SystemKotlin: предметно-ориентированный язык (DSL) для моделирования аппаратуры

Воробьёв Сергей Владимирович

Санкт-Петербургский Политехнический Университет

Санкт-Петербург, Россия

seryoga.vor@mail.ru

Соболь Валентин Олегович

Санкт-Петербургский Политехнический Университет

Санкт-Петербург, Россия

TODO: my.address@at

Глухих Михаил Игоревич

Санкт-Петербургский Политехнический Университет

Санкт-Петербург, Россия

mikhail.glukhikh@gmail.com

*Аннотация* — TODO

Ключевые слова — моделирование аппаратуры, Kotlin, DSL, предметно-ориентированный язык, SystemC

# Введение

При проектировании сложных аппаратных систем значительную часть времени разработки занимает моделирование, тестирование и отладка. По этой причине удобство и производительность предназначенных для этого средств играет важную роль в технологической цепочке. Традиционными языками описания аппаратуры являются VHDL [] и Verilog []. Для данных языков существуют мощные средства синтеза, однако удобство средств разработки, тестирования и отладки для этих языков оставляет желать лучшего. В связи с этим уже более десяти лет известны предметно-ориентированные языки (DSL) на базе языков высокого уровня, позволяющие использовать средства разработки и отладки для базовых языков.

На данный момент известно несколько предметно-ориентированных языков, предназначенных для описания и моделирования аппаратуры. Наибольшей популярностью среди них пользуется язык SystemC [], разрабатываемый Accelera System Initiative. В качестве базового языка используется C++. Язык SystemC позволяет создавать описание как на уровне регистровых передач (RTL), так и на уровне транзакций (TLM). Для моделирования внутри SystemC имеется две реализации планировщика: одна на базе библиотеки нитей pthreads [], удобная для отладки, но имеющая крайне низкую производительность, и другая на базе встроенных в язык быстрых нитей (qthreads), имеющая приемлемую производительность, но ориентированная на однопроцессорное исполнение и почти не позволяющая производить отладку. Существует несколько статей, исследующих возможности повышения производительности исполнения SystemC-кода [] [].

Для SystemC существуют автоматизированные преобразователи в язык Verilog, а именно, Cadence C-to-Silicon Compiler [] и Calypto Catapult C []. Их наличие позволяет полноценно использовать SystemC в технологической цепочке разработки аппаратуры (SystemC 🡪 Verilog 🡪 ASIC / FPGA).

Среди недостатков SystemC можно отметить небезопасность базового языка, использование средств препроцессора, отсутствие удобные средств аннотирования. Отсюда вытекает некоторое неудобство имеющихся средств разработки, анализа и отладки программ на SystemC. Также некоторой проблемой является сравнительно медленная производительность SystemC-программ. TODO: здесь нужно какое-нибудь сравнение.

Другими предметно-ориентированными языками описания аппаратуры являются Chisel [] на базе языка Scala и Spin [] на базе языка Java. Spin, по сути дела, является попыткой переноса SystemC на Java и имеет похожую архитектуру. В [] на наборе тестов показывается, что программы на основе Spin и на основе SystemC имеют близкую производительность.

Chisel разрабатывается в UC Berkeley, ориентирован на уровень регистровых передач, позволяет генерировать Verilog-код для синтеза или C++ код для моделирования. Согласно [] производительность моделирования Chisel в 8 раз выше аналогичного Verilog-кода в Synopsys VCS Simulation []. Однако, подобный подход затрудняет отладку, поскольку программисту приходится постоянно переключаться между программой на Chisel и синтезированной из неё программой на C++.

TODO: стартовый абзац про Kotlin и SysKotlin.

# Структура и основные компоненты

## Структура ядра библиотеки

Функции планировщика в библиотеке выполняет класс SysScheduler. Он принимает решение о запуске функций (SysFunction) на основе анализа их списков чувствительностей и списка произошедших на данный момент событий (SysWait). Имеется несколько видов событий:

* Time — событие, которое происходит через определённый промежуток времени, заданный в фемтосекундах (по умолчанию) или в других единицах времени.
* Event — пользовательское событие, происходящее в момент вызова метода happens.
* OneOf — событие-контейнер “одно из”, происходит одновременно с первым случившимся событием внутри контейнера.

DSL поддерживает два вида функций: обычные (SysFunction) и их расширение — функции с состояниями (SysStateFunction). Любые функции запускаются в зависимости от их списка чувствительности, выполняют предписанные действия и завершаются в тот же момент модельного времени с указанием следующего или следующих ожидаемых событий. Функции с состояниями дополнительно включают в себя логику перехода из состояния в состояние после каждого их завершения.

SysModule представляет собой элемент моделируемого проекта. Модули задают структуру проекта: верхний уровень описывается расширением класса SysTopModule, все следующие классом SysModule. В свою очередь, модуль включает в себя каналы (SysSignal, SysFifo) и порты SysPort для подключения внешних сигналов, а каналы реализуют различные интерфейсы (SysInterface). Имеется несколько видов каналов:

* Сигналы: SysSignal (обычный), SysSignalStub (сигнал-заглушка с постоянным значением), SysBitSignal (битовый сигнал), SysClockedSignal (тактовый сигнал)
* Очереди: SysFifo (обычная), SysBitFifo (битовая) и SysAsynchronousFifo (асинхронная) TODO: возможно пояснить подробности

Битовые каналы, расширяющие класс SysEdged, формируют события posEdgeEvent (фронт сигнала) и negEdgeEvent (спад сигнала). Для их подключения к модулям имеются соответствующие порты: SysInput (вход), SysBitInput, SysOutput, SysFifoInput, SysFifoOutput.

Битовый тип реализуется перечислением SysBit. Он имеет всего 4 состояния ONE, ZERO, X (неопределённое значение), Z (высокий импеданс). Над битовыми переменными реализованы операторы “и”, “или”, “не”.

Для численных типов данных реализованы оболочки, которые позволяют контролировать их ширину (количество бит) и определяют для каждого бита три состояния: 1, 0, Х.

* SysInteger — целочисленный тип с шириной от 0 до 64 бит. Поддерживает основные арифметические операции (сложение, вычитание, умножение, деление), битовые операции (“и”,”или”,”не”) и битовые сдвиги.
* SysUnsigned — целочисленный беззнаковый тип с шириной от 0 до 64 бит. Возможности аналогичны SysInteger.
* SysBigInteger — целочисленный тип с неограниченной шириной. Поддерживаемые операции аналогичны SysInteger.
* SysFloat — число с плавающей точкой (IEEE 754). Поддерживает основные арифметические операции.

Все типы данных, которые могут быть переданы по каналам связи, расширяют класс SysData, что позволяет переменным этого типа иметь неопределённое состояние и ограничивает возможные варианты передаваемых данных.

Для хранения данных реализован готовый модуль-регистр SysRegister и ряд триггеров: DFF, JKFF, RSFF, TFF.

## Использование библиотеки

Основа проекта на SysKotlin — это модуль. Создание проекта стоит начинать с создания модуля верхнего уровня (SysTopModule). Это нужно для упрощения синхронизации внутренних элементов. При создании модуля верхнего уровня автоматически создаётся планировщик, который будет заниматься симуляцией процессов в этой системе. Функция start() этого модуля запускает процесс моделирования. У этого метода есть аргумент, который задаёт время до остановки. Свойство currentTime содержит время, прошедшее с начала запуска. В SysModule есть ряд методов для создания портов, каналов и функций данного модуля. При создании элементы автоматически регистрируются в текущем модуле и планировщике.

Методы input, bitInput, output, fifoOutput, fifoInput создают соответствующие порты. Метод signalStub создаёт сигнал с постоянным состоянием (заглушку на порт). У портов может присутствовать значение по умолчанию, действующее при отсутствии подключения канала к данному порту. Также в модуле присутствуют методы создания каналов: signal, bitSignal, clockedSignal, clockedSignal, fifo, asynchronousFifo. Для того, чтобы присоединить канал к порту, необходимо воспользоваться методом bind у порта или использовать один из методов модуля.

Создание новых типов данных в библиотеке требует расширения интерфейса SysData. Каждый тип данных обязан иметь неопределённое состояние, задаваемое либо его конструктором по умолчанию, либо свойством undefined его объекта-спутника (companion object).

Для того, чтобы создать обычную функцию нужно воспользоваться функцией function, у которой есть три аргумента sensitivities (событие, которое вызывает эту функцию), initialize (если равен true, то у функции присутствует стадия инициализации), run (тело функции, определяющее конкретные выполняемые ей действия). Вызвать этот метод можно несколькими способами, наиболее удобный из них использует синтаксис функций-литералов в языке Kotlin:

*function (sensitivities, initialize) {*

*//Body*

*}*

Вместо списка чувствительности аргументом функции может быть тактовый сигнал clock (порт или канал, реализующий интерфейс SysEdged) с признаком positive (по какому фронту тактового сигнала будет запускаться эта функция).

Для того, чтобы создать функцию с состояниями нужно воспользоваться функцией stateFunction с похожим набором аргументов. Однако, тело такой функции содержит набор состояний, которые будут запускаться в определённом порядке:

* state — элементарное состояние, которое запускается один раз
* block — блок-состояний, которые будут запускаться в определённом порядке
* infinite — состояние, которое не сменяется на другое
* infiniteBlock — аналогичен infinite с блоком состояний внутри, сменяющихся по кольцу
* jump — состояние, переносящее исполнение на соответствующую метку внутри того же блока состояний
* label — задаёт метку для jump
* sleep — пропускает заданное количество запусков
* If — является аналогом block, запускается если переданная в аргументах лямбда-функция возвращает true.
* Else — является аналогом block и продолжением If, запускается если переданная в аргументах If лямбда-функция возвращает false.

Пример создания функции с состояниями (TODO: пояснить пример, возможно добавить рисунок с конечным автоматом):

*stateFunction(clk, false)*

*{*

*var switch = false*

*label("start")*

*state { println("standard: before block") }*

*block {*

*state {*

*switch = !switch*

*println("standard: before IF-Else")*

*}*

*If ({ switch }) {*

*state { println("standard: If-1") }*

*state { println("standard: If-2") }*

*state { println("standard: If-3") }*

*}*

*Else {*

*state { println("standard: Else-1") }*

*state { println("standard: Else-2") }*

*state { println("standard: Else-3") }*

*}*

*state { println("standard: after IF-Else") }*

*}*

*state { println("standard: after block") }*

*jump("start")*

*}*

# Сравнение с SystemC

Основные типы данных, поддерживаемые SystemC, реализованы в SysKotlin с одним отличием: все типы поддерживают неопределённое состояние, что позволяет точнее задавать состояние системы после сброса питания. Так же, как и в SystemC при написании кода можно использовать базовые типы языка. TODO: расширить общее сравнение.

Для оценки производительности были разработаны следующие два примера:

1. Producer/Consumer. Пара модулей, один из которых записывает данные (случайные символы) в очередь, а другой читает данные из очереди. TODO: здесь стоит добавить рисунок.
2. RSA. Реализация алгоритма шифрования RSA на основе SysBigInteger. Генерируются ключи и создаётся сообщение, которое зашифровывается, а затем расшифровывается. Пример предназначен для тестирования численных типов данных.

Для Producer/Consumer размер fifo был установлен на 100 элементов. Количество повторений при тестировании 100000000. Среднее время выполнения составило 7с 777мс. Время выполнения аналогичной реализации на SystemC составило 6с 550мс.

Для оценки скорости выполнения RSA была выбрана длина ключа 250 бит. Время работы составило 180мс. Для аналогичной реализации на SystemC время выполнения 150мс.

По результатам экспериментов можно сделать вывод, что на данных примерах SysKotlin оказывается на 15-20 процентов медленнее.

##### ЗАКЛЮЧЕНИЕ

Заключение, заключение

##### СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. SystemC
2. pthreads
3. Cadence CtoS
4. Calypto Catapult C
5. Chisel
6. Spin
7. Chisel Introduction Article
8. Synopsys VCS Simulation