Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

ПРЕДСТАВЛЕНИЕ МОДЕЛЕЙ ДИНАМИЧЕСКИХ СИСТЕМ КАК ПОТОКА ДАННЫХ

Автор Шалин Никита Андреевич _____
(Фамилия, Имя, Отчество) (Подпись)

Направление подготовки (специальность) *09.03.04 Программная инженерия*

Квалификация *бакалавр*
(бакалавр, инженер, магистр)

Руководитель Пенской А.В., к.т.н. _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

К защите допустить

Зав. кафедрой Муромцев Д.И., доцент, к.т.н. _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

«___» _____ 20 18 г.

Санкт-Петербург, 20 18 г.

Студент Шалин Никита Андреевич **Группа** *P3418* **Кафедра** *ИПМ* **Факультет** *ПИиКТ*

Направленность (профиль), специализация *Разработка программно-информационных систем*

Консультант (ы):

а) _____ _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

б) _____ _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

ВКР принята “ ____ “ _____ 20 _____ г.

Оригинальность ВКР %

ВКР выполнена с оценкой _____

Дата защиты “ ___ “ _____ 20 __ г.

Секретарь ГЭК _____ _____
ФИО Подпись

Листов хранения _____

Демонстрационных материалов / Чертежей хранения _____

Содержание

Введение

Большая часть ПО для расчета моделей системной динамики позволяет производить симуляции только на процессорах общего назначения [1]. Использование специализированных под задачу вычислительных архитектур, потенциально позволяет:

- значительно повысить скорость моделирования за счёт параллелизма и использования специализированных вычислительных узлов;
- снизить потребление электроэнергии на цикл моделирования за счёт минимизации накладных расходов.

Кроме того, метод моделирования динамических систем в реальном времени позволяет применять их в задачах проектирования, макетирования и проведения испытаний информационно-управляющих систем и киберфизических систем для моделирования операционного окружения.

В рамках данной работы чтобы автоматически конфигурировать ПЛИС для расчёта моделей системной динамики с помощью специального ПО, было решено сначала представить модели в виде синхронного потока данных (synchronous dataflow). Для этого надо выполнить задачу разбора описания модели, построения последовательности операторов, вычисляющих данную модель и кодирования описания этого потока данных, приемлемого для дальнейшего использования.

Исходным языком описания моделей системной динамики выбран XMILE. Так как этот язык структурирован (формат XML) и является открытым кроссплатформенным стандартом [2].

1. Системная динамика

Системная динамика - это способ представления и моделирования поведения комплексных динамических систем во времени [3]. Любые динамические системы характеризуются взаимозависимостями, общим взаимодействием, обратной связью и циклической причинной связью компонентов. В основе системной динамики лежит следующий принцип: структура любой системы, множество циклических, взаимоблокирующих или зависящих от времени связей между её компонентами, так же важна для определения её поведения, как и сами компоненты [4].

Как подход к компьютерному моделированию системная динамика была разработана в 50-х годах, чтобы помочь аналитикам в понимании и структурировании промышленных процессов и возникающих в них проблем. Данная область получила свое начало из книги Джея Форрестера (Jay W. Forrester) “Industrial dynamic” (1961). В течение 10 лет после публикации спектр решаемых проблем расширился от проблем корпораций и промышленности до проблем более общих и глобальных, таких как увеличение популяций видов в конечном пространстве. Сейчас используется в таких областях как экономика, политика, защита и изучение окружающей среды, исследование теорий в социальных науках и других, помимо первоначальной управленческой [4].

1.1 Применение системной динамики

Задача моделей системной динамики - улучшить понимание того, как связана эффективность организации с её внутренней структурой и рабочими процессами, и затем использовать это понимание для проектирования более эффективных и успешных процессов. Моделирование позволяет исследовать долгосрочные эффекты каких-либо решений и быстрее изучать протекающие процессы за счёт возможности ускорять или замедлять время симуляции.

Модели системной динамики используются, чтобы отображать всё: от низкоуровневых переменных, таких как имущество, прибыль и выручка до высокоуровневых как доверие бренду и лояльность клиентов. Именно возможность показать, смоделировать и проанализировать взаимодействие и влияние высокоуровневых компонентов системы, бизнеса или экономики выделяет системную динамику среди других аналитических подходов для помощи в принятии решений. Несмотря на это, системная динамика не широко распространена.

1.2 Принципы построения моделей

В моделях системной динамики поведение описывается с помощью накопителей и потоков. Накопители являются сущностями с собственными состояниями, а потоки представляют собой действия или процесса протекающие во времени [5]. То есть именно потоки создают динамику в модели. После того, как определено какие части рассматриваемой проблемы будут накопителями, какие процессы будут выражены через потоки, и построена связь между этими элементами, мы получаем модель системной

динамики пригодную для изучения и решения рассматриваемой проблемы и понимания, как эта система будет вести себя в будущем. Разработчик определяет потоки, чтобы задать правила, по которым протекают процессы в симуляции. Модель системной динамики интерпретирует эти потоки в совокупности с уровнями, создавая дифференциальные уравнения, используемые для вычисления новых значений модели на каждом шаге симуляции.

Стоит заметить, что системы, которые могут быть выражены таким способом, могут быть физическими (например распространение инфекции, см. Рисунок 1) или нефизическими (например симуляция поведения рынка, удовлетворенность клиентов). В экономической и производственных средах существует множество нефизических проблем требующих анализа для того, чтобы максимизировать финансовую эффективность предприятия. Например прогнозирование эффекта от изменения занимаемой доли рынка на количество продаж или влияние на оборот продукции.

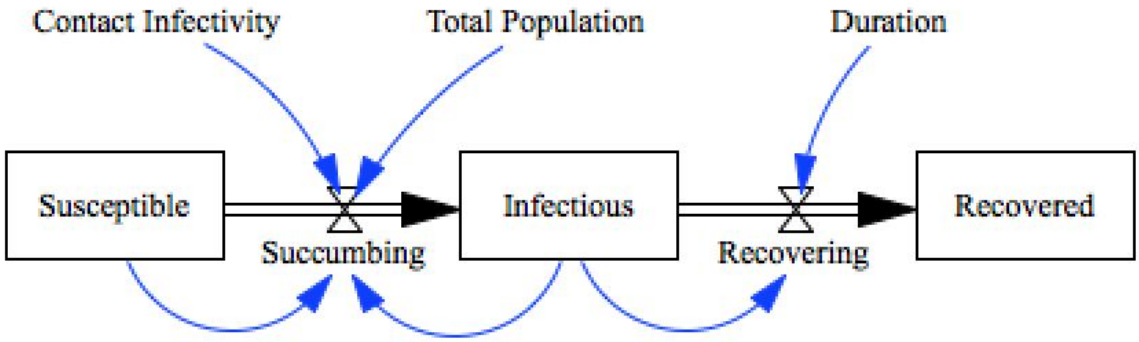


Рисунок 1. Модель распространения инфекции.

Хотя подход и был разработан в 50х годах, широкое распространение модели системной динамики получили с появлением персональных компьютеров в середине 80х. В то же время появилось и программное

обеспечение для симуляций и разработки. В последнее время множество компаний, консультирующих агентств и государственных предприятий используют данный подход для анализа критически важных проблем.

Системно-динамический подход подразумевает:

- Определение проблемы глобально.
- Представление понятий из рассматриваемой области в качестве непрерывных величин взаимосвязанных в циклах с обратной связью.
- Определение независимых накопителей (уровней) в системы и входящих и исходящих потоков.
- Создание модели, способной воспроизводить рассматриваемую проблему. Обычно это компьютерная симуляция, выраженная через нелинейные уравнения.
- Оптимизацию производственных или иных процессов на основе данных полученных при изучении системно-динамической модели.

1.3 Технологии

Для описания и решения задач системной динамики используется различное ПО и различные языки описания моделей. Первым из таких языков является DYNAMO [6]. В настоящий момент для построения моделей в основном используются диаграммы и пользовательский интерфейс, встроенной в ПО. В таком случае язык описания модели может быть скрыт от пользователя и не подходит для работы напрямую с текстом, например, язык описания моделей Vensim. Однако некоторыми программными продуктами помимо этого поддерживается язык описания моделей XMILE (Vensim, STELLA). XMILE не привязан к какому-то конкретному ПО, в проекте

кафедры ИПМ “SDCloud” есть поддержка данного языка и он позволяет описывать модели в виде текста. Поэтому для данной работы исходным языком описания моделей был выбран XMILE.

Таким образом, целью данной работы является написание транслятора языка XMILE в код, с которым может работать специальное ПО, конфигурирующее ПЛИС для симуляции входной системно-динамической модели. Для этого нужно решить следующие задачи:

- Разбор XML-файла, с закодированной в нем с помощью XMILE моделью;
- Построение графа синхронного потока данных, реализующего вычисление одного шага симуляции;
- Генерация кода, в соответствии с правилами, обусловленными вычислительной платформой NITTA

2. Язык XMILE

XMILE - это открытый язык описания моделей системной динамики в разных программных средах, в основе которого лежит язык описания моделей системной динамики SMILE, который выражен в XML [7]. SMILE имеет много общего с языком DYNAMO. При этом SMILE может быть закодирован и другими способами, помимо XMILE.

Преимущества, которые дает XMILE:

- Унифицированность формата для различных сред симуляции
- Возможность работать с моделью в обычном текстовом редакторе
- Отсутствие привязанности к какому-либо проприетарному ПО
- Возможность использовать системы контроля версий такие как Git и SVN

Помимо описания поведения моделей язык также позволяет построить графическое представление модели в виде диаграммы, однако в данной работе эта часть протокола рассматриваться не будет.

2.1 Общая структура

Общая структура модели описанной согласно протоколу XMILE выглядит следующим образом:

- Вся модель описана в файле с кодировкой UTF-8
- Содержит корневой XML-тег - <xmile>
- Содержит тег <header> с информацией о происхождении файла и системных требованиях.

- Файл может содержать более одной модели. Корневой XML-тег для модели - <model>
- (Опционально) Содержит тег <sim_specs> с параметрами симуляции по умолчанию.
- (Опционально) Содержит тег <model_units>, где описаны условные единицы, используемые в модели
- (Опционально) Содержит тег <dimensions> с массивом констант.
- (Опционально) Содержит тег <data> с информацией о постоянных данных.

Корневой элемент:

```
<xmile version="1.0"
     xmlns="http://docs.oasis-open.org/xmile/ns/XMILE/v1.0">
   ...
</xmile>
```

Рассмотрим подробнее секции, которые содержать информацию, необходимую для построения системы уравнений, описывающих модель.

```
<model>
   <sim_specs>
     <start>...</start>
     <stop>...</stop>
     <dt>...</dt>
   </sim_specs>
   <behavior>
     ...
   </behavior>
   <variables>
     <stock name=”...”>
          <inflow>...</inflow>
          <outflow>...</outflow>
          <eqn>...</eqn>
     </stock>
     <flow name=”...”>
          <eqn>...</eqn>
```

```
      </flow>
      <aux name="...">
            <eqn>...</eqn>
      </aux>
   </variables>
   <views>
      ...
   </views>
</model>
```

В секции sim_specs определены следующие параметры:

- start - время начала симуляции
- stop - время окончания симуляции
- dt - шаг симуляции

В секции behavior содержатся какие-либо ограничения накладываемые на компоненты модели. В данной работе эти ограничения не учитываются.

В секции variables могут находиться следующие вложенные элементы:

- Stock - Накопитель. Содержит начальное значение и входящие и исходящие потоки. Начальное значение задается в теге <eqn>, входящие потоки перечисляются по идентификаторам в тегах <inflow>, исходящие - в <outflow>
- Flow - Поток. Содержит формулу для расчета изменений происходящих в системе в теге <eqn>. Должен быть связан с каким-то накопителем.
- Aux - Вспомогательный элемент. Содержит математическое выражение в теге <eqn>, которое может быть переиспользовано в других вспомогательных элементах, накопителях или потоках.

Каждый из этих элементов обязательно имеет атрибут name (имя) - валидный XMILE-идентификатор и элемент <eqn>, содержащий выражение, задающее начальное значение накопителя или формулу.

2.2 Идентификаторы

Идентификатор элемента XMILE может включать в себя произвольные строки внутри кавычек (“”) с экранированными специальными символами. В общем случае, идентификаторы состоят из одного или более символа включая латинские буквы (A-Z, a-z), нижнее подчеркивание (“_”), знак доллара (“$”), цифры (0-9) и символы Юникода больше 127-ми. Идентификаторы не начинаются с цифры или знака доллара и не начинаются и не заканчиваются нижним подчеркиванием. Любой идентификатор может быть взят в кавычки, которые не являются частью идентификатора. Допустимые для экранирования внутри кавычек символы:

- \” - кавычки
- \n - перенос строки
- \\ - обратный слэш

Идентификаторы не чувствительны к регистру.

2.3 Пространства имен

Протокол XMILE использует пространство имен (xmlns), например "http://docs.oasis-open.org/xmile/ns/XMILE/v1.0". Это нужно учитывать при разборе.

Касательно областей видимости пространства имен делятся на четыре категории: область xml-тегов, область имен переменных, область имен

функций и область имен единиц измерения. Пространство имен тегов глобально и не пересекается с остальными. Пространство имен единиц измерения относится к набору моделей в файле и не пересекается с области имен функций и переменных. Область имен функций сочетает в себе глобально определенные функции и функции, определенные в рамках модели. Имена переменных определены только в рамках конкретной модели. Однако имена переменных и имена функций находятся в одном контексте и в связи с этим эти множества имен не должны пересекаться для избежания неопределенности.

2.4 Математические выражения

Математические выражения заключены между тегами <eqn>...</eqn>. Простейшим выражением является константа, то есть какое-то число. Выражения используют алгебраическую (infix) форму записи, то есть аргументы стоят по обе стороны от оператора, и следуют общим алгебраическим правилам приоритетности действий: действие в скобках, возведение в степень, умножение/деление и сложение/вычитание (в порядке уменьшения приоритета). Для функций и операторов выходящих за рамки привычной алгебры приняты установившиеся правила Си-подобных языков программирования.

2.5 Пример

Следующий пример иллюстрирует объявление накопителя и потока. При этом в потоке используется вспомогательный элемент.

```
<flow name="graduating">
   <eqn>Students*graduating_coef</eqn>
```

```
</flow>

<stock name="Students">
   <eqn>1000</eqn>
   <inflow>matriculating</inflow>
   <outflow>graduating</outflow>
</stock>

<aux name=”graduating_coef”>
     <eqn>0.8</eqn>
</aux>
```

3. Моделирование

Математически, базовая структура формальной системно-динамической модели - это система связанных нелинейных дифференциальных уравнений первого порядка. $\frac{d}{dt}x(t) = f(x, p)$, где x - вектор накопителей, p - это множество параметров, а $f(x, p)$ - нелинейная функция, возвращающая вектор значений.

В сложных системах часто возникают ситуации, когда невозможно получить результаты аналитическим путем решения. Однако, решение системы уравнений может быть получено путем приближенных вычислений с помощью. При расчете моделей системной динамики в основном используются следующие численные методы: метод Эйлера и метод Рунге-Кутта 2-го порядка [8].

3.1 Методы интегрирования

Симуляция таких систем выполняется за счет разбиения времени симуляции на дискретные интервалы длины dt и расчета значений системы по шагам в один такой интервал. Значение каждого накопителя рассчитывается исходя из его значения на предыдущем шаге и его связанного суммарного потока, то есть суммы входящих потоков за вычетом исходящих. Метод Эйлера (1) требует меньше вычислительных ресурсов, однако дает наименее точные результаты.

$$S(t) = S(t - dt) + dt * S'(t - dt) \quad (1)$$

$$S(t) = S(t - dt) + dt * \frac{S'(t-dt)+S'(t)}{2} \quad (2)$$

Временной интервал *dt* выбирается достаточно малым для того, чтобы не иметь заметного эффекта на протекающие в симуляции процессы. В общем случае, чем меньше шаг симуляции, тем точнее полученные результаты, но больше времени потребуется на вычисления. Также метод Рунге-Кутта 2-го (2) порядка дает более точные результаты, но требует дополнительно вычислять производную от накопителя на момент следующего шага симуляции [9].

В современном ПО для симуляций доступны более сложные численные методы для интегрирования. При генерации транслятором синхронного потока данных, реализующего интегрирование исходной модели, необходимо учитывать по какому именно методу будут производиться расчеты. Поэтому пользователю дана возможность выбирать требуемый метод интегрирования. В текущей версии транслятора доступен только метод Эйлера.

4. Облачная система моделирования SdCloud

SdCloud - облачная платформа для вычисления, хранения и обмена моделями системной динамики. Основная цель проекта - объединение существующих утилит для решения задач системной динамики в одном сервисе и предоставление широкой аудитории пользователей удобного инструмента для симуляции системно-динамических моделей.

На данный момент поддерживает следующие языки описания моделей:

- Vensim
- XMILE

Симуляция данных моделей выполняется с помощью библиотеки PySD, позволяющей преобразовывать входные модели в исполняемые модули на языке программирования Python. Для моделей Vensim также доступен вариант использования транслятора в язык Си - SDEverywhere.

4.1 SdCloud-ES

Модели системной динамики, решающие реальные задачи, например моделирование жизни города (“Индустриальная динамика”), отличаются высокой сложностью. Для вычисления таких моделей требуется много вычислительных ресурсов. В связи с этим для ускорения облачных вычислений было решено использовать реконфигурируемые аппаратные вычислители на базе ПЛИС. Так появился проект SdCloud-ES, целью которого является разработка вычислительной платформы для моделирования динамических систем на специализированных вычислителях на базе вычислительной платформы - NITTA.

5. Платформа NITTA

NITTA - реконфигурируемая вычислительная платформа (ВПл) реального времени на базе ПЛИС, позволяющая создавать специализированные вычислители, соответствующие особенностям системно-динамического моделирования [10]. Уровневая организация данной вычислительной платформы выглядит следующим образом:

- Прикладной уровень – высокоуровневый язык описания моделей XMILE.
- Промежуточный уровень – универсальное представление прикладных алгоритмов, представляющее собой расширенный вариант модели вычислений синхронных потоков данных (СПД). Средствами СПД описывается один шаг моделирования. Для тонкой оптимизации на этом уровне можно вручную отредактировать алгоритм.
- Внутренний уровень – совокупность архитектуры целевого процессора, моделей его вычислительных блоков, распределения прикладного алгоритма по вычислительным ресурсам и расписания пересылок данных между блоками. Множество реализованных вычислительных блоков определяет степень поддержки всех функций прикладного языка.
- Уровень реализации - совокупность программного и аппаратного обеспечения, автоматически преобразуемого в целевую систему.

В рамках данной работы разрабатывается транслятор из языка прикладного уровня в представление промежуточного уровня.

5.1 Промежуточный уровень NITTA

На промежуточном уровне вычислительной платформы NITTA синхронные потоки данных кодируются на языке Haskell. В рамках проекта существуют модули реализующие функциональные блоки. Функциональный блок – это элемент схемы синхронного потока данных, принимающий данные на свои входы и формирующий значение на своих выходах, согласно своей функции.

Функциональные блоки вычислительной платформы NITTA, используемые в данной работе (см. Приложение 4):

- Loop - блок, позволяющий закольцевать поток данных для циклического исполнения. Имеет свое начальное значение и запоминает последнее значение на своем входе до начала следующей итерации. Последнее значение с предыдущей итерации подается на выходы, если итерация первая, то на выходах будет начальное значение.
- Constant - блок, поддерживающий на своих выходах некое постоянное значение.
- Add - блок-сумматор. Имеет два входа. На выходах генерирует арифметическую сумму значений со входов.
- Sub - блок, аналогичный сумматору, но на выходах генерирует арифметическую разность значений со входов.
- Mul - блок, генерирующий на выходах арифметическое произведение значений со входов. Два входа.

- Div - блок, генерирующий на выходах арифметическое частное значений со входов. Два входа.

В текущей версии платформы функциональные блоки поддерживают вычисления с фиксированной запятой. В общем случае, вычисления могут проводиться с числами с плавающей запятой или с фиксированной в зависимости от решаемых задач. Формат с фиксированной запятой также может использоваться для оптимизации, если погрешность допустима в рамках решаемой задачи. Целочисленная арифметика требует менее сложных вычислительных блоков.

Согласно протоколу XMILE данные также могут иметь тип. Транслятор может учитывать его, внедряя дополнительные конвертирующие элементы в цепочку исполнения синхронного потока данных, однако это не реализовано в текущей версии.

6. Синхронный поток данных

Синхронный поток данных (СПД) это модель вычислений впервые предложенная Эдвардом Ли в 1986 году. При использовании данной модели вычислений все операции и обмен данными размечены статично. То есть реализация алгоритма, выраженного в виде синхронного потока данных всегда гарантировано занимает конечное время и конечную память для выполнения всех задач. Таким образом, цепочка операторов СПД может быть выполнена повторно без требования дополнительных ресурсов. Такой подход удобен для цифровой обработки сигналов и коммуникационных систем, где часто приходится обрабатывать бесконечный объем данных. Так же как и при симуляции моделей системной динамики [11].

Граф СПД состоит из узлов и линий передачи данных. Каждый узел является исполняемым оператором. В общем случае каждая линия передачи представляет собой FIFO-очередь с конкретными значениями данных любого типа - токенами [12]. В подмножестве СПД, используемом в проекте NITTA на линии хранится одно значение. Граф СПД подчиняется следующим правилам:

1) Оператор исполняется сразу как только достаточно данных доступно на всех его входах. Операторы исходных данных всегда активны.
2) Когда оператор исполняется, он всегда производит и поглощает строго определенное количество токенов.
3) Путь данных по графу не может зависеть от конкретных входных значений.

4) Задержка – это свойство линии передачи данных. Таким образом задержка осуществляется добавлением токенов в очередь данной линии перед началом исполнения.

Из второго правила следует, что данные, поступившие на вход оператора исчезают из буфера на входящей линии данных и уже не восстанавливаются. Таким образом, когда выполнение цепочки операторов завершено, входные данные отсутствуют на входах.

Третье правило означает, что СПД граф не может содержать условных ветвлений, зависимых от данных, таких как “if-then-else”, “switch” или условных циклов (например, “for”), однако сами операторы могут содержать подобные конструкции, потому что организация СПД графа независима от того, как внутри работают входящие в него операторы (принцип черного ящика). Все возможные пути графа вычисляются, независимо от входных данных, а выбор конкретного результата осуществляется мультиплексором.

7. Транслятор XMILE в промежуточный уровень ВПл NITTA

Задача транслятора языка XMILE в промежуточный уровень вычислительной платформы NITTA заключается в построении синхронного потока данных, реализующего симуляционные вычисления (интегрирование по времени) модели системной динамики, поданной на вход, и кодировании их средствами языка промежуточного уровня вычислительной платформы NITTA. Также для упрощения отладки и большей наглядности результатов работы было решено визуализировать результат работы транслятора в виде графа.

Как было сказано в разделе 3.1 “Методы интегрирования” для текущей версии транслятора выбор методов интегрирования ограничен только методом Эйлера. Таким образом, задача предоставления пользователю выбора метода интегрирования отложена для следующих версий.

Для разработки транслятора из прикладного языка в описание алгоритма в виде синхронного потока данных используется язык программирования Python 3.6.5. Для решения задач транслятора требуется решать задачи обработки и изменения строк. Python позволяет работать со строками как со списками, это удобнее чем аналоги в других рассмотренных языках (C#, Java), где работа со строками осуществляется методами substring(), indexOf() и т.п.. При этом Python предоставляет возможность реализовать полноценное объектно-ориентированное программирование в отличие от также рассмотренного Haskell. В связи с необходимостью

считывать, хранить и обрабатывать модели системной динамики и структуру графа СПД, удобнее использовать объектно-ориентированную парадигму. Python не требует установки каких-то дополнительных пакетов, кроме самого интерпретатора (в отличие от JRE, .NET Runtime), что упрощает интеграцию транслятора в какие-либо существующие решения. Также одним из критериев выбора языка программирования является наличие встроенных средств для обработки XML-файлов, однако такие средства есть почти во всех современных инструментах разработки, поэтому данный критерий несущественен.

Для решения задачи разбора XML-файла язык программирования Python предоставляет встроенный пакет xml.etree.ElementTree. Данный модуль предоставляет API для поиска узлов в xml-дереве, извлечения дочерних элементов и считывания атрибутов узлов [13]. Этого достаточно для поставленной задачи разбора xml-файла и считывания данных из необходимых узлов (то, какие узлы нужны определяется спецификацией XMILE), однако также были рассмотрены сторонние библиотеки: untangle и xmltodict [14]. Их было решено не использовать, так как в рамках данной задачи, эти библиотеки не дали бы выигрыша в скорости разработки и только влекут за собой лишние зависимости.

Для решения задачи визуализации было решено использовать утилиту graphviz, так как она предоставляет API для рендера PDF с графом, закодированным в языке описания графов dot, а для генерации кода на языке dot есть есть сторонняя библиотека для Python [15].

7.1 Разбор XML-файла модели

Первым шагом работы транслятора является разбор XML-файла с закодированной в нем посредством XMILE моделью системной динамики и создание объекта класса Model, который хранит в себе все необходимые поля модели. Удобно хранить все параметры модели в одном объекте, так как для построения синхронного потока данных, нужно знать и постоянно обращаться к таким сущностям, как stocks, flows и aux, а при дальнейшей работе нам также понадобится имя модели и шаг симуляции dt.

Объект класса Model содержит в себе следующие элементы:

- Имя модели;
- Время начала симуляции;
- Время окончания симуляции;
- Временной интервал симуляции;
- Список накопителей в виде списка объектов класса Stock;
- Список потоков в виде списка объектов класса Flow;
- Список вспомогательных элементов в виде списка объектов класса Aux.

Объекты классов Stock, Flow и Aux формируются по мере считывания новых узлов модели и добавляются к объекту класса Model.

Объект класса Stock содержит следующие поля:

- Имя накопителя;
- Список идентификаторов входящих потоков;
- Список идентификаторов исходящих потоков;
- Объект класса Equation определяющий начальное значение накопителя.

Объекты классов Flow и Aux содержат поля:

- Имя потока/вспомогательного элемента;
- Объект класса Equation определяющий формулу, связанную с этим элементом (может быть константой).

Задача класса Equation – решить проблему разбора математических выражений в infix записи, т.е. когда оператор записывается между аргументами. Этот класс преобразует строку, содержащую математическое выражение в набор из аргументов и оператора. Ключевой особенностью является то, что аргументы внутри объектов этого класса могут также быть математическими выражениями, именами переменных (ссылками) или числами.

Если внутри объекта класса Equation содержатся объекты класса Equation, представляющие собой математическое выражение, то это означает, что внешний объект представляет собой последнее действие в исходном математическом выражении, а вложенные объекты - последующие действия. Максимально вложенные Equation представляют собой числа или ссылки. Таким образом, любой Equation можно вычислить рекурсивно вычисляя оба вложенных в него Equation-аргумента.

7.2 Построение синхронного потока данных.

Для формирования синхронного потока данных вводится еще одна сущность - функциональный блок (FunctionalBlock).

Объекты класса FunctionalBlock имеют следующие поля:

- Имя;

- Имя функции;
- Имена аргументы;
- Список идентификаторов входов;
- Список идентификаторов выходов.

Также объекты этого класса умеют генерировать идентификаторы выходов так, чтобы они были уникальными.

Задача этого класса – предоставить возможность построения графа синхронного потока данных. Это достигается за счет генерирования уникальных имен выходов у одних функциональных блоков и добавления их в список входов других.

Идентификаторы входов и выходов функциональных блоков формируются по следующим правилам:

- Название выхода = Название блока + номер выхода;
- Каждый выход может быть присоединен только к одному входу.

Объекты этого класса формируются из объектов класса Equation. Каждый функциональный блок представляет одну из математических операций, которую нужно совершить в рамках вычисления одного шага симуляции. Например выражение “a + b * c” превратится в функциональные блоки Mul(b, c) и Add(a, b*c). Если блок представляет собой последнюю операцию отображаемого Equation, то он носит имя элемента, который содержал данный Equation. Это сделано для удобства просмотра результатов трансляции и вычислений. Названия промежуточные функциональных блоков – исходные строки математических выражений, которые они представляют.

После построения списка функциональных блоков на основе списка объектов класса Equation создается блок constant “t_step” со значением шага симуляции dt, который нужен для вычисления изменений произошедших именно за это время. Вместе с ним создается блок loop “t”, являющийся накопителем и служащим для отсчета времени симуляции и блок add(t, t_step), который осуществляет добавление шага симуляции к текущему значению в накопителе “t”.

На случай, если какому-либо из блоков не найдется достаточно входных аргументов, создается блок constant “zero”, который подается на входы блоков, у которых не хватает входных аргументов.

После этого для каждого потока каждого накопителя, кроме времени создаются блоки “stock_delta_in_flow” и “stock_delta_out_flow” отображающие входящие и выходящие потоки. Блок “stock_delta” считает суммарный поток, вычитая выходящий из входящего (если какого-то из них нет, то на вход подается “zero”). Затем блок add складывает значение delta с текущим значением в блоке.

После того, как все блоки созданы, начинается построение графа. (см. рисунок 2)

В процессе построение графа из функциональных блоков, также осуществляется генерация кода для последующей визуализации этого графа с помощью инструмента Digraph из библиотеки graphviz. Каждый функциональный блок является узлом графа, а входы и выходы определяют направленные ребра графа. Ребро направлено от узла с выходом в узел, в который этот выход ведет.

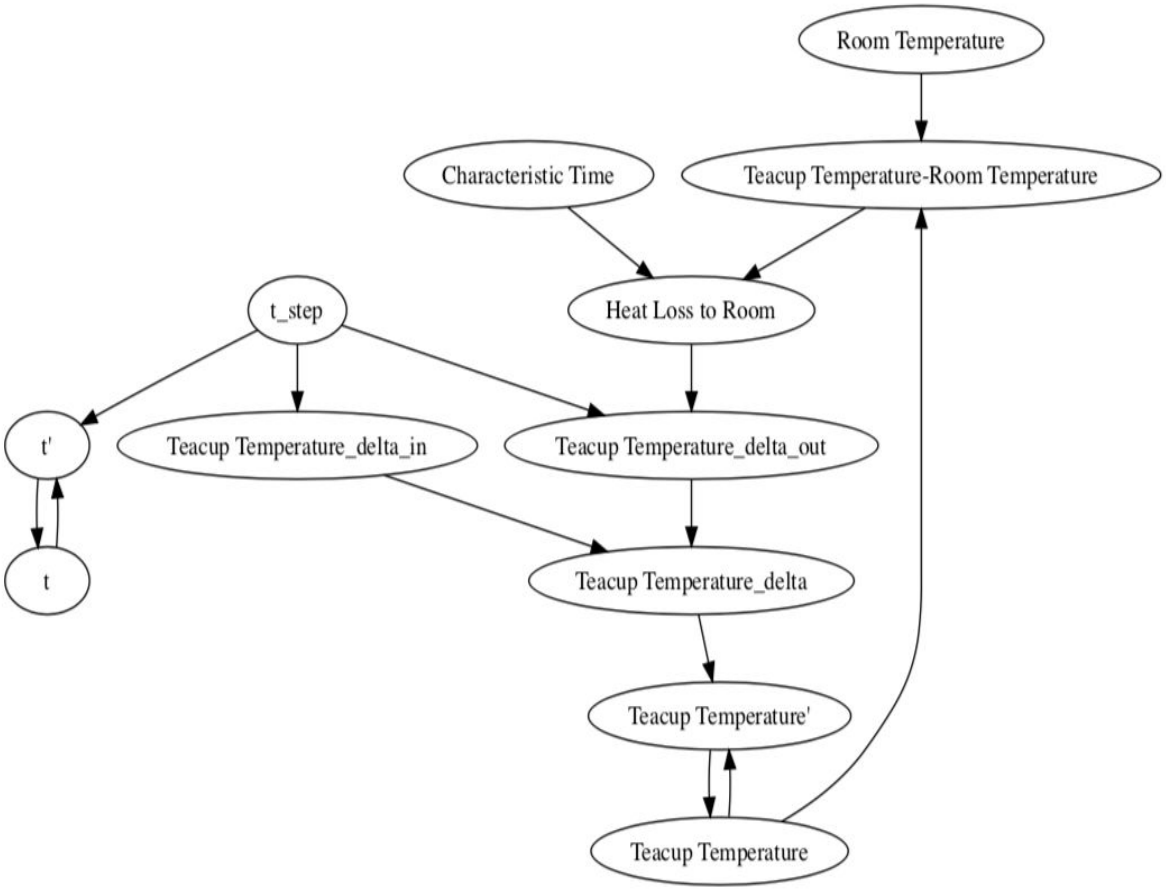


Рисунок 2. Результат отрисовки графа СПД, сгенерированного транслятором

Следующий этап – генерация целевого кода.

7.3 Генерация кода промежуточного уровня вычислительной платформы NITTA

Транслятор должен генерировать код, понятный вычислительной платформе NITTA, для этого нужно сначала описать правила форматирования и синтаксиса этого кода.

После изучения исходного кода функциональных блоков вычислительной платформы NITTA и примеров алгоритмов, реализованных на ней, были выявлено, что синтаксис описания расширенной версии

синхронного потока данных в данной вычислительной платформе выглядит следующим образом:

- Список задействованных функциональных блоков заключен в квадратные скобки (“[…]”)
- Функциональные блоки разделяются запятыми (“,”)
- Названия функциональных блоков состоят из строчных букв английского алфавита и имеют префикс “FB.”. Пример: “FB.loop”, “FB.add”, “FB.mul” и т.п.
- Соединения между блоками определяются идентификаторами их входов и выходов. Идентификатор должен быть использован в качестве выхода и входа функционального блока только по одному разу. Идентификатор - строка, подчиняющаяся синтаксису языка Haskell.
- Идентификаторы выходов перечисляются в квадратных скобках через запятую.
- Для арифметических блоков (add, sub, div, mul) два идентификатора входов указываются после названия блока через пробел, затем через пробел перечисляются идентификаторы выходов.
- Для блоков constant и loop целочисленное значение указывается после названия блока. После этого через пробел указываются идентификаторы выходов. Для блока loop после этого через пробел указывается идентификатор входа.
- Числовые константы преобразуются в вид с фиксированной запятой. Количество знаков после запятой определяется параметром, при этом запятая не пишется. (например, три знака после запятой: 125 = 0.125)

  Формат:

```
[ FB.operator "<inputA>" "<inputB>" ["<output0>",…,"<outputN>"]
, FB.loop <initial_value> ["<output0>",…,"<outputN>"] "<input>"
, FB.constant <value> ["<output0>",…,"<outputN>"]
]
operator = add | sub | div | mul.
```

Пример:

```
[ FB.constant 125 ["t_step0", "t_step1"]
, FB.loop 0 ["t0"] "next_t0"
, FB.add "t0" "t_step0" ["next_t0"]
]
```

С предыдущего шага имеется множество функциональных блоков. Для генерации требуемого кода, по очереди генерируем описания всех блоков.

Для каждого блока, генерируется код, согласно выше перечисленным требованиям. У каждого объекта класса FunctionalBlock для этого есть все необходимые поля.

7.4 Использование транслятора

Использование транслятора осуществляется из командной строки вызовом интерпретатора python3. Вторым аргументом передается точка входа в программу, т.е. скрипт convert.py. Третьим аргументом передается путь к файлу модели.

Например:

```
> python3 src/convert.py models/teacup.xml
```

7.5 Тестирование

Для тестирования работы транслятора выполнялась трансляция тестовых моделей и получение кода промежуточного представления платформы NITTA. Если при трансляции возникали ошибки, то поиск ошибок осуществлялся добавлением подробных логов и выполнением некоторых функций в интерпретаторе, независимо от остальной части программы.

Если код был получен, то он запускался в симуляционной среде платформы NITTA. Для этого использовалась утилита для работы с Haskell проектами “stack”. Данная утилита позволяет установить Haskell-инфраструктуру, включающую в себя GHC (компилятор Haskell) и собрать проект.

Чтобы получить результат симуляции, нужно поместить полученный из транслятора код в функцию simulateModel, запустить GHCi (интерпретатор Haskell) и выполнить функцию simulateModel, указав количество шагов.

Для тестирования работы транслятора использовалась системно-динамическая модель Teacup, моделирующая остывание кружки в комнате и модель SIR, моделирующая распространение инфекции. Данные модели взяты из открытого доступа, также к ним прилагаются эталонные значения, полученные в результате симуляции [16].

Текущая версия транслятора успешно транслирует модели, подобные модели Teacup и SIR. Полученные результаты симуляции на платформе NITTA близки к эталонным. Результаты отображены в таблице 1.

Таблица 1. Результаты тестирования модели Teacup

Time	Heat loss to room		Teacup temperature	
	Эталон	Тест	Эталон	Тест
0	11	11	180	180
125	10.86	11	178.625	178.625
250	10.72	11	177.267	177.25
375	10.59	11	175.926	175.875
500	10.46	10	174.602	174.5
675	10.33	10	173.295	173.25
750	10.2	10	172.004	172
875	10.07	10	170.729	170.75
1000	9.94	10	169.469	169.5

Заключение

В рамках данной выпускной квалификационной работы был разработан транслятор из языка описания моделей системной динамики XMILE в язык описания расширенного синхронного потока данных промежуточного представления вычислительной платформы NITTA.

На начальном этапе работ были изучены основные концепции построения моделей системной динамики и рассмотрены существующие программные решения. На основе этой информации было решено использовать язык описания моделей XMILE. На втором этапе работ изучена спецификация языка XMILE и промежуточный уровень вычислительной платформы NITTA. Это позволило определиться с требованиями к транслятору.

Были реализованы алгоритмы разбора системно-динамических моделей закодированных посредством языка XMILE, разбора математических выражений, построение на основе считанных данных графа СПД, реализующего вычисление одного шага симуляции модели с помощью метода Эйлера и генерации кода промежуточного уровня платформы NITTA на основе этого графа.

Для первой версии было решено транслировать только базовый функционал языка XMILE, который используется в модели “Teacup”. Полученный транслятор успешно решает эту задачу. В дальнейшем можно расширять функционал транслятора, учитывая больше функций протокола XMILE и улучшая алгоритм разбора выражений, и параллельно с этим необходимо развивать функционал вычислительной платформы NITTA.

Например, добавлять новые функциональные блоки, которые затем могут быть использованы в трансляторе.

Список литературы

1. Список программных продуктов для симуляции системно-динамических моделей [Электронный ресурс] URL: https://www.systemdynamics.org/core-software
2. OASIS XML Interchange Language (XMILE) for System Dynamics Overview [Электронный ресурс] URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xmile
3. Introduction to system dynamics [Электронный ресурс] URL: http://lm.systemdynamics.org/what-is-s/ дата обращения: 07.04.2018
4. Forrester, J. W., “Industrial Dynamics - After the first Decade”. Management Science Vol. 14 No. 7, Март, 1968. URL: http://www.sfu.ca/~vdabbagh/Forrester68.pdf
5. Sterman, J. D. (2000). “Business Dynamics: System Thinking and Modeling for a Complex World”. Irwin/McGraw-Hill, 2000, с. 231-241.
6. “Introduction to System Dynamics Modeling with Dynamo” (1981), George P. Richardson; Alexander L. Pugh III, Pegasus Communications
7. Спецификация XMILE. [Электронный ресурс] URL: http://docs.oasis-open.org/xmile/xmile/v1.0/cos01/xmile-v1.0-cos01.html
8. “Mathematics behind system dynamics”, T Choopojcharoen, A. Magzari, 2012, с. 28-33.
9. Endre Su ¨li (2014). “Numerical solution of ordinary differential equations” Mathematical Institute, University of Oxford. с 4-19

10. Пенской А.В. Проектирование вычислительной платформы для моделирования динамических систем. / Пенской А. В. Санкт-Петербург 2018.
11. Introduction to Synchronous Dataflow [Электронный ресурс] URL: http://users.ece.utexas.edu/~bevans/courses/ee382c/lectures/08_sdf/sdf.html, дата обращения: 20.04.2018
12. Shuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Press, Norwell, MA, ISBN 0-7923-9722-3, 1996
13. Документация Python 3, xml.etree.ElementTree [Электронный ресурс] URL: https://docs.python.org/3/library/xml.etree.elementtree.html
14. XML parsing in python [Электронный ресурс] URL: http://docs.python-guide.org/en/latest/scenarios/xml/
15. Документация утилиты graphviz [Электронный ресурс] URL: https://graphviz.gitlab.io/documentation/
16. Git-репозиторий с моделями системной динамики, использованных в тестах. [Электронный ресурс] URL : https://github.com/SDXorg/test-models/tree/master/samples

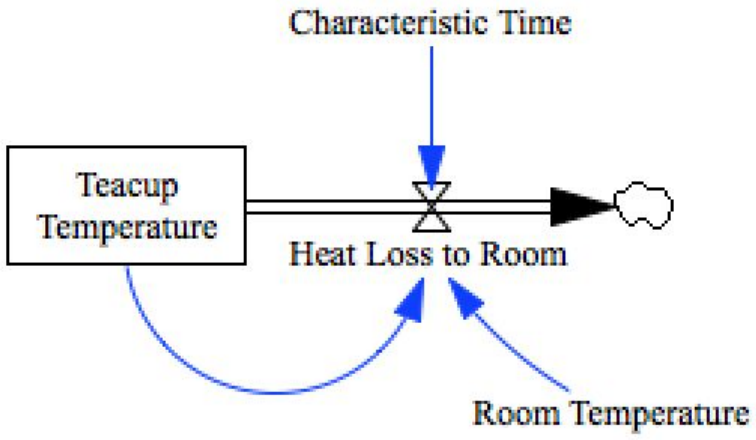
Приложения

Приложение 1

“Teacup” XMILE model

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xmile version="1.0" xmlns="http://docs.oasis-open.org/xmile/ns/XMILE/v1.0">
   <header>
       <vendor>James Houghton</vendor>
       <name>Teacup</name>
       <options>
           <uses_outputs/>
       </options>
       <product version="1.0">Hand Coded XMILE</product>
   </header>
   <sim_specs>
       <stop>30.0</stop>
       <start>0.0</start>
       <dt>0.125</dt>
   </sim_specs>
   <model>
       <variables>
           <flow name="Heat Loss to Room">
               <doc>Heat Loss to Room</doc>
                  <eqn>("Teacup Temperature"-"Room Temperature")/"Characteristic
Time"</eqn>
           </flow>
           <aux name="Room Temperature">
               <doc>Ambient Room Temperature</doc>
               <eqn>70</eqn>
           </aux>
           <stock name="Teacup Temperature">
```

```
            <doc>The average temperature of the tea and the cup</doc>
            <outflow>"Heat Loss to Room"</outflow>
            <eqn>180</eqn>
        </stock>
        <aux name="Characteristic Time">
            <eqn>10</eqn>
        </aux>
    </variables>
</model>
</xmile>
```



Приложение 2

Листинг файла convert.py

```
import sys
import re
import xml.etree.ElementTree
import math
from Equation import Equation
from graphviz import Digraph

class FunctionalBlock:
   def __init__(self, name, function, args):
       self.name = name
       self.function = function
       self.args = args
       self.outputs = []
       self.inputs = []

   def __eq__(self, other):
       return self.name == other.name

   def __str__(self):
       argsStr = ", ".join(map(lambda a: str(a), self.args))
       inputs = ", ".join(self.inputs)
       outputs = ", ".join(self.outputs)
       return self.name + ": " + self.function + "(" + argsStr + ")\n\tInputs:
[" + inputs + "]\n\tOutputs: [" + outputs + "]\n"
   def __repr__(self): return str(self)

   def addInput(self, inpt):
       self.inputs.append(inpt)

   def output(self):
       index = len(self.outputs)
       output = self.name + str(index)
       self.outputs.append(output)
       return output

class Model:
   def __init__(self, name, flows, stocks, auxies, start, stop, dt):
       self.name = name
       self.flows = flows
       self.stocks = stocks
```

```
        self.auxies = auxies
        self.start = start
        self.stop = stop
        self.dt = dt
        # raise NameError(alias + " not found in flows, stocks and auxies")

class Flow:
   def __init__(self, name, eqn):
        self.name = name
        self.eqn = Equation(eqn)

   def __str__(self):
        return "\"" + self.name + "\": " + str(self.eqn)
   def __repr__(self): return str(self)

class Stock:
   def __init__(self, name, eqn, inflows, outflows):
        self.name = name
        self.eqn = Equation(eqn)
        self.inflows = inflows
        self.outflows = outflows

   def __str__(self):
        return "\"" + self.name + "\": " + str(self.eqn)
   def __repr__(self): return str(self)

class Aux:
   def __init__(self, name, eqn):
        self.name = name
        self.eqn = Equation(eqn)

   def __str__(self):
        return "\"" + self.name + "\": " + str(self.eqn)
   def __repr__(self): return str(self)

def parse_xmile(filename):
   ns = { "xmile": "http://docs.oasis-open.org/xmile/ns/XMILE/v1.0" }
   tree = xml.etree.ElementTree.parse(filename)
   root = tree.getroot()

   name = root.find("xmile:header", ns).find("xmile:name", ns).text

   simSpecs = root.find("xmile:sim_specs", ns)
   start = simSpecs.find("xmile:start", ns).text
   stop = simSpecs.find("xmile:stop", ns).text
   dt = simSpecs.find("xmile:dt", ns).text

   variables = root.find("xmile:model", ns).find("xmile:variables", ns)
```

```
    auxies = []
    for aux in variables.findall("xmile:aux", ns):
        name = aux.get("name")
        eqn = aux.find("xmile:eqn", ns).text
        auxies.append(Aux(name, eqn))

    stocks = []
    for stock in variables.findall("xmile:stock", ns):
        name = stock.get("name")
        eqn = stock.find("xmile:eqn", ns).text
        inflows = []
        for inflow in stock.findall("xmile:inflow", ns):
            inflows.append(inflow.text)
        outflows = []
        for outflow in stock.findall("xmile:outflow", ns):
            outflows.append(outflow.text)
        stocks.append(Stock(name, eqn, inflows, outflows))

    flows = []
    for flow in variables.findall("xmile:flow", ns):
        name = flow.get("name")
        eqn = flow.find("xmile:eqn", ns).text
        flows.append(Flow(name, eqn))

    return Model(name, flows, stocks, auxies, start, stop, dt)

def symbolToFBName(symbol):
    symbols = "+-*/"
    names = ["add", "sub", "mul", "div"]
    return names[symbols.index(symbol)]

def listExpressions(expression, lst, name = ""):
    if len(name) == 0:
        name = expression.text.replace("\"", "")
    args = []
    if expression.tokens[0].type == Equation.TYPE_EXPRESSION:
        args.append(expression.tokens[0].text.replace("\"", ""))
        listExpressions(expression.tokens[0], lst)
    else:
        args.append(expression.tokens[0].tokens[0])
    if expression.tokens[2].type == Equation.TYPE_EXPRESSION:
        args.append(expression.tokens[2].text)
        listExpressions(expression.tokens[2], lst)
    else:
        args.append(expression.tokens[2].tokens[0])
    operator = symbolToFBName(expression.tokens[1].tokens[0])
    lst.append(FunctionalBlock(name, operator, args))
```

```
def resolveToNubmer(eqn, model):
   if eqn.type == Equation.TYPE_NUMBER:
       return eqn.tokens[0]
   if eqn.type == Equation.TYPE_EXPRESSION:
       raise NotImplementedError("Expressions not supported")
   for namedEqn in model.auxies:
       if namedEqn.name == eqn.text:
           return resolveToNubmer(namedEqn.eqn, model)

def build_sdf_model(model):
   constants = []
   fbs = []
   stocks = []
   for namedEqn in model.flows + model.auxies:
       if namedEqn.eqn.type == Equation.TYPE_EXPRESSION:
           listExpressions(namedEqn.eqn, fbs, namedEqn.name )
   for stock in model.stocks:
       val = resolveToNubmer(stock.eqn, model)
       stocks.append(FunctionalBlock(stock.name, "loop", [val, stock.name +
"'"]))
   for aux in model.auxies:
       if aux.eqn.type == Equation.TYPE_NUMBER:
           constants.append(FunctionalBlock(aux.name, "constant",
[aux.eqn.tokens[0]]))

   constants.append(FunctionalBlock("t_step", "constant", [str(model.dt)]))
   zero = FunctionalBlock("zero", "constant", "0")
   constants.append(zero)

   fbs.append(FunctionalBlock("t'", "add", ["t", "t_step"]))
   stocks.append(FunctionalBlock("t", "loop", [0, "t'"]))

   for stock in model.stocks:
       previousFlow = None
       for inflow in stock.inflows:
           flowName = inflow.replace("\"", "")
           if previousFlow != None:
               fbs.append(FunctionalBlock(stock.name + "_" + flowName + " + " +
previousFlow, "add", [previousFlow, flowName]))
           previousFlow = flowName
       fbs.append(FunctionalBlock(stock.name + "_delta_in", "mul",
[previousFlow, "t_step"]))
       previousFlow = None
       for outflow in stock.outflows:
           flowName = outflow.replace("\"", "")
           if previousFlow != None:
```

```
                fbs.append(FunctionalBlock(stock.name + "_" + flowName + " + " +
previousFlow, "add", [previousFlow, flowName]))
            previousFlow = flowName
        fbs.append(FunctionalBlock(stock.name + "_delta_out", "mul",
[previousFlow, "t_step"]))
        fbs.append(FunctionalBlock(stock.name + "_delta", "sub", [stock.name +
"_delta_in", stock.name + "_delta_out"]))
        fbs.append(FunctionalBlock(stock.name + "'", "add", [stock.name,
stock.name + "_delta"]))

    dot = Digraph(comment=model.name)

    for fb in fbs + stocks:
        dot.node(fb.name)
        for arg in fb.args:
            connected = False
            for fb_in in (fbs + constants + stocks):
                if arg == fb_in.name:
                    fb.addInput(fb_in.output())
                    dot.edge(fb_in.name, fb.name)
                    connected = True
                    break
            if not (connected or type(arg) is float or type(arg) is int):
                fb.addInput(zero.output())

    print(fbs, constants, stocks, sep='\n--------------------------------\n',
end='\n--------------------------------\n')
    print(dot.source)
    return (fbs, constants, stocks, dot)

def generate_haskell_fb_code(fbs, constants, stocks):
    fbSrc = "[ "
    nodes = []
    for constant in constants:
        node = "FB." + constant.function + " " + str(int(float(constant.args[0])
* 1000)) + " [" + ", ".join(map(lambda x: '"' + x + '"', constant.outputs)) +
"]"
        nodes.append(node)
    for stock in stocks:
        node = "FB." + stock.function + " " + str(int(float(stock.args[0]) *
1000)) + " [" + ", ".join(map(lambda x: '"' + x + '"', stock.outputs)) + "] " +
"\"" + stock.inputs[0] + "\""
        nodes.append(node)
    for fb in fbs:
        node = "FB." + fb.function + " " + " ".join(map(lambda x: "\"" + x +
"\"", fb.inputs)) + " [" + ", ".join(map(lambda x: '"' + x + '"', fb.outputs))
+ "]"
        nodes.append(node)
```

```
    fbSrc += "\n, ".join(nodes)
    fbSrc += " :: FB (Parcel String Int)\n]"
    return fbSrc

if len(sys.argv) != 2:
    print("Usage:\nxmile2sdf.py <input_file.xml>")
    exit(0)
print("\n================================\nSTARTED\n============================
=====\n")
print("Opening", sys.argv[1])
model = parse_xmile(sys.argv[1])
print("\n================================\nXMILE ->
DONE\n================================\n")
sdf = build_sdf_model(model)
print("\n================================\nSDF ->
DONE\n================================\n")
print(generate_haskell_fb_code(sdf[0], sdf[1], sdf[2]),
end="\n================================\nFB ->
DONE\n================================\n")
sdf[3].render('output/graph.gv')
```

Приложение 3

Листинг файла Equation.py

```
class Equation:

    TYPE_NUMBER = "NUMBER"
    TYPE_REFERENCE = "REFERENCE"
    TYPE_EXPRESSION = "EXPRESSION"
    TYPE_OPERATOR = "OPERATOR"

    DIGITS = "0123456789"
    OPERATORS = "*/+-"
    CONSTANTS = ["pi", "e"]
    FUNCTIONS = ["sqrt", "pow", "sin", "cos", "tg"]
    QUOTES = "\"\'"

    @staticmethod
    def getType(token):
        if type(token) is Equation:
            return token.type
        if len(token) == 1 and token in Equation.OPERATORS:
            return Equation.TYPE_OPERATOR
        if token.isdigit():
            return Equation.TYPE_NUMBER
        for op in list(Equation.OPERATORS) + Equation.FUNCTIONS:
            if op in token:
                return Equation.TYPE_EXPRESSION
        return Equation.TYPE_REFERENCE

    def __init__(self, text):
        self.text = text
        self.type = Equation.getType(text)
        self.tokens = self.parse(text)
```

```
    def __str__(self):
        return str(self.tokens) + " -> " + self.type

    def __repr__(self):
        return str(self)

    def __eq__(self, other):
        if not type(other) is Equation or len(self.tokens) != len(other.tokens):
            return False
        for i, self_token in enumerate(self.tokens):
            if self_token != other.tokens[i]:
                    return False
        return True

    # returns token and next index
    def readToken(self, equationString, i):
        # skip whitespaces
        while i < len(equationString) and equationString[i] == ' ':
            i += 1
           # if token starts with brace ( then it is equationString wrapped in
braces
        if equationString[i] == '(':
            # scan to closing brace
            j = i + 1
            openingBracesCount = 1
            while j < len(equationString) and openingBracesCount > 0:
                j += 1
                if equationString[j] == '(':
                    openingBracesCount += 1
                elif equationString[j] == ')':
                    openingBracesCount -= 1
            # exclude braces from returning value
            return (equationString[i + 1:j], j + 1)
        # if token starts with quote then it is var name
        if equationString[i] in Equation.QUOTES:
```

```
            # scan to next matching quote
            j = i + 1
                    while j < len(equationString) and equationString[j] !=
equationString[i]:
                j += 1
            # exclude quotes in returning value
            token = equationString[i:j]
            token = token.replace("\"", "")
            return (token, j + 1)
        # if token starts with digits then it is number
        if(equationString[i] in Equation.DIGITS):
            # scan to last digit in number
            j = i + 1
                    while j < len(equationString) and equationString[j] in
Equation.DIGITS:
                j += 1
            return (equationString[i:j], j + 1)
        if(equationString[i] in Equation.OPERATORS):
            return (equationString[i], i + 1)
        # if token is not a number, operator or var name wrapped in quotes
        # then it is a var name without quotes and possible math constant
        j = i + 1
                while j < len(equationString) and equationString[j] not in
(Equation.DIGITS + " " + Equation.OPERATORS):
            j += 1
        token = equationString[i:j+1]
        if token in Equation.CONSTANTS:
            # replace constant with its value maybe?
            return (token, j + 1)
        elif token in Equation.FUNCTIONS:
            # todo: consider function token format
            return (token, j + 1)
        else:
            return (token, j + 1)

    def splitInTokens(self, equationString):
```

```
        tokens = []
        i = 0
        while i < len(equationString):
            t = self.readToken(equationString, i)
            tokens.append(Equation(t[0]))
            i = t[1]
        return tokens

    def parse(self, equationString):
        tokens = []
        if self.type == Equation.TYPE_EXPRESSION:
            tokens = self.splitInTokens(equationString)
        elif self.type == Equation.TYPE_NUMBER:
            tokens = [int(self.text)]
        elif self.type == Equation.TYPE_OPERATOR:
            tokens = [self.text]
        elif self.type == Equation.TYPE_REFERENCE:
            tokens = [self.text.replace("\"", "").replace("'", "")]
        return tokens

#print(Equation('("Teacup    Temperature"-"Room    Temperature")/"Characteristic
Time"'))
```

Приложение 4

Используемые функциональные блоки вычислительной платформы NITTA

```
data Loop io = Loop (X io) (O io) (I io) deriving ( Typeable )
deriving instance ( IOType io v x ) => Show (Loop io)
deriving instance ( IOType io v x ) => Eq (Loop io)
loop x bs a = FB $ Loop (X x) (O $ fromList bs) $ I a

instance ( IOType io v x ) => FunctionalBlock (Loop io) v where
 inputs  (Loop _ _a b) = variables b
 outputs (Loop _ a _b) = variables a
 insideOut _ = True
instance ( Ord v, Show v, Show x ) => FunctionSimulation (Loop (Parcel v x)) v x where
 simulate cntx (Loop (X x) (O v2) (I v1)) = do
   let x' = fromMaybe x $ cntx `get` v1
   set cntx v2 x'

data Add io = Add (I io) (I io) (O io) deriving ( Typeable )
deriving instance ( IOType io v x ) => Show (Add io)
deriving instance ( IOType io v x ) => Eq (Add io)
add a b c = FB $ Add (I a) (I b) $ O $ fromList c

instance ( IOType io v x ) => FunctionalBlock (Add io) v where
 inputs  (Add  a  b _c) = variables a `union` variables b
 outputs (Add _a _b  c) = variables c
 dependency (Add a b c) = [ (y, x) | x <- elems $ variables a `union` variables b
                                   , y <- elems $ variables c
                                   ]
instance ( Ord v, Num x ) => FunctionSimulation (Add (Parcel v x)) v x where
 simulate cntx (Add (I k1) (I k2) (O k3)) = do
   v1 <- cntx `get` k1
   v2 <- cntx `get` k2
   let v3 = v1 + v2
   set cntx k3 v3
```

```
data Sub io = Sub (I io) (I io) (O io) deriving ( Typeable )
deriving instance ( IOType io v x ) => Show (Sub io)
deriving instance ( IOType io v x ) => Eq (Sub io)
sub a b c = FB $ Sub (I a) (I b) $ O $ fromList c

instance ( IOType io v x ) => FunctionalBlock (Sub io) v where
 inputs  (Sub  a  b _c) = variables a `union` variables b
 outputs (Sub _a _b  c) = variables c
 dependency (Sub a b c) = [ (y, x) | x <- elems $ variables a `union` variables b
                                   , y <- elems $ variables c
                                   ]
instance ( Ord v, Num x ) => FunctionSimulation (Sub (Parcel v x)) v x where
 simulate cntx (Sub (I k1) (I k2) (O k3)) = do
   v1 <- cntx `get` k1
   v2 <- cntx `get` k2
   let v3 = v1 - v2
   set cntx k3 v3

data Mul io = Mul (I io) (I io) (O io) deriving ( Typeable )
deriving instance ( IOType io v x ) => Show (Mul io)
deriving instance ( IOType io v x ) => Eq (Mul io)
mul a b c = FB $ Mul (I a) (I b) $ O $ fromList c

instance ( IOType io v x ) => FunctionalBlock (Mul io) v where
 inputs  (Mul  a  b _c) = variables a `union` variables b
 outputs (Mul _a _b  c) = variables c
 dependency (Mul a b c) = [ (y, x) | x <- elems $ variables a `union` variables b
                                   , y <- elems $ variables c
                                   ]
instance ( Ord v, Num x ) => FunctionSimulation (Mul (Parcel v x)) v x where
 simulate cntx (Mul (I k1) (I k2) (O k3)) = do
   v1 <- cntx `get` k1
   v2 <- cntx `get` k2
   let v3 = v1 * v2
   set cntx k3 v3

data Div io = Div (I io) (I io) (O io) deriving ( Typeable )
deriving instance ( IOType io v x ) => Show (Div io)
```

```
deriving instance ( IOType io v x ) => Eq (Div io)
div a b c = FB $ Div (I a) (I b) $ O $ fromList c

instance ( IOType io v x ) => FunctionalBlock (Div io) v where
 inputs  (Div  a  b _c) = variables a `union` variables b
 outputs (Div _a _b  c) = variables c
 dependency (Div a b c) = [ (y, x) | x <- elems $ variables a `union` variables b
                                   , y <- elems $ variables c
                                   ]
instance ( Ord v, Num x, Integral x ) => FunctionSimulation (Div (Parcel v x)) v x
where
 simulate cntx (Div (I k1) (I k2) (O k3)) = do
   v1 <- cntx `get` k1
   v2 <- cntx `get` k2
   let v3 = fromIntegral v1 / fromIntegral v2 :: Double
   set cntx k3 $ round v3

data Constant io = Constant (X io) (O io) deriving ( Typeable )
deriving instance ( IOType io v x, Show x ) => Show (Constant io)
deriving instance ( IOType io v x, Eq x ) => Eq (Constant io)
constant x vs = FB $ Constant (X x) $ O $ fromList vs

instance ( IOType io v x, Show x, Eq x, Typeable x ) => FunctionalBlock (Constant io)
v where
 outputs (Constant _ o) = variables o
instance ( Ord v ) => FunctionSimulation (Constant (Parcel v x)) v x where
 simulate cntx (Constant (X x) (O k))
   = set cntx k x
```

Приложение 5

Результат работы транслятора для модели “Teacup”

```
[ FB.constant 70000 ["Room Temperature0"] :: FB (Parcel String Int)
   , FB.constant 10000 ["Characteristic Time0"]
   , FB.constant 125 ["t_step0", "t_step1"]
   , FB.constant 0 ["zero0"]
     , FB.loop 180000 ["Teacup Temperature0", "Teacup Temperature1"] "Teacup
Temperature'0"
   , FB.loop 0 ["t0"] "t'0"
    , FB.sub "Teacup Temperature0" "Room Temperature0" ["Teacup Temperature-Room
Temperature0"]
     , FB.div "Teacup Temperature-Room Temperature0" "Characteristic Time0"
["Heat Loss to Room0"]
   , FB.add "t0" "t_step0" ["t'0"]
   , FB.mul "Heat Loss to Room0" "t_step1" ["Teacup Temperature_delta_out0"]
          , FB.sub "zero0" "Teacup Temperature_delta_out0" ["Teacup
Temperature_delta0"]
      , FB.add "Teacup Temperature1" "Teacup Temperature_delta0" ["Teacup
Temperature'0"]
   ]
```