SysKotlin: предметно-ориентированный язык 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
napster_1997@mail.ru

Глухих Михаил Игоревич
Санкт-Петербургский Политехнический Университет
Санкт-Петербург, Россия
mikhail.glukhikh@gmail.com

***Аннотация* — Рассматривается предметно-ориентированный язык SysKotlin, предназначенный для описания и моделирования аппаратуры и использующий Kotlin в качестве базового языка. Рассмотрены основные примитивы SysKotlin: функции, модули, сигналы, порты, интерфейсы, численные типы. Рассмотрен механизм функций с состояниями, предназначенный для описания сложной управляющей логики. Произведено краткое сравнение возможностей SysKotlin и популярного языка описания аппаратуры SystemC. С точки зрения синтаксиса языки имеют сходный набор возможностей. SysKotlin уступает около 20% в производительности моделирования, выигрывая при этом в удобстве имеющихся средств разработки и отладки.**

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

I. 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 [1], 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 [2], 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 ряд статей, исследующих возможности повышения производительности исполнения SystemC-кода, в качестве примеров можно привести [3] [4].

Для SystemC существуют автоматизированные преобразователи в язык Verilog, а именно, Cadence C-to-Silicon Compiler [5] и Calypto Catapult C [6]. 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.

Другими предметно-ориентированными языками описания аппаратуры являются Chisel [7] на базе языка Scala и Spin [8] на базе языка Java. Spin, по сути дела, является попыткой переноса SystemC на Java и имеет похожую архитектуру. В [9] 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 [9] производительность моделирования Chisel в 8 раз выше аналогичного Verilog-кода в Synopsys VCS Simulation [10]. Однако, подобный подход затрудняет отладку, поскольку программисту приходится постоянно

переключаться между программой на Chisel и синтезированной из неё программой на C++.

Kotlin [11] — новый язык для платформ JVM и Android со статической типизацией. Версия 1.0 вышла в феврале 2016 года, однако язык уже успел набрать определённую популярность, по данным [12] репозитории GitHub уже содержат более двух миллионов строк кода на Kotlin. Среди достоинств данного языка, важных с точки зрения данной статьи, следует отметить его лаконичность, безопасность (в частности, встроенная защита от NullPointerException), наличие поддержки DSL [13].

В данной статье рассматривается предметно-ориентированный язык SysKotlin [14], предназначенный для описания и моделирования аппаратуры и использующий Kotlin в качестве базового языка. SysKotlin в настоящий момент находится в стадии разработки. Создание SysKotlin ориентировано, в первую очередь, на следующие цели:

- Получить DSL, более удобный в разработке и отладке по сравнению с SystemC
- Добиться производительности моделирования, по меньшей мере сопоставимой с производительностью SystemC

II. Структура и основные компоненты

A. *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 (асинхронная), SysBitFifo (асинхронная битовая), SysSynchronousFifo (синхронная), SysSynchronousBitFifo (синхронная битовая)

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

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, X.

- 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.

B. *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(), в этом модуле, 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:

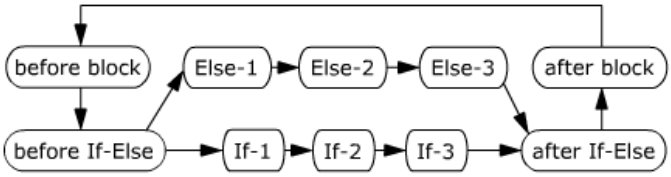
```
function (sensitivities, initialize) {
        //Body
}
```

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 используются ею в определённом порядке:

- state — элементарное состояние, в котором функция находится в течение одного такта
- infinite — loopback-состояние, из которого отсутствуют переходы в другие состояния
- infiniteBlock — блок состояний, перебирающихся поочередно с возвращением от последнего состояния к первому после завершения перебора
- forEach – блок состояний, перебирающихся поочередно ограниченное количество раз, в соответствии с указанным интервалом-аргументом, например forEach(0..9) выполняется 10 раз
- jump — псевдосостояние, переносящее исполнение на соответствующую метку внутри того же блока состояний
- label — не является состоянием, но задаёт метку для jump
- sleep — спящее состояние, предназначенное для пропуска определённого количества тактов
- If — условный блок, запускается, если переданная в аргументах лямбда-функция имеет результат true
- Else — продолжение условного блока If, запускается, если переданная в аргументах If лямбда-функция имеет результат false
- While — блок состояний, который перебирает поочерёдно состояния, пока переданная в аргументах лямбда-функция имеет результат true

Пример создания функции с состояниями:




```
stateFunction(clk, false)
{
    var switch = false
    label("start")
    state { println("before block") }
    state {
        switch = !switch
        println("before IF-Else")
    }
    If ({ switch }) {
        state { println("If-1") }
        state { println("If-2") }
        state { println("If-3") }
    }
    Else {
        state { println("Else-1") }
        state { println("Else-2") }
        state { println("Else-3") }
    }
    state { println("after IF-Else") }
    state { println("after block") }
    jump("start")
}
```

III. 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.

Одним из преимуществ, обеспечиваемых базовым языком Kotlin, является поддержка функций высшего порядка, во многих случаях, позволяющих заменить макросы в SystemC и при этом создающих гораздо меньше проблем с пониманием и отладкой кода.

В SystemC имеются три вида процессов, позволяющих описывать логику разрабатываемых устройств: методы (methods, примерный аналог функций SysKotlin) и нити, простые (threads) и с тактовым сигналом (clocked threads), непосредственного аналога в SysKotlin не имеющие. Заменой SystemC-нитей в SysKotlin служат функции с состояниями, позволяющие, в частности, описывать переходы из состояния в состояния, циклические последовательности состояний, пропускать такты по аналогии с методом wait(n) в SystemC и так далее.

Для оценки производительности были разработаны следующие два примера, аналогичные имеющимся в дистрибутиве SystemC:

1. Producer/Consumer. Пара модулей, один из которых записывает данные (случайные символы) в очередь, а другой читает данные из очереди.
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

Учитывая результаты имеющихся тестов производительности и проведённое сравнение возможностей SystemC и SysKotlin на данный момент, поставленные во введении цели кажутся нам достижимыми.

Задачи на ближайшее будущее:

- Дальнейшее упрощение процесса разработки аппаратуры и наращивание базы примитивов
- Ускорение процесса моделирования, используя другие способы распараллеливания процессов, протекающих в планировщике
- Автоматизированное преобразование кода на SysKotlin в Verilog

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Accelera Systems initiative, SystemC standard page: http://www.accellera.org/downloads/standards/systemc.

[2] POSIX threads: https://en.wikipedia.org/wiki/POSIX_Threads.

[3] Ezudheen, P. and Chandran, P. and Chandra, J. Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines. PADS 2009, 22-25 June 2009, Lake Placid, New York, USA, pp. 80 – 87.

[4] Lu, K. and Mueller-Gritschneder, D. and Schlichtmann, U. Removal of unnecessary context switches from the systemc simulation kernel for fast VP simulation. SAMOS 2011, 18-21 July 2011, Samos, Greece, pp. 150 – 156.

[5] Cadence C-to-Silicon Compiler: http://www.cadence.com/products/sd/silicon_compiler/pages/default.aspx.

[6] Calypto Catapult: http://calypto.com/en/products/catapult/overview/.

[7] Chisel site: https://chisel.eecs.berkeley.edu/.

[8] Spin Experimental Project: http://toem.de/index.php/projects/spin.

[9] Bachrach, Jonathan and Vo, Huy and Richards, Brian. Chisel: Constructing Hardware in a Scala Embedded Language // DAC 2012, June 3-7, 2012, San Francisco, California, USA, pp. 1212 – 1221.

[10] Synopsys VCS Simulation: http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/VCS.aspx.

[11] Официальный сайт Kotlin: http://kotlinlang.org.

[12] Kotlin 1.0 Released: https://blog.jetbrains.com/kotlin/2016/02/kotlin-1-0-released-pragmatic-language-for-jvm-and-android.

[13] Type Safe Builders in Kotlin: https://kotlinlang.org/docs/reference/type-safe-builders.html.

[14] Проект SysKotlin: https://github.com/mglukhikh/SysKotlin.