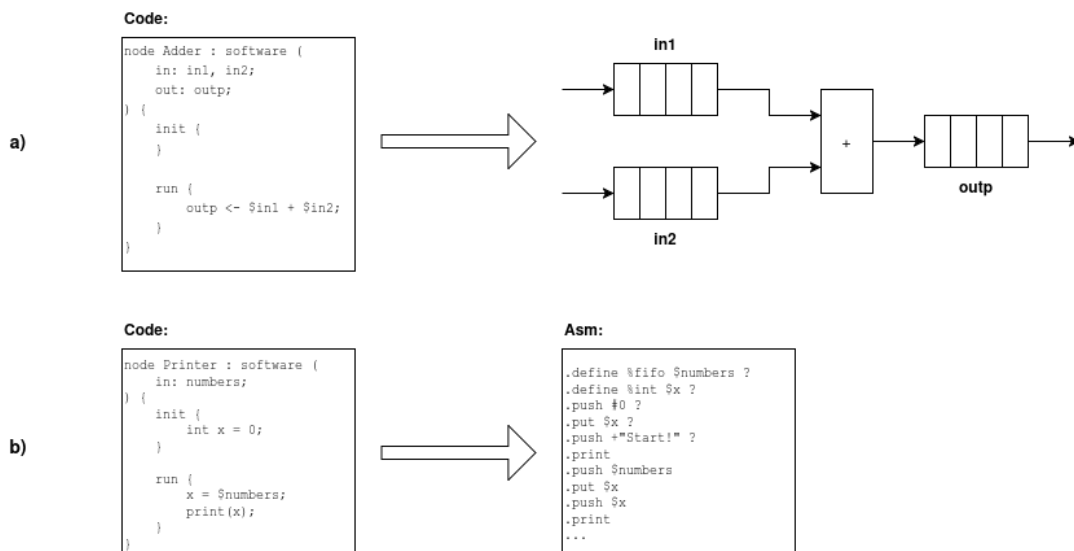


Рис. 1: Преобразование кода HW-узла (а), и SW-узла (б)

1. Парсинг и анализ:

- исходный текст *.solace разбирается по грамматике ANTLR;
- строится AST, на основе которого выполняется анализ сети и проверка корректности.

2. Генерация байткода:

- для каждого аппаратного узла формируется список инструкций симулятора и кодируется в строку/байты;
- для каждого программного узла формируется список инструкций Harv и аналогично кодируется.

3. Упаковка в пакет:

- строится таблица строк с именами узлов и портов;
- формируется поток инструкций NODE_DEF/CONNECT, описывающий структуру сети;
- в конец файла последовательно добавляются контейнеры S0LB с байткодом узлов.

4. Загрузка и построение сети:

- сетевая VM читает заголовок S0LP, таблицу строк и поток инструкций;
- создаёт структуру BuiltNetwork с узлами, портами и каналами.

5. Запуск узловых VM:

- выбранная фабрика VM создаёт конкретные экземпляры узловых VM (симулятор, Harv или заглушка);
- каждая VM запускается в отдельной корутине, обмен данными осуществляется через каналы.

Такое разделение позволяет независимо развивать язык, форматы файлов и реализации виртуальных машин.

На рисунке приведена схематичная иллюстрация процесса исполнения программы.

Модель параллелизма и синхронизации

Solace исходит из модели, где каждый узел — это независимый параллельный процесс, взаимодействующий с другими узлами только через FIFO-каналы:

- каждый узел выполняется в собственной корутине Kotlin;
- на уровне сети Кана отсутствует общая память: состояние хранится внутри узлов и в self-очередях;
- блокирующие операции `read$` и `write <-` служат единственным механизмом синхронизации.

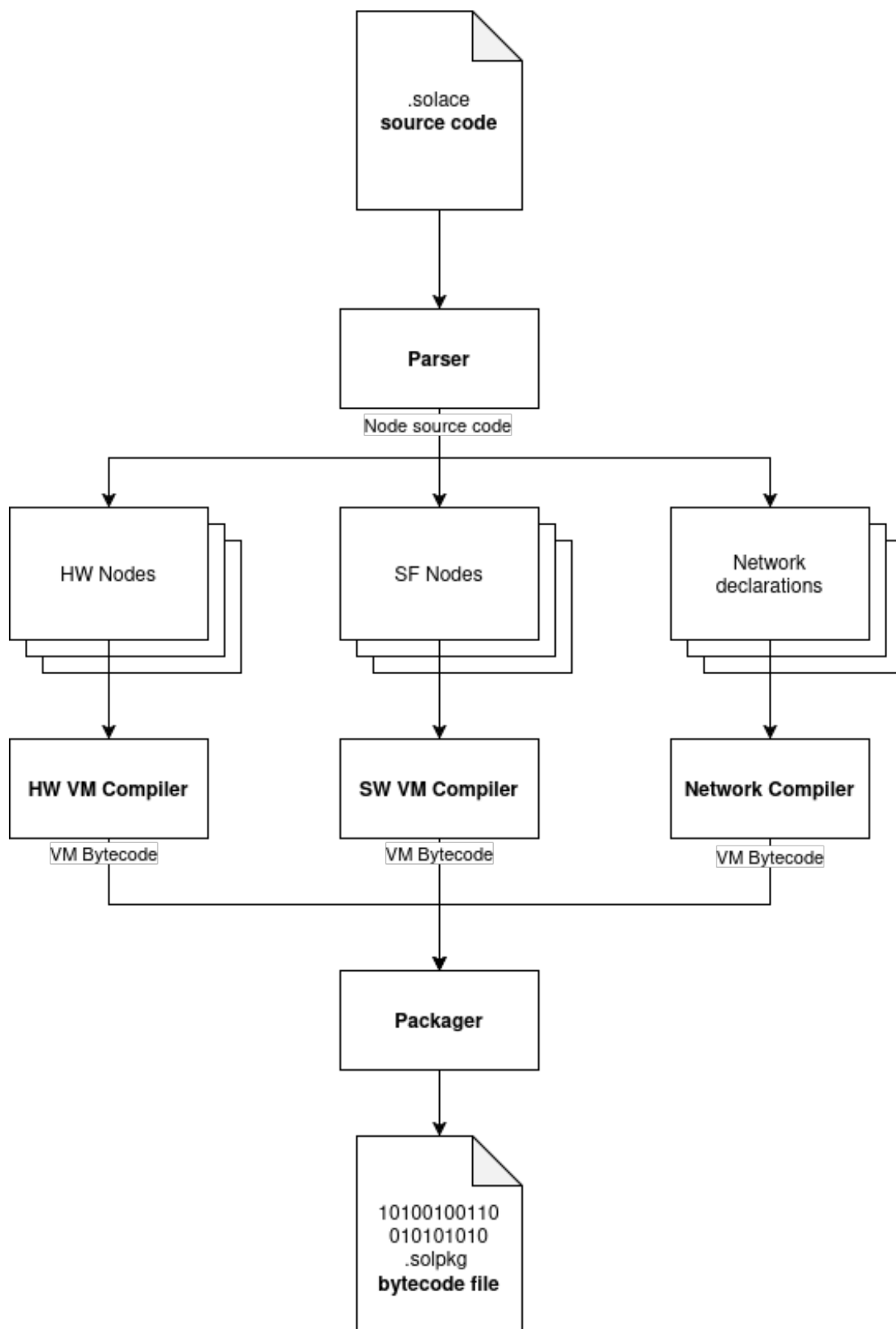


Рис. 2: Процесс компиляции в языке Solace

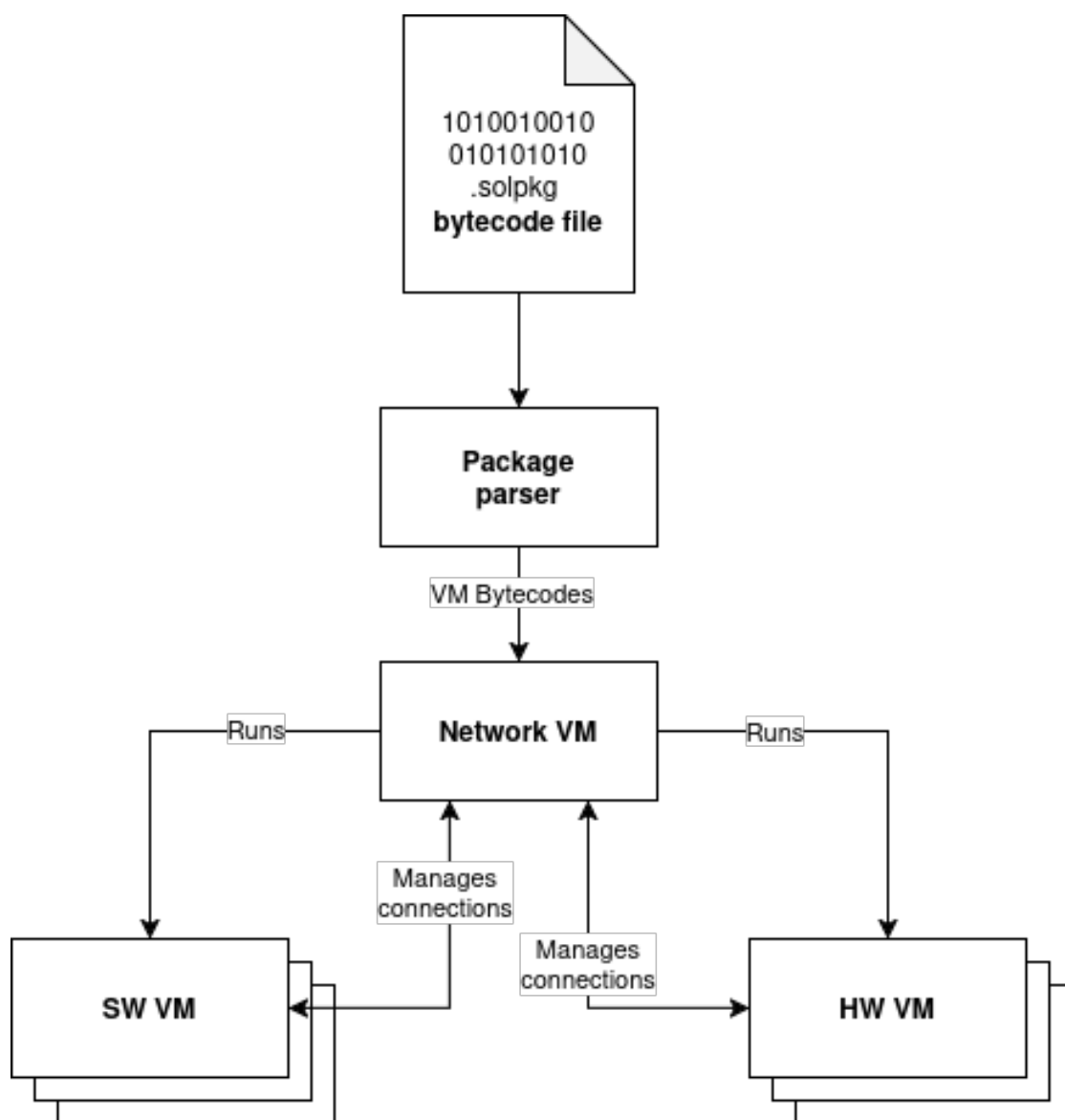


Рис. 3: Процесс исполнения байткода Solace

Это облегчает рассуждение о параллельных программах: отсутствие общей памяти снижает риск гонок, а использование FIFO-очереди делает явными все точки взаимодействия между компонентами сети.

Пример смешанной сети (Fibonacci)

В каталоге `examples` приведён пример смешанной программы `fibonacci_mixed.solace`, иллюстрирующий совместную работу аппаратных и программных узлов:

- `TickSource` : hardware — аппаратный узел-источник тактов:
 - в блоке `init` помещает начальное значение в self-очередь `loop`;
 - в блоке `run` на каждом шаге увеличивает счётчик и отправляет его в выходной порт `tick`, одновременно обновляя `loop`.
- `FibStepper` : software — программный узел, вычисляющий числа Фибоначчи:
 - использует вход `step` как «тактирующий» сигнал (чтение из FIFO блокирует выполнение до прихода данных);
 - хранит два последних значения последовательности в self-очередях `prev` и `curr`;
 - на каждом шаге вычисляет следующее число и отправляет его в порт `fib`.
- `FibPrinter` : software — программный узел-потребитель:
 - читает входной порт `inFib`;
 - выводит текущее значение при помощи операции `print`.

Секция `network` соединяет эти узлы в конвейер:

- `TickSource.tick -> FibStepper.step;`
- `FibStepper.fib -> FibPrinter.inFib;`

На уровне архитектуры данный пример демонстрирует:

- использование аппаратного узла для генерации событий/тактов;
- реализацию состояния и логики на программной стороне (Harv);
- передачу данных и синхронизацию исключительно через FIFO-очереди.

Описание реализации

Грамматика языка Solace

Синтаксис языка Solace формализован в виде ANTLR-грамматики `Solace.g4`. Ниже приводится текстовое описание основных конструкций, соответствующих правилам грамматики.

Структура программы

На верхнем уровне программа представляет собой последовательность описаний узлов и одной (глобальной) сети:

- правило `program`:
 - допускает любое количество объявлений узлов `nodeDecl` и объявление сети `networkDecl` в произвольном порядке;
 - файл всегда завершается маркером конца входа `E0F`.

Объявление узлов

Объявление узла имеет общий шаблон:

```
node Имя : hardware|software ( сигнатура_каналов ) {  
    init { ... }  
    run  { ... }  
}
```

В грамматике это задаётся правилом `nodeDecl`:

- для аппаратных узлов (`hardware`):
 - используется вариант с `hardwareInitBlock` и `hardwareRunBlock`;
 - тело `init/run` состоит из `hardwareStatement`, отражающих ограничения на допустимые конструкции;
- для программных узлов (`software`):
 - используются общие блоки `initBlock` и `runBlock` с произвольными операторами `statement`.

Сигнатура каналов описывается правилами `channelSignature` и `channelClause`:

- допускается перечисление трёх типов портов:
 - `in: in1, in2;`
 - `out: out1, out2;`
 - `self: loop;`
- список имён задаётся правилом `idList` (идентификаторы через запятую).

Операторы в software-узлах

Тело `init/run` программных узлов описывается правилом `block`, включающим последовательность операторов `statement`:

- `varDeclStmt` — объявление переменной:
 - `int x = 0;`
 - `string s = "hi";`
- `assignStmt` — присваивание:
 - `x = expr;`
- `fifoWriteStmt` — запись в FIFO-очередь:
 - `fifoName <- expr;`
- `printStmt` — вывод:
 - `print(expr);`
- `ifStmt` — условный оператор:
 - `if (cond) { ... } else { ... }`

- `exprStmt` — «голое» выражение как оператор:
 - `expr`;
- пустой оператор `;` для удобного форматирования.

Типы переменных ограничены двумя ключевыми словами:

- `int (INT_TYPE)`;
- `string (STRING_TYPE)`.

Операторы в hardware-узлах

Для аппаратных узлов вводится отдельное правило `hardwareStatement`, отражающее более функциональный стиль:

- `hardwareVarDeclStmt`:
 - `x = expr`; — неявное объявление целочисленной переменной;
 - `x = if (cond) expr1 else expr2`; — тернарная форма через `hardwareIfStmt`;
- `hardwareFifoWriteStmt`:
 - `fifo <- expr`;
 - `fifo <- if (cond) expr1 else expr2`; — выбор значения при записи в очередь;
- `printStmt` и `exprStmt` — разрешены для отладки и вычислений, не затрагивающих проводку.

Полный `if` с блоками `{ ... }` в hardware-узлах не используется; вместо него применяются условные выражения, возвращающие значения (аналог тернарного оператора).

Выражения и приоритеты операторов

Правило `expr` задаёт иерархию приоритетов (от низкого к высокому):

1. Логическое `||` (OR);
2. Логическое `&&` (AND);
3. Сравнения `==, !=` (EQ, NEQ);
4. Отношения `<, <=, >, >=` (LT, LE, GT, GE);
5. Сдвиги `<<, >>` (SHIFT_LEFT, SHIFT_RIGHT);
6. Сложение/вычитание `+, -` (PLUS, MINUS);
7. Умножение/деление `*, /` (STAR, SLASH);
8. Унарные операции:
 - логическое отрицание `!expr` (NOT);
 - числовое отрицание `-expr`;
 - чтение из FIFO `$fifo` или `$fifo?` (DOLLAR ID QUESTION?).

Базовые элементы (`primary`) включают:

- целочисленные литералы (`INT_LITERAL`);
- строковые литералы (`STRING_LITERAL`);
- идентификаторы (`ID`);
- скобочные выражения (`expr`).

Таким образом, язык поддерживает привычный набор арифметических, логических и сравнительных операций, а также специальные операции работы с FIFO-очередями.

Описание сети network

Глобальная сеть Кана задаётся отдельной конструкцией:

```
network {
  A.out1 -> B.in0;
  B.out0 -> C.in0;
}
```

В грамматике это выражается правилами:

- `networkDecl` — ключевое слово `network`, за которым следует блок `{ ... }` с перечнем соединений `connection`;
- `connection` — оператор вида `endpoint ARROW endpoint`; где `ARROW` — `->`;
- `endpoint` — пара ID `'.'` ID, соответствующая `ИмяУзла.ИмяПорта`.

Это правило ограничивает язык одной глобальной сетью без имени, но допускает произвольное количество соединений между узлами.

Лексические элементы

Лексер грамматики Solace определяет:

- ключевые слова (`NODE`, `HARDWARE`, `SOFTWARE`, `INIT`, `RUN`, `NETWORK`, `IN`, `OUT`, `SELF`, `INT_TYPE`, `STRING_TYPE`, `PRINT`, `IF`, `ELSE`);
- операторы и знаки (`+`, `-`, `*`, `/`, `!`, `&&`, `||`, `<<`, `>>`, `<-`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `=`, `->`, скобки, точки с запятой, запятые, двоеточие, точка);
- специальные символы для FIFO (`$`, `?`);
- литералы:
 - `INT_LITERAL` — последовательность цифр;
 - `STRING_LITERAL` — строка в двойных кавычках с поддержкой экранирования;
- идентификаторы (ID) — буква/подчёркивание, далее буквы/цифры/подчёркивания.

Пробелы и переводы строк (WS) игнорируются; поддерживаются построчные (`// ...`) и блочные (`/* ... */`) комментарии.

В совокупности эта грамматика задаёт компактный, но выразительный язык, пригодный для описания узлов, их портов и логики работы, а также для спецификации топологии сети Кана.

Компилятор (compiler)

Реализация компилятора включает следующие ключевые элементы:

- использование ANTLR-генерируемых классов `SolaceLexer` и `SolaceParser` для получения AST;
- модуль анализа сети (`analyzeProgram`), который:
 - извлекает из AST список узлов, их типы и сигнатуры портов;
 - строит модель соединений;
 - выполняет набор проверок корректности (уникальность имён, отсутствие конфликтов портов и соединений, использование портов);
 - при обнаружении ошибок выбрасывает `ValidationException` с информативными сообщениями;
- генераторы байткода:
 - `HardwareVisitor` — обходит аппаратные узлы, формируя последовательности инструкций для симулятора; инструкции кодируются функцией `AsmParser.encodeInstructions` в строковое представление и далее в байты;
 - `SoftwareVisitor` — аналогично обходит программные узлы, генерируя инструкции для стековой машины `Harv`;
- упаковщик пакетного файла:
 - строит таблицу строк (имена узлов и портов, используемых в топологии);
 - формирует поток инструкций `NODE_DEF/CONNECT`, вычисляя смещения и размеры контейнеров `SOLB`;
 - записывает заголовок `SOLP`, таблицу строк, поток инструкций и далее блоки байткода;
 - проверяет корректность размеров и типов узлов при упаковке.

Отдельно специфицирован формат выходных файлов (`FILE_FORMATS.md`), фиксирующий бинарный протокол взаимодействия компилятора и BM.

Формат выходных файлов компилятора

Документ FILE_FORMATS.md подробно описывает два ключевых формата:

1. Пакет SOLP (*.solpkg):

- заголовок фиксированной длины (16 байт, little-endian) со следующими полями:
 - magic ("SOLP") — сигнатура файла;
 - container_version (0x01) — версия формата пакета;
 - flags и reserved — зарезервированные поля;
 - meta_size — длина мета-секции (таблица строк + поток инструкций);
 - node_count — количество узлов в пакете (дублирует информацию в мета-секции);
- мета-секция, состоящая из:
 - таблицы строк:
 - * u32 string_count;
 - * далее string_count строк, каждая в формате u16 byte_length + byte[byte_length] (UTF-8 без нуль-терминатора);
 - * строки индексируются от 0 и используются для кодирования имён узлов и портов;
 - потока инструкций:
 - * инструкции идут подряд до опкода END (0xFF);
 - * все идентификаторы узлов и портов представлены индексами строк из таблицы;
 - * каждая инструкция кодирует либо объявление узла (NODE_DEF), либо соединение (CONNECT);
- секция байткода, содержащая контейнеры SOLB для каждого узла.

2. Контейнер байткода SOLB:

- заголовок с полями magic, версия, тип узла, версия ISA, флаги, размеры секций init и run;
- последовательность байтов init, за которой следует последовательность байтов run.

Отдельно спецификация описывает формат инструкций мета-секции:

- NODE_DEF (0x01) — объявление узла и привязка к блоку байткода:
 - содержит идентификатор имени узла (node_name_id), тип узла (0 = hardware, 1 = software), количество и идентификаторы имён входов (in_*), выходов (out_*) и self-портов (self_*);
 - включает абсолютные смещения и размеры контейнера SOLB (bc_offset, bc_size) и поле формата байткода (bc_format, в текущей версии всегда 1 = solbc);
 - требует уникальности имён узлов в пакете.
- CONNECT (0x02) — соединение двух портов:
 - хранит четыре идентификатора строк: from_node, from_port, to_node, to_port;
 - по этим идентификаторам сетевая ВМ восстанавливает ориентированное ребро сети Кана.
- END (0xFF) — завершение потока инструкций.

В FILE_FORMATS.md приведён пример дампа небольшого пакета, в котором:

- таблица строк содержит имена узлов ("Sensor", "Controller"), имена портов ("data", "cmd") и служебные строки;
- две инструкции NODE_DEF объявляют hardware-узел Sensor и software-узел Controller, связывая их с соответствующими блоками байткода;
- одна инструкция CONNECT описывает соединение Sensor.data -> Controller.data.

Компилятор строго следует этой спецификации, вычисляя смещения и размеры блоков и проверяя, что фактическая длина записанных инструкций совпадает с расчётной. Это позволяет сетевой ВМ однозначно интерпретировать содержимое файла и надёжно извлекать байткод узлов, а также открывает возможность внешним инструментам анализировать и визуализировать собранные пакеты Solace без знания внутренней реализации компилятора и ВМ.

Сетевая ВМ (vm/network)

Основные компоненты реализации:

- загрузчик пакета:
 - проверяет заголовок SOLP, версии и размеры;
 - читает таблицу строк и поток инструкций, восстанавливая описание узлов и соединений;
 - для каждого узла определяет смещение и размер соответствующего контейнера SOLB.
- строитель сети (buildNetwork):
 - по описанию узлов создаёт структуру NetworkNode с портами и каналами;
 - для каждого соединения создаёт канал FIFO, а для каждого self-порта — отдельный канал с разветвлением на вход/выход;
 - при активированном sniff-режиме вставляет промежуточные каналы wire/deliver и корутины-наблюдатели, пишущие трафик в лог или CSV.
- фабрики узловых ВМ:
 - StubNodeVmFactory создаёт заглушки, проверяющие корректность подключения портов и периодически сообщаящие о «живости» узла;
 - SimNodeVmFactory парсит контейнеры SOLB для аппаратных узлов, создаёт симуляторы и управляет их запуском, обмениваясь данными через FIFO;
 - HarvNodeVmFactory и MixedNodeVmFactory связывают сетевую ВМ со стековой машиной Harv и обеспечивают смешанный режим исполнения.
- управляющая логика (runNetwork):
 - создаёт корутины для всех узлов, запускает их и по необходимости завершает работу сети по таймауту;
 - корректно останавливает sniffer-задачи и закрывает ресурсы (файлы CSV и т. п.).

Sniff-режим и CSV-логирование

Отдельного внимания заслуживает реализация наблюдения за трафиком в сети:

- при отключённом sniff-режиме:
 - для каждого соединения создаётся один буферизованный канал, используемый и отправителем, и получателем;
- при включённом sniff-режиме:
 - создаются два канала: wire (канал «провода») и deliver (канал доставки);
 - в отдельной корутине запускается sniffer, который:
 - * считывает значения из wire;
 - * при активной настройке sniff-режима:
 - печатает сообщения в текстовом формате [sniff] A.out -> B.in: value;
 - или записывает строки в CSV-формате from_node, from_port, to_node, to_port, value;
 - соблюдает ограничение по числу записей, отключая вывод при достижении лимита;
 - * пересылает каждое значение далее в канал deliver.

Такой подход позволяет прозрачно «подключить осциллограф» к любому соединению в сети, не изменяя логику узловых ВМ и не вмешиваясь в формат пакетного файла.

Симулятор (vm/sim)

Реализация симулятора аппаратных узлов основана на классе NetlistGraph, представляющем вычислительную схему как граф элементов и соединений. Поддерживаются:

- базовые элементы (арифметические, логические, сдвиги, сравнения);
- специальные элементы Register, Mux2, Fifo для организации состояния и ветвления потоков данных;
- FIFO-очереди, используемые для взаимодействия с другими ВМ и для реализации self-портов.

Выполнение симулятора заключается в пошаговом распространении значений по графу с учётом зависимостей и наличия/отсутствия данных во входных FIFO. При нехватке данных симулятор возвращает статус BLOCKED, при успешном завершении шага — SUCCESS, при ошибках — ERROR.

Особенность модели комбинационной логики в том, что большую часть схемы можно рассматривать как чистую функцию от входов к выходам. Элементы Register и self-FIFO вносят во взаимодействие «память», позволяя строить последовательные схемы (счётчики, генераторы импульсов, фильтры), которые эволюционируют от такта к такту. Компилятор для hardware-узлов генерирует для симулятора набор инструкций, которые фактически описывают, как из исходного кода Solace построить и обновлять такой граф.

Стековая ВМ Harv (vm/harv)

Основной класс StackMachine инкапсулирует:

- стек значений;
- таблицу переменных и их типов;
- таблицу меток;
- список инструкций и счётчик команд.

Двухфазное исполнение реализовано методами tryInit (инициализационная фаза) и tryRun (основная фаза). Парсер AsmParser обеспечивает:

- преобразование текстового представления инструкций в закодированный байткод;
- обратное декодирование байткода в список инструкций;
- разбор инструкций в объекты с общим интерфейсом Instruction.

Инструкции покрывают арифметику, сравнения, логику, работу со стеком, управление выполнением, определение переменных, работу с FIFO (pushsize, операции чтения/записи) и вывод (print). FIFO реализованы как отдельный тип HarvFifo с блокирующим поведением: при попытке чтения из пустой очереди машина возвращает статус BLOCKED.

Текстовый синтаксис инструкций Harv унифицирован и использует префиксы:

- . — начало инструкции (.define, .push, .add и т. д.);
- % — указание типа значения (%int, %string, %fifo);
- \$ — имя переменной или FIFO-очереди;
- # — целочисленный литерал;
- + — строковый литерал;
- ? — пометка инструкций, относящихся к инициализационной фазе.

AsmParser поддерживает несколько режимов:

- кодирование списка инструкций из текстового вида в компактный бинарный вид;
- обратную декодировку бинарного байткода в список инструкций (для отладки и тестирования);
- парсинг текста непосредственно в объекты Instruction, которые затем исполняются StackMachine.

Такая архитектура парсера позволяет экспериментировать с форматами хранения программ (текст/бинарный) без изменения логики самой виртуальной машины.

Описание тестов

Тестирование системы Solace организовано на нескольких уровнях.

Тесты компилятора

Тесты модуля `compiler` расположены в `compiler/src/test/kotlin/solace/compiler` и покрывают несколько уровней: парсер, анализ сети, генерацию байткода и интеграцию с сетевой ВМ.

ParserConstructsTest.kt

Этот класс проверяет, что ANTLR-грамматика и парсер корректно разбирают основные конструкции языка:

- `parsesExpressionsAndControlFlow`:
 - создаёт программный узел `Calc` с нетривиальным выражением `1 + 2 * 3 << 1`, унарным минусом и логическими операциями `&&`, `||`;
 - содержит `if (x > 0 && y != 3 || x == y) { ... } else { ... }`;
 - после парсинга проверяет, что в дереве присутствуют узлы типов `IfStmtContext`, `ShiftLeftExprContext`, `OrExprContext`.
- `parsesHardwareIfStatements`:
 - описывает `hardware`-узел `I0` с тернарным выражением `if ($inp == 0) 25 else (x >> 2) * 3` при записи в `FIFO`;
 - убеждается, что парсер создаёт `HardwareFifoWriteStmtContext` и `HardwareIfStmtContext`.
- `parsesFifoReadWriteAndSelfOptional`:
 - моделирует чтение из обычного входа `v = $inp`; и опциональное чтение из `self`-порта `opt = $selfp?;`, а также запись `selfp <- opt;`;
 - проверяет наличие узлов `HardwareFifoWriteStmtContext` и `FifoReadExprContext` в `AST`.
- `parsesNetworkDeclaration`:
 - собирает минимальный пример с двумя узлами и секцией `network { A.o -> B.i; }`;
 - проверяет, что разобраны `NetworkDeclContext` и `ConnectionContext`.
- `parsesMinimalNode`:
 - тестирует самый простой `hardware`-узел без портов с пустыми блоками `init` и `run`;
 - проверяет наличие `NodeDeclContext` в дереве.

Таким образом, `ParserConstructsTest` фиксирует, что грамматика поддерживает ключевые конструкции языка (арифметика, логика, `if/else`, `FIFO`-операции, `network`) и не допускает синтаксических ошибок на типичных примерах.

NetworkAnalysisTest.kt

Этот класс фокусируется на семантическом анализе сети (функция `analyzeProgram`):

- `collectsNodesAndConnections`:
 - описывает два узла `A` и `B` (`hardware/software`) и одно соединение `A.aOut -> B.bIn`;
 - проверяет, что анализатор:
 - * собирает два узла в `topology.nodes`;
 - * корректно восстанавливает типы узлов и их порты;
 - * создаёт ровно одно соединение с правильными именами узлов и портов.
- `failsOnUnknownNode`:
 - содержит соединение с несуществующими узлами/портами `A.x -> B.y`;
 - ожидает выброс `ValidationException`, проверяя обработку ссылок на неизвестные сущности.
- `failsOnDuplicateConnections`:
 - задаёт два одинаковых соединения `A.x -> B.y`;
 - проверяет, что анализатор запрещает дублирование рёбер в графе сети.
- `failsOnUnusedPorts`:

- объявляет у hardware-узла выходы `used` и `unused`, но подключает только `used`;
- ожидает ошибку из-за неиспользованного порта `unused`.
- `selfUsedInCodeIsNotReportedUnused`:
 - описывает self-порт `loop`, который используется только в коде узла (`x = $loop?; loop <- x;`);
 - проверяет, что такой self-порт не считается неиспользованным, несмотря на отсутствие внешних соединений.
- `failsOnUnusedSelfPort`:
 - объявляет два self-порта `used` и `unused`, при этом в коде используется только `used`;
 - ожидает `ValidationException` из-за неиспользованного self-порта.
- `failsOnDuplicatePorts`:
 - пытается объявить входы `in: x, x;`;
 - проверяет, что анализатор запрещает дублирующиеся имена портов в сигнатуре.

В итоге `NetworkAnalysisTest` гарантирует, что граф сети строится корректно, а все основные ошибки топологии (неизвестные узлы, дубликаты, неиспользованные порты) выявляются на этапе компиляции.

HardwareVisitorTest.kt

Тесты этого класса проверяют генерацию аппаратного байткода и его исполнение симулятором:

- `hardwareVisitorBasicTest`:
 - определяет hardware-узел `Counter` с self-очередью `loop` и выходом `numbers`, увеличивающий значение на каждом шаге;
 - пропускает AST через `HardwareVisitor`, собирая структуру `HardwareVisitor.Node`;
 - кодирует инструкции `init/run` через `AsmParser.encodeInstructions` и загружает их в `Simulator`;
 - проверяет, что:
 - * после `tryInit` в self-очереди `loop` находится одно значение 0;
 - * последовательные вызовы `tryRun` дают статусы `SUCCESS` и накапливают в FIFO `numbers` значения 1, 2, 3, 4.
- `hardwareVisitorNegativeNumbersTest`:
 - описывает узел `AntiCounter`, который считает вниз от 10 до 0, затем сбрасывает состояние обратно в 10;
 - после генерации байткода и выполнения в `Simulator` проверяет:
 - * корректную инициализацию self-очереди `loop` значением 10;
 - * накопление в выходной очереди `numbers` последовательности 9, 8, ..., 0, 9;
 - * соответствие размеров FIFO ожидаемым значениям после каждого шага.
- `hardwareVisitorTestAdder`:
 - тестирует простейший сумматор `Adder`, который читает два значения из входного FIFO `inp` и пишет их сумму в `outp`;
 - моделирует ситуацию, когда симулятор сначала блокируется при недостатке данных (`ExecStatus.BLOCKED`), а затем, после дозаписи во вход, успешно вычисляет сумму и кладёт её в `outp`.
- `hardwareVisitorMultipleNodesTest`:
 - создаёт программу с двумя hardware-узлами (`AntiCounter`, `Adder`) и одним software-узлом `SoftwareNode`;
 - проверяет, что `HardwareVisitor` возвращает список из двух аппаратных узлов в правильном порядке и игнорирует программный узел.

Эти тесты доказывают, что `HardwareVisitor`:

- корректно собирает метаданные об аппаратных узлах (имена, порты, регистры);
- генерирует байткод, который интерпретатор `Simulator` исполняет согласно ожидаемой семантике (счётчики, условные переходы, арифметика, блокировки FIFO).

SoftwareVisitorTest.kt

Класс `SoftwareVisitorTest` отвечает за проверку генерации байткода для программных узлов и его исполнения стековой машиной `Harv`:

- `testAdderNode`:
 - описывает программный узел `Adder` с двумя входами и одним выходом;
 - прогоняет AST через `SoftwareVisitor` и проверяет:
 - * имя узла ("Adder");
 - * корректное распознавание входов и выходов (наличие `in1`, `outp` в соответствующих множествах).
- `testBasicNode`:
 - реализует узел `Counter` с self-очередью `loop`, который инкрементирует значение и записывает результат в `FIFO numbers`;
 - проверяет:
 - * корректность собранных метаданных (`name`, `outs`, `selves`, `declaredVariables`);
 - * генерацию `init/run`-кода, дальнейшее кодирование через `Harv-AsmParser` и успешное исполнение в `StackMachine`;
 - * то, что:
 - после `tryInit` self-очередь `loop` содержит 0;
 - последовательные вызовы `tryRun` формируют в `FIFO numbers` значения 1, 2, 3, 4.
- `testIfElse`:
 - моделирует более сложный узел `AntiCounter` с несколькими `if/else`, строковыми переменными и self-очередями;
 - проверяет:
 - * корректную работу `SoftwareVisitor` с условными блоками, объявлением и использованием строковых переменных;
 - * корректность структуры сгенерированных инструкций (класс хранит и кодирует `init/run`-код, хотя в тесте проверка ограничена метаданными и успешным кодированием).

В сумме эти тесты подтверждают, что фронтенд для программных узлов правильно интерпретирует синтаксис языка `Solace` и генерирует байткод, исполняемый стековой машиной `Harv`.

PackagingIntegrationTest.kt

Интеграционный тест `PackagingIntegrationTest` связывает все части компилятора и сетевую ВМ:

- `compiler emits runnable solpkg for hardware network`:
 - описывает простую hardware-сеть из двух узлов:
 - * `Pulse` — генерирует единичный импульс `tick <- 1`;
 - * `Accumulator` — поддерживает self-состояние `state`, суммирует вход `inc` и накопленное значение, выдаёт результат в `total`;
 - выполняет полный конвейер:
 1. парсинг программы в AST;
 2. генерацию hardware-байткода (`buildHardwareBytecode`);
 3. анализ сети (`analyzeProgram`);
 4. упаковку в пакет `*.solpkg` (`writeProgramPackage`);
 5. загрузку пакета (`loadProgramPackage`) и построение сети (`buildNetwork`);
 6. запуск сети с `SimNodeVmFactory` и чтение трёх значений с выхода `Accumulator.total`;
 - проверяет:
 - * что в загруженном пакете два узла и у каждого есть ненулевой байткод;
 - * что последовательность значений на выходе `total` равна [1, 2, 3].

Этот тест демонстрирует работоспособность всей цепочки «язык → компилятор → формат SOLP → сетевая ВМ → симулятор hardware-узлов».

Тесты симулятора (vm/sim)

Для симулятора аппаратных узлов тестируются:

- корректность построения и расчёта графа NetlistGraph;
- корректность работы парсера инструкций и преобразований между байткодом и объектами Kotlin;
- выполнение примеров программ, сгенерированных компилятором или написанных на ассемблероподобном языке;
- механизмы блокировки при недостатке данных во входных FIFO.

Также тестируется связка «компилятор → симулятор»: программы на языке Solace компилируются в байткод, загружаются в симулятор и сравниваются ожидаемые и фактические результаты вычислений.

Тесты стековой ВМ Harv (vm/harv)

Тесты стековой машины сосредоточены на:

- проверке блокирующего поведения FIFO (чтение из пустой очереди даёт статус BLOCKED, после поступления данных выполнение продолжается);
- корректности реализации счётчиков и циклических конструкций с использованием FIFO для хранения состояния;
- верификации сериализации и обратного парсинга инструкций (тесты round-trip для всех типов инструкций).

Тесты сетевой ВМ (vm/network)

Тесты сетевой ВМ расположены в vm/network/src/test/kotlin/solace/network и проверяют как базовую проводку портов, так и работу фабрик ВМ на реальных сетях, включая пайплайны Фибоначчи.

RuntimeTest

- buildNetwork wires self port to same channel:
 - строит сеть с одним узлом Counter, имеющим self-порт loop;
 - проверяет, что buildNetwork создаёт один общий канал для inputs["loop"], outputs["loop"] и self["loop"], а множества портов содержат и обычные, и self-порты.
- stub factory rejects miswired self channel:
 - вручную конструирует NetworkNode с self-портом, подключённым к отличному от входа/выхода каналу;
 - ожидает, что StubNodeVmFactory бросает IllegalArgumentException при такой некорректной проводке.
- stub factory accepts valid wiring:
 - создаёт корректный NetworkNode, где self-порт и соответствующие вход/выход разделяют один канал;
 - проверяет, что фабрика создаёт ВМ без исключений.
- sniffer forwards traffic between nodes:
 - собирает минимальную программу A.out -> B.in и включает sniff-режим;
 - убеждается, что значение, отправленное из A, корректно доходит до входа B через sniffer-каналы.

SimNodeVmFactoryTest

- sim factory runs hardware network end-to-end:
 - вручную описывает hardware-узлы Pulse и Accumulator, кодируя их solbc-контейнеры через AsmParser;

- * Pulse генерирует тик (`tick <- 1`);
- * Accumulator накапливает сумму входных импульсов в self-состоянии `state` и выдаёт её в `total`;
- строит сеть с соединением `Pulse.tick -> Accumulator.inc`, запускает её через `SimNodeVmFactory`;
- читает три значения с выхода `Accumulator.total` и проверяет, что они равны `[1, 2, 3]`.

FibonacciNetworkTest

Этот класс тестирует hardware-реализацию поточного вычислителя Фибоначчи.

- `fibonacci pipeline matches first 45 numbers`:
 - вручную конструирует три hardware-узла:
 - * `TickSource` — источник тиков, генерирующий единичные импульсы;
 - * `FibStepper` — вычислитель, хранящий состояние Фибоначчи в self-FIFO `prev/curr` и выдающий следующий член последовательности в `fib`;
 - * `FibSink` — приёмник, сохраняющий последнее полученное значение в self-FIFO `last`;
 - описывает соединения `TickSource.tick -> FibStepper.step` и `FibStepper.fib -> FibSink.inFib`;
 - строит сеть с включённым `sniff`-режимом и CSV-логированием, ограничивая число записей;
 - читает `expectedCount + 1` значений с выхода `FibSink.last`, отбрасывает начальное семя и сверяет оставшиеся с эталонной последовательностью Фибоначчи, начиная с `F2`;
 - дополнительно читает CSV-лог и проверяет, что значения, переданные по ребру `FibStepper.fib -> FibSink.inFib`, совпадают с ожидаемыми числами Фибоначчи.

Таким образом, тест `FibonacciNetworkTest` подтверждает, что сетевую ВМ и симулятор можно использовать для построения и верификации потоковых аппаратных вычислителей, а `sniff`-режим корректно отражает реальный трафик в сети.

HarvNodeVmFactoryTest

Этот класс проверяет работу Harv-фабрики на программных узлах.

- `harv factory echoes input to output`:
 - создаёт один software-узел `Echo` с входом `in` и выходом `out`, чья программа на Harv-ассемблере просто пересылает данные из `in` в `out`;
 - строит сеть без соединений, запускает её через `HarvNodeVmFactory`;
 - отправляет в входной канал значения `10` и `20` и проверяет, что на выходе они приходят в том же порядке.
- `harv factory computes fibonacci sequence with software nodes`:
 - описывает три software-узла:
 - * `TickSource` — программный генератор тиков (увеличивает счётчик и отправляет его в FIFO `tick`);
 - * `FibStepper` — программный вычислитель Фибоначчи, поддерживающий переменные `prev` и `curr`, вычисляющий следующее значение и отправляющий его в FIFO `fib`;
 - * `FibSink` — приёмник, сохраняющий последнее значение последовательности в FIFO `last`;
 - строит программу с соединениями `TickSource.tick -> FibStepper.step` и `FibStepper.fib -> FibSink.inFib`;
 - использует обёртку над `HarvNodeVmFactory`, чтобы для узла `TickSource` вместо интерпретации Harv-байткода явно отправлять `expectedCount` тиков;
 - запускает сеть через `runNetwork` с включённым `sniff`-режимом и CSV-логом в файл;

- извлекает из CSV-файла значения, проходящие по ребру `FibStepper.fib` -> `FibSink.inFib`, и сравнивает их с эталонной последовательностью Фибоначчи (также начиная с F2).

Именно во второй части `HarvNodeVmFactoryTest` реализован тест вычисления Фибоначчи на чисто программной сети, что дополняет аппаратный сценарий из `FibonacciNetworkTest`.

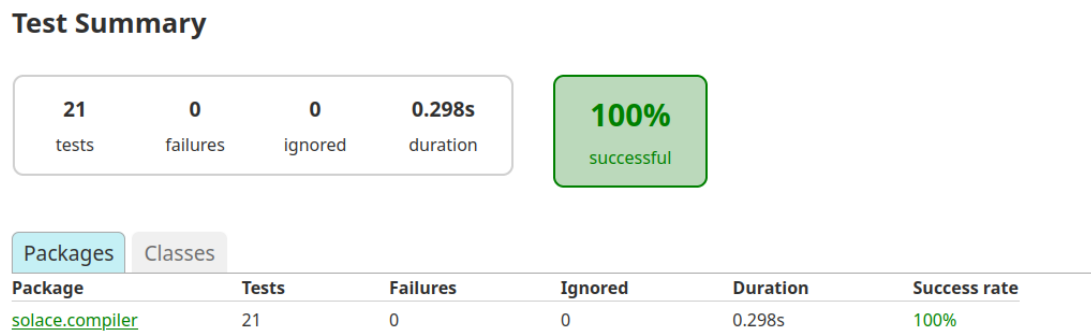
DotOutTest

- `dot-out` writes Graphviz file for loaded program:
 - создаёт искусственный пакет `*.solpkg` с двумя узлами и одним соединением, используя тот же бинарный формат, что и компилятор;
 - запускает `main` сетевой ВМ с флагом `--dot-out`, указывая путь к временным файлам;
 - проверяет, что:
 - * DOT-файл действительно создаётся;
 - * его содержимое совпадает с результатом `buildNetwork(...).toDOTNetwork()`, то есть корректно отражает топологию сети.

В совокупности эти тесты демонстрируют, что сетевая ВМ:

- верно строит и валидирует граф сети Кана по загруженному `*.solpkg` или вручную сконструированной программе;
- корректно работает с `hardware`- и `software`-узлами через соответствующие фабрики ВМ;
- поддерживает наблюдение трафика (`sniff/CSV`) и экспорт топологии в формат Graphviz DOT.

Успешное выполнение тестов приведено на рисунке



Generated by [Gradle 9.1.0](#) at Dec 9, 2025, 1:20:29 PM

Рис. 4: Успешное выполнение тестов

Вывод

В рамках проекта Solace реализована экспериментальная платформа для моделирования вычислений на основе сети Кана, объединяющая язык описания узлов и сети, компилятор и набор виртуальных машин. Предложенная архитектура чётко разделяет ответственность между компонентами: компилятор отвечает за анализ и упаковку программы в единый бинарный формат, узловые ВМ реализуют конкретные модели вычислений (комбинационная логика и стековая машина), а сетевая ВМ строит и исполняет сеть, обеспечивая взаимодействие узлов через FIFO-каналы.

Реализация сопровождается достаточно подробной спецификацией форматов выходных файлов и комплексным набором модульных и интеграционных тестов, проверяющих корректность парсинга, анализа сети, генерации байткода и исполнения программ. Это создаёт основу для дальнейшего развития системы: расширения языка, добавления новых типов узловых ВМ, экспериментирования с альтернативными моделями вычислений и интеграции с реальными встраиваемыми и распределёнными системами.