# 1 Введение

В последние годы во многих областях науки, таких как, например, био-информатика или экспериментальная физика, требуется обрабатывать все большее и большее количество данных. В качестве примера можно привести Большой Адронный Коллайдер (БАК). В результате одного эксперимента поток «сырых» данных с детектора *ALICE* может превышать 1 гигабайт в секунду [1].

Для обработки больших объемов данных используются распределенные вычислительные системы, мощности которых могут достигать пентафлопов. Так, для обработки данных БАК используется сложная распределенная система *LHC Computing Grid*. Такие мощности достигаются за счет громадного количества вычислительных узлов. Например, супер-компьютер Sequoia мощностью более 1 пентафлопа имеет более миллиона ядер [2].

Однако с ростом объемов данных и, как следствие, необходимых вычислительных ресурсов увеличивается сложность задач эффективной обработки. Известно, что принципы, лежащие в основе эффективных алгоритмов, могут существенно усложняться при переходе к параллельным вычислениям и даже при изменении масштабов вычислительных систем.

Другой проблемой становится сложность создания самих распределенных алгоритмов, так как многие из них теряют наглядность из-за реализации большого количества технических аспектов, которые необходимо учитывать при их разработке. Более того, некоторые распределенные системы не имеют четких границ, так как задействованные в ней участники могут постоянно меняться, что дополнительно усложняет контроль за вычислениями. Так, для ускорения обработки результатов БАК был задействован проект *LHC@home* [3], помочь которому мог любой желающий путем предоставления части вычислительной мощности собственного компьютера.

Одним из решений описанных выше проблем стала концепция *потоков данных* (*dataflow*). Под потоком данных обычно понимают модель представления алгоритмов. Описание вычислительного процесса в такой концепции, обычно включает в себя следующие элементы:

- набор атомарных исполнителей-функций, реализованных вне концепции потоков данных, например, в виде программ или динамических библиотек;
- описание зависимостей между исполнителями — результат работы одних исполнителей передается на вход (в качестве аргументов) другим;
- описание стратегии поведения потока данных;
- иерархичность — иногда целый поток данных может быть представлен в виде одного исполнителя внутри другого потока данных.

Иными словами, поток данных представляется в виде наборов исполнителей, каждый из которых выполняет некое преобразование поступивших ему на вход данных, соединенных каналам передачи данных.

Концепция потоков данных обладает важным преимуществом — она полностью изолирует создателя алгоритма от необходимости описывать многие технические аспекты вычислений. В этом они родственны декларативным языкам программирования, в которых программы выглядят как наборы определений или описаний зависимостей. В частности, некоторые потоки данных можно без потери структуры преобразовать к программе, например, на языке Haskell [4].

Другими преимуществами описания вычислительных процессов в виде потоков данных являются широкие возможности для автоматического распределения ресурсов при запуске потока данных в распределенных средах, а также для предварительного анализа необходимых ресурсов для эффективного

запуска. Эти свойства являются крайне востребованными для научных вычислений, так как объемы вычислений явно требуют больших ресурсов недостижимых на системах с одним потоком выполнения. Как в примере с проектом *LHC@home* потоки данных также хорошо описывают вычисления в динамических распределенных средах. Возможность предварительной оценки требуемых ресурсов важна для эффективной утилизации мощностей супер-компьютеров, вычислительные способности которых могут заметно превосходить требуемые ресурсы. Задаче получения таких оценок посвящена данная работа.

Родственной к потокам данных является концепция потоков управления (controlflow), основное отличие которых от потоков данных в том, что связи между исполнителями означают передачу управления, и часто исполнители имеют доступ к общему хранилищу данных. Программы в терминах потоков управления имеют явно указанные участки параллельного выполнения, создания и слияния параллельных ветвей.

Эти две концепции объединяют под названием поток работ (workflow). Среди достоинств концепции потока работ следует выделить следующие:

- изолированность программы, сценария или схемы от технических особенностей среди выполнения;
- наглядность программ — чаще всего поток работ можно представить в графической нотации близкой к блок-схемам;
- простота создания программ — создание программ, особенно в графической нотации, не требует специальных знаний из области информатики, что позволяет, например, физикам-экспериментаторам не тратить большое количество времени на изучение новых языков или сред обработки результатов.

В отличии от потоков управления потоки данных, как уже было упомянуто, ориентированы на научные вычисления. Среди особенностей можно выделить следующие:

- работа в распределенных средах сложной динамичной структуры, ресурсы которой могут быть заранее неизвестны;
- высокая отказоустойчивость,
- работа с большим объемом данных.

Заметим, что среды построения и запуска потоков данных значительно отличаются по внутренним механизмам от аналогичных сред для потоков управления из-за перечисленных выше отличий в концепциях.

Также стоит отметить, что потоки данных имеют много общего с *сетями процессов Кана* (*Kahn Process Networks*), сходства и различия с которыми будет рассмотрено ниже.

Отметим также растущую популярность концепции потоков данных в научных кругах — так в описанном выше примере с БАК, для обработки результатов используются потоки данных или близкие к ним концепции [1].

## 1.1 Мотивация

В данной работе рассматривается модель потоков данных, которая, по мнению автора, предоставляет наиболее естественное описание программ для научных расчетов в концепции потоков данных. Также

рассматриваемая модель близка к моделям существующих и активно используемых сред для построения и запуска потоков данных.

В данной работе следует выделить несколько задач:

- построение формальной модели потока данных;
- анализ потока данных на наличие ошибок;
- оценка ресурсов и стратегий, требуемых для эффективного запуска потока данных в распределенных средах.

Главные алгоритмы были реализованы в виде модуля языка Python для возможности использования в интерактивном интерпретаторе.

## 1.2 Обзор существующих моделей представления потоков данных

Среди моделей представления потоков данных следует выделить следующие:

- программы на высокоуровневых языках программирования;
- графы;
- сети Петри;
- сети процессов Кана.

Как уже было сказано, концепция сетей процессов Кана имеет некие отличия от рассматриваемой в этой работе концепции потоков данных, однако определение потоков данных может варьироваться и обозначает скорее направление в представлении алгоритмов обработки, поэтому можно считать сети процессов Кана, если не одной из моделей представления потоков данных, то как минимум родственной моделью, и следовательно, заслуживающей рассмотрения как минимум для демонстрации возможных подходов.

Более подробно описания представлений потоков данных можно найти в [5]

### 1.2.1 Программы на высокоуровневых языках программирования

Любой вычислительный процесс можно наглядно описать в терминах современных языков программирования при должном уровне абстракции от технических особенностей среды выполнения. Наиболее подходящими языками для описания потоков данных являются скриптовые языки, например, Python, либо декларативные языки, например, Haskell или Erlang. Декларативная парадигма, как уже было сказано, достаточно близка концепции потоков данных, однако порой декларативные языки слишком строги для описания потоков данных.

Основной проблемой такого подхода является недостаточная наглядность для людей не знакомых с конкретным языком программирования и требует знания возможных тонкостей этого языка. Более того, обычно визуальное преставление является более наглядным для человека, чем текстуальное. Этим можно объяснить меньшую популярность такого подхода.

В качестве примера системы, использующей языки программирования как представление потока данных, можно привести систему *GridAnd* [6].

### 1.2.2 Графы

Графы являются самой популярной моделью потоков данных. Главным достоинством является возможность интуитивно понятного визуального представления графов, понимание которого не требует специальных знаний в области информатики.

Другим достоинством графов как модели представления потока данных является возможность построения иерархичных потоков данных, где один поток данных, представляется в виде вершины в потоке более высокого уровня, тем самым реализуя принципы модульности и абстракции.

Особо стоит выделить хорошо изученные свойства графов и множество алгоритмов для их анализа, а так же выразительную силу графов. Например, в данной работе с помощью графов описываются сами потоки данных, качественное поведение каждого блока (в виде конечного автомата), поведение блока во время запуска потока данных (также в виде автомата).

### 1.2.3 Сети Петри

Другой популярной концепцией представления потоков данных являются сети Петри, которые можно считать развитием идеи графов. Как и в случае с графами, сети Петри можно наглядно представить используя визуальные средства.

Также доступен широкий набор средств для анализа сетей Петри. Кроме того, в сетях Петри явно присутствуют элементы параллелизма, хотя сети Петри лучше подходят для описания потоков управления (чем возможно объясняется их популярность в системах построения потоков управления), нежели для потоков данных, так как переход так называемых маркеров от одного узла к другому лучше описывает передачу управления, нежели передачу данных. Однако представление потоков данных в виде сетей Петри также возможно.

Более подробно сети Петри описаны, например, в [7] и [8].

### 1.2.4 Сети процессов Кана

Сети процессов Кана используются для описания потоков данных ориентированных на потоковую обработку, в отличии от рассматриваемой в этой работе модели, где данные подаются на вход поэлементно или одной «порцией». Формально в этой концепции каждый процесс (исполнитель) представляется функцией, которая преобразует входящие потоки данных целиком, и лишь в частном случае это эквивалентно поэлементной обработке. В таких случаях мы получим концепцию близкую к концепции потоков данных. Однако в сетях процессов Кана для разрешения возможных неопределенностей в поведении процессов накладываются ограничения на функции, которыми представляется процесс, вводятся правила поглощения и испускания данных; в концепции потоков данных вместо наложения ограничений на функции стараются анализировать саму схему потока данных, которая, так как не является потоковой, сама по себе может не допускать потенциальных неопределенностей [4].

Стоит также отметить, что сети процессов Кана, аналогично потокам данных, могут быть естественно представлены с помощью декларативных языков программирования. Например, используя функции преобразующие бесконечные списки в языке Haskell, можно добиться точного соответствия формального определению сетей процессов Кана [4].

## 1.3 Неформальное описание модели

В качестве основы для построения модели были взяты основные принципы работы популярных сред построения и запуска потоков данных. Сам поток данных представляется в виде ориентированного гра-

фа, вершинами которого являются блоки. Блоки реализуют некий алгоритм преобразования данных и в зависимости от способа создания этого алгоритма делятся на два типа:

- атомарные — встроенные в систему построения и запуска потоков данных в виде внутренних подпрограмм;
- составные — блок, который сам представляется в виде потока данных и является вложенным потоком данных.

У каждого блока (кроме специальных или «фиктивных») существуют наборы входных и выходных портов. Входные порты в некотором смысле являются «аргументами» блока — по ним передаются данные для преобразования. Выходные порты имеют смысл результата выполнения преобразования — по ним данные передаются для дальнейшей обработки. Каждый порт обычно обладает определенным смыслом, который определяется алгоритмом преобразования.

Ребро графа потока данных соединяет выходной порт одного блок с входным портом другого блока, задавая тем самым, последовательность обработки данных. Ребра можно также рассматривать как FIFO-каналы, передача по которым ради простоты будем считать мгновенной.

Естественно, что работа потока данных зависит от внутренних состояний программ его блоков. В данной работе алгоритмы блоков заменяются на конечные автоматы Мили, описывающие зависимости наборов выходных данных от входных. Таким образом мы получаем качественное описание работы блоков, что существенно упрощает задачу анализа, освобождая от необходимости описывать анализ алгоритмов блоков, который выходит за рамки рассматриваемых вопросов.

Стоит отметить, что обычно данные имеют тип, однако рассмотрение типов данных приводит к усложнению модели, не приводя при этом ни к каким дополнительным результатам.

## 1.4 Базовые понятия и определения

В данном разделе будут введены базовые понятия, используемые в данной работе, в том числе, понятия атомарного блока, составного блока, потока данных, автомата Мили соответствующего блоку и так далее.

Основной единицей потока данных является атомарный блок.

**Определение 1.** ***Шаблоном атомарного блока*** *$s$ назовем кортеж имеющий два непересекающихся набора портов: входной и выходной. $s = (FA, I, O)$, где:*

- $I = \{b_i^I | i = \overline{1, N_I}, N_I \in \mathbb{N}\}$ *— конечное множество уникальных входных портов;*
- $O = \{b_i^O | i = \overline{1, N_O}, N_O \in \mathbb{N}\}$ *— конечное множество уникальных выходных портов;*
- $FA$ *— конечный автомат Мили, смысл и структура которого будет обсуждаться в дальнейшем.*

Сразу заметим, что не любая тройка $(FA, I, O)$ является шаблоном атомарного блока, так как $FA$ должен соответствовать множествам $I$ и $O$.

Стоит отметить, что природа множеств $O$ и $I$ совершенно не важна, например, они могут быть просто множеством чисел: $I, O = \{i \in \overline{1, \ldots, N_{I,O}}\}$, на практике часто используют строки для именования портов, так как порты могут существенно различаться по смыслу, а их именование помогает избежать путаницы. Последний подход используется в рисунках.

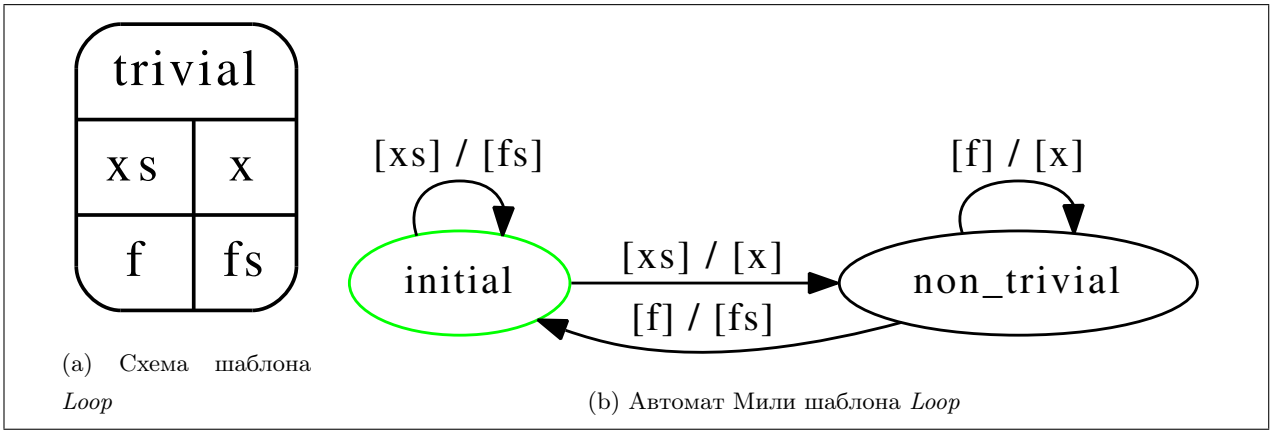
**Определение 2.** ***Атомарным блоком*** *назовем $b = (id, s, state)$, где:*

- *id — уникальный для всех блоков идентификатор;*
- *s — шаблон атомарного блока;*
- *state ∈ S, где S множество состояний $FA_s$ — текущее состояние атомарного блока.*

Идентификатор *id* введен из-за следующих соображений: в потоке данных некие блоки могут описываться одним и тем же шаблоном, находиться в одном и том же состоянии, но все равно будут являться различными блоками, в частности, могут перейти в различные состояния в следующие моменты времени либо могут иметь в потоке данных разный смысл и, соответственно, иметь различные соединения с другими блоками.

В реальных системах для построения и запуска потока данных шаблон атомарного блока представляется в виде подпрограммы или модуля (функции, класса некого объектно-ориентированного языка программирования, динамической библиотеки), который используя данные некого набора своих входных портов, вычисляет и записывает данные в свои выходные порты. В таком случае, атомарный блок представляется в виде набора внутренних переменных, либо (что тоже самое) в виде экземпляра класса объектно-ориентированного языка программирования соответствующего шаблону. В такой интерпретации некую вычислимую функцию $f(x_1, x_2, \ldots, x_n)$ можно представить в виде шаблона блока $b_f$ с входными портами $x_1, x_2, \ldots, x_n$ и одним выходным портом $F$, который при получении данных со всех входных портов выписывает значения функции $f(x_1, x_2, \ldots, x_n)$ в порт $F$.

На рисунке 1 схематично изображен шаблон блока *Loop*, для которого $inputs(Loop) = \{xs, f\}$, $outputs(Loop) = \{fs, x\}$.




(a) Схема шаблона *Loop*

(b) Автомат Мили шаблона *Loop*

Рис. 1: Схематичное изображение шаблона блока *Loop* (a) и соответствующего ему конечного автомата Мили (b).

Основной структурой для построения схемы потока данных является составной блок.

**Определение 3.** ***Шаблоном составного блока*** *будем называть кортеж $c = (B, E, I, O)$, где:*

- *B - конечное множество блоков (сразу заметим, что блоком может являться и составной блок, определенный ниже).*
- *E - множество ребер вида $e_b = (b_1, b_1^O, b_2, b_2^I)$, $b_1, b_2 \in B$, $b_1^O \in outputs(b_1), b_2^I \in inputs(b)$, либо вида $e_I = (c^I, b, b^I)$, $e_O = (b, b^O, c^O)$, где $b \in B$, $b^I \in inputs(b)$, $b^O \in outputs(b)$, $c^O \in O$, $c^I \in I$. Множества ребер $e_b$ будем обозначать как $E^B$, $e_I$ и $e_O$ как $E^I$ и $E^O$ соответственно.*
- *I, O - конечные непересекающиеся наборы входных и выходных портов (по аналогии с атомарным блоком).*

В дальнейшем в выражениях вида $I_c$ или $O_c$ нижний индекс $c$ будет указывать на шаблон блока, которому принадлежат эти множества.

Для анализа внутренней структуры удобно рассматривать иное представление шаблона составного блока $c$, добавляя фиктивные блоки STOCK, $I_{\text{STOCK}} = O_{\text{STOCK}} = O_c$, и SOURCE, $I_{\text{SOURCE}} = O_{\text{SOURCE}} = I_c$:

$$\hat{E}^I = \{(\text{SOURCE}, \text{SOURCE}^O, b, b^I) | (\text{SOURCE}^O, b, b^I) \in E^I\}$$

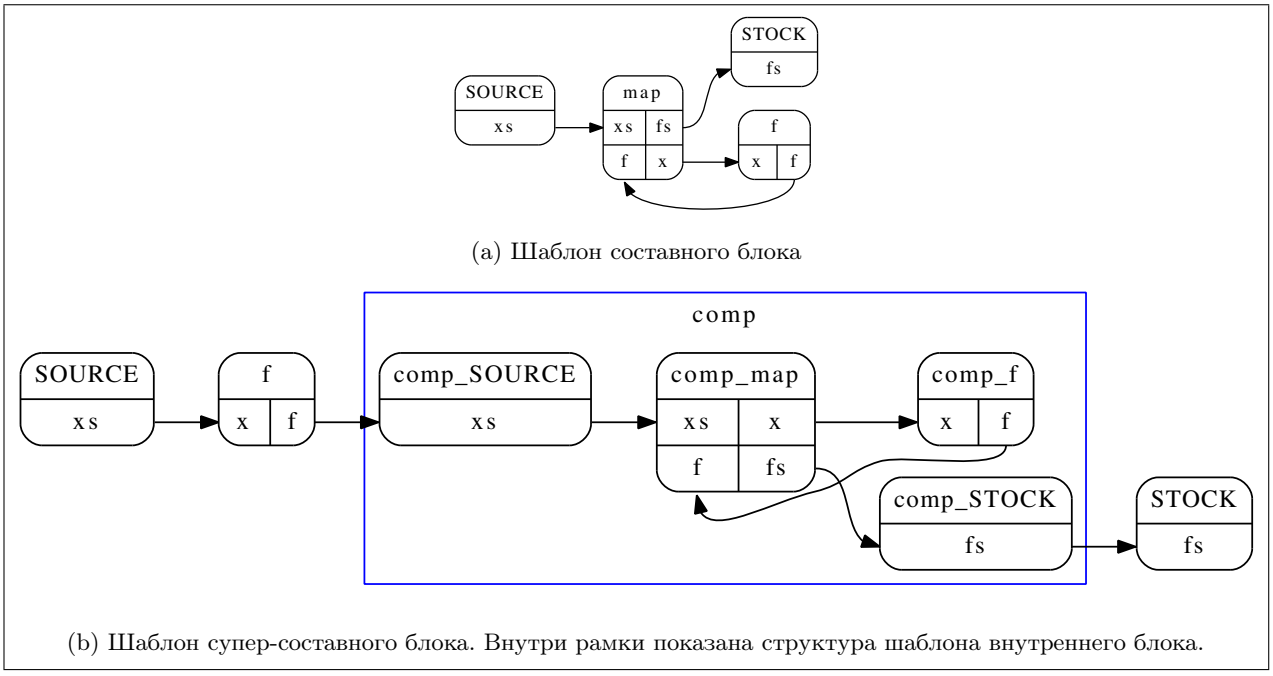
$$\hat{E}^O = \{(b, b^O, \text{STOCK}, \text{STOCK}^I) | (b, b^O, \text{STOCK}) \in E^O\}$$

$$\hat{c} = (B \cup \{\text{STOCK}, \text{SOURCE}\}, E^B \cup \hat{E}^I \cup \hat{E}^O)$$

В этом случае шаблон составного блока $\hat{c}$ описывается графом с ребрами вида $(u, u^O, v, v^I)$, а входными и выходными портами всего шаблона считаются входные и выходные порты блоков SOURCE и STOCK. На рисунке 2 изображен примеры шаблонов составных блоков во второй интерпретации.

**Определение 4.** ***Составным блоком*** *будем называть* $b = (id, c = (B, E, I, O), state)$*, где*

- $id$ *— уникальный идентификатор;*
- $c$ *— шаблон составного блока;*
- $state \in \Sigma \times 2^E = \prod S(B) \times 2^E$ *- текущие состояние составного блока, включающее состояние каждого блока (подпространство* $\Sigma = \prod S(B)$*), активную волну (*$\omega \in \Omega = 2^E$*).*




(a) Шаблон составного блока

(b) Шаблон супер-составного блока. Внутри рамки показана структура шаблона внутреннего блока.

Рис. 2: Примеры графов составного и супер-составного блока.

Заметим, что составной блок (в исходной интерпретации) внешне неотличим от атомарного блока. Поэтому под понятием блока мы будем подразумевать либо атомарный блок, либо составной блок, различая их только в том случае, когда речь заходит о их внутреннем строении. Составной блок может включать в себя другие составные блоки. На рисунке 2b показан пример составного блока, включающего другой составной блок, иными словами супер-составного блока. На практике свойство вложенности реализует принцип модульности программ — модулем в нашем случае будет шаблон составного блока.

Бессмысленно рассматривать случаи, в которых структура составного блока бесконечна в глубину, например, в случае когда шаблон составного блок содержит содержит блок того же шаблона. Поэтому в дальнейшем, мы будем рассматривать только составные блоки конечной глубины.

Заметим, что составным блоком можно описать любую программу, имея в распоряжении необходимые примитивы — шаблоны атомарных блоков, реализующие Тьюринг-полную систему операций. Доказательства этого факта выходит за задачи данной работы и может быть найдено, например, в работе [8]. Из-за достаточной выразительности составного блока, схему потока данных мы определим просто как схему некого составного блока.

Теперь неформально поясним схему работы составного блока и смысл его структуры. В общепринятых определениях потока данных ребра графа шаблона составного блока обозначают зависимость по данным, то есть ребро $e = (u, u^O, v, v^I)$ представляет собой *FIFO*-канал соединения между портами блоков и можно словесно интерпретировать следующим образом: блок $v$ в качестве данных из входного порта $v^I$ должен использовать данные выходного порта $u^O$ блока $u$ при их наличии. Блок $u$ после завершения своей работы может "испустить данные по порту $u^O$ которые мгновенно "переносятся"на все ребра из $u^O$ блока $u$, если эти ребра свободны. Когда блоку $v$ требуются данные из порта $v^I$, данные с ребра $e$ мгновенно "переносятся"в соответствующий порт блока. В этом и есть смысле подпространства $\Omega = 2^E$ в определении состояния составного блока — *активной волной* $\omega$ мы назовем множество ребер, которые содержат на данный момент данные.

Однако такая схема работы составного блока порождает множество неопределенностей в работе некоторых составных блоков, например, неопределенность выбора ребра для "поглощения"данных из определенного порта (состояние гонки). Неопределенности подобного рода мы будем рассматривать как ошибки, обнаружению которых посвящена часть данной работы. Такого рода неоднозначности, особенно в реальных системах, то есть в условиях неопределенного времени вычислений блоков и их параллельного выполнения, может привести к совершенно иному ходу потока данных, нежели было задумано автором потока данных. Поэтому наличие подобного рода неопределенностей (состояний гонки) говорит скорее о неправильном построении схемы потока данных, чем о некой задумке автора потока данных. Более подробно и строго об неоднозначностях в поведении потока данных будет идти речь ниже. Стоит заметить, что подобного рода неоднозначности возникают и в процессе определения сетей процессов Кана, которые решаются наложением условий на правила поглощения и испускания, а также на класс функции процесса. Мы же пойдем другим путем, накладывая ограничения лишь на граф шаблона потока данных, оставляя подпрограммы и правила поглощения каждого шаблона блока без ограничений. В нашем случае правила поглощения состоят в том, что формально блок может поглотить любые доступные данные из любых портов по любым ребрам, но будем расценивать любые возможные неоднозначности как ошибку построения потока данных.

Выше были рассмотрены определения и термины, которые с той или иной точностью присутствуют практически во всех моделях потока данных. В данной работе мы абстрагируемся от данных, а значит нас интересуют только возможные зависимости наборов входных портов от набора выходных портов, иными словами только качественное поведение блоков. Оказывается, конечный автомат Мили хорошо подходит для описания качественного поведения блоков. Заметим, что в данной работе используется не стандартное определение автомата Мили, а его недетерминированные аналог. Дадим теперь строгое определение конечного автомата Мили.

**Определение 5.** ***Конечный автомат Мили*** *— кортеж* $FA = (S, s_0, \Sigma, \Lambda, E)$*, где:*

- $S$ *— конечное множество состояний;*
- $s_0 \in S$ *— начальное состояние;*

- $\Sigma$ — *входной алфавит;*
- $\Lambda$ – *выходной алфавит;*
- $E \subseteq S \times \Sigma \times \Lambda \times S$ — *отношение возможных переходов.*

Для удобства будем рассматривать функцию конечного автомата Мили $FA = (S, \cdot, \Sigma, \Lambda, E)$, которую определим как $FA(p, \sigma) = \{(\lambda, q, \sigma') | (p, \sigma', \lambda, q) \in E, \sigma' \subseteq \sigma$. Стоит обратить внимание на нестандартное определение функции автомата. Далее символами в алфавитах $\Sigma$ и $\Lambda$ будут множества, поэтому функция по текущему состоянию и входному множеству возвращает множество троек вида: выходной символ-множество, новое состояние и поглощенное множество.

Конечный автомат Мили выполняет преобразование последовательности входных символов из алфавита $\Sigma$ в последовательность символов в алфавите $\Lambda$. Вернемся теперь к определению шаблона атомарного блока. В данной работе мы абстрагируемся от конкретного алгоритма преобразования входных данных в выходные и преобразования внутреннего состояния блока. Вместо этого мы будем описывать лишь качественное поведение алгоритма автоматом Мили $FA$, в котором входной алфавит $\Sigma = 2^I \setminus \varnothing$ (запрещается запуск по пустому набору входных портов), а выходной алфавит $\Lambda = 2^O$, иными словами ребро $e = (u, is, os, v)$ автомата $FA$ описывает возможный переход из состояния $u$ в состояние $v$, при считывании данных из портов $is$ и выписывании данных в порты $os$ (именно поэтому функция $FA$ имеет странный на первый взгляд вид). На практике подпрограмма реализующая логику атомарного блока может иметь значительное количество внутренних состояний, однако, для наших целей важны лишь группы таких внутренних состояний, которые описывают возможное поведение на определенном наборе данных из входных портов. Более того практически во всех системах построения и запуска потока данных подпрограмма описывающая блок имеет детерминированное поведение. Неизбежный недетерминизм в автомате Мили, описывающий блок, возникает из-за исключения из рассмотрения самих данных, так как при разных значениях на одном и том же наборе портов из одного состояния, блок может перейти в различные конечные состояние, даже выписав при этом данные в один и тот же набор портов.

**Пример 1.** *На рисунке 5b изображен автомат Мили для шаблона атомарного блока Loop, описывающего цикл. Поясним эту схему. Изначально блок с шаблоном Loop находиться в состоянии initial. Далее он может считать из порта xs начальные параметры цикла (например, некую коллекцию данных). Если условие выхода из цикла изначально выполнено (например, пустая коллекция), то блок переходит в прежнее состояние выписав в порт fs. Иначе, блок отправляет данные по порту x переходя в состояние non_trivial. В состоянии non_trivial блок имеет не тривиальное внутреннее состояние, например, остаток коллекции. В этом состоянии блок ожидает поступление преобразованных некой внешней функцией данных на порт f. И в этом случае существует два варианта перехода, в зависимости от выполнения условия выхода из цикла:* $(non\_trivial, \{f\}, \{x\}, non\_trivial)$ *и* $(non\_trivial, \{f\}, \{fs\}, trivial)$ *для случаев продолжения цикла и выхода из него соответственно.*

Конечный автомат Мили может быть также определен для составного блока. Единственное отличие от автомата Мили для атомарного блока состоит в том, что зная все автоматы внутренних блоков, можно вычислить автомат составного блока как будет показано ниже.

### 1.4.1 Алгоритм работы потока данных

Теперь опишем алгоритм вычисления потока данных, представленного в виде составного блока $c$ со схемой $C = (\hat{B}_c = B_c \cup \{\text{SOURCE}, \text{STOCK}\}, \hat{E}_c = E_c \cup E_c^I \cup E_c^O)$ в представлении графа. Мы будем

рассматривать вычисление потока данных в предположении, что время работы блока может быть абсолютно любым (но конечным) при любых условиях, что полностью соответствует реальным потокам данных. В таком случае нас прежде всего будет интересовать всевозможные поведения потока данных с точки зрения относительных расположений времен запуска и завершения любого блока. Введем модельное время $t \in \mathbb{Z}_+$. Каждое множество, связанное с состоянием системы в момент времени $t$, снабдим верхним индексом, например, $\omega^t \in \Omega_c$.

**Определение 6.** *Определим некоторые вспомогательные функции.*

$$\begin{aligned} O_v(\omega) &= \{v^O | (v, v^O, \cdot, \cdot) \in \omega\}, \\ I_v(\omega) &= \{v^I | (\cdot, \cdot, v, v^I) \in \omega\}, \\ \Delta_{O,v} &= \{(v, v^O, \cdot, \cdot) \in E_c | v^O \in O\}, \\ \Delta_{I,v}(\omega) &= \{\tilde{\omega}_{I,v} \subseteq \omega\},\ \forall \tilde{\omega}_I \forall v^I \in I \exists! e \in \omega_I : e = (\cdot, \cdot, v, v^I) \end{aligned}$$

Последняя функция возвращает множество вариантов поглощения по набору входных портов $I$ блока $v$.

**Определение 7.** *Фронтом $\Psi_c(\omega, s)$ активной волны $\omega = \{(u, u^O, v, v^I)\} \in \Omega_c$ в состоянии $(s = (s_{u_1}, s_{u_2}, \ldots, s_{u_m}), \omega, \cdot)$ назовем подмножество блоков $\Psi_c(\omega, s) \subseteq B_c$, такое что:*

$$\Psi_c(\omega, s) = \{v \in B | FA_v(s_v, I_v(\omega)) \neq \varnothing\}$$

Иными словами, фронт активной волны в заданном состоянии — множество блоков, которые могут быть запущены на следующем шаге.

**Определение 8.** *Семейство $\pi_v^t \in E_v,\ FA_v = (\cdot, \cdot, E_v, \cdot)$ показывает текущий переход, который совершает блок $v$ в момент времени $t$.*

Заметим, что значение $\pi_v^t$ имеет смысл только для работающих на шаге $t$ блоков.

**Определение 9.** *Условием излучения назовем $\zeta(\omega^t, \omega^O) \Leftrightarrow (\omega^O \cap \omega^t = \varnothing)$. Также определим семейства $\zeta_v^t$:*

$$\zeta^t = \{v \in B_c | zeta(\omega^t, \omega_v^O)\}$$

$$\omega_v^O = \{e \in E_c | e = (v, v^O, \cdot, \cdot), v^O \in O_v^t\},\ (\cdot, O_v^t, \cdot, \cdot) \in \pi_v^t$$

Условие излучение звучит довольно просто: блок может излучить, если все ребра соединенные с выходным набором портов свободны.

Пусть $\phi^t \subseteq \vartheta^t$ — множество блоков, которые закончат работу после шага $t$. Для любого времени это множество может быть произвольным подмножеством $\vartheta^t$. Однако для любого семейства этих множеств должно выполняться условие конечной работы блока: $\forall t \forall b \in \theta^t \exists \tau > t : b \in \phi^\tau$.

Так же для удобства введем обозначение для любого множества $A_c$ блоков из $B_c$ — $A_c(v) \Leftrightarrow v \in A_c, v \in B_c$.

Теперь выпишем алгоритм преобразования потока данных в виде условий на переходы каждого блока из одного множества в другое. В каждый момент времени блок $v$ может находиться в одном из трех состояний (множеств): свободен ($v \in \xi^t$), работает($v \in \theta^t$), заблокирован($v \in \chi^t$). Время работы блоков задается $\phi^t \subseteq \vartheta^t$ — множеством блоков, которые закончат работу после шага $t$. Для любого времени это множество может быть произвольным подмножеством $\vartheta^t$. Однако для любого семейства этих множеств должно выполняться условие конечной работы блока: $\forall t \forall b \in \theta^t \exists \tau > t : b \in \phi^\tau$. Эти состояние взяты из реальных систем запуска потоков данных и служат для интуитивного понимания

модели. В дальнейшем, однако, мы избавимся от этих состояний. Также введем семейства — переход автомата Мили блока, который совершает блок в текущий момент.

$$\xi^t(v) \wedge \neg\Psi_c(\omega^t, s^t)(v) \quad \Rightarrow \quad \begin{cases} \xi^{t+1}(v), \\ s_v^{t+1} = s_v^t, \\ \omega_{I,v}^t = \omega_{O,v}^t = \varnothing \end{cases} \tag{1}$$

$$\xi^t(v) \wedge \Psi_c(\omega^t, s^t)(v) \wedge v \neq \text{STOCK} \quad \Rightarrow \quad \begin{cases} \theta^{t+1}(v), \\ (\cdot, \cdot, I_v^t) \in FA_v(s_v^t, I_v(\omega^t)), \\ \pi_v^{t+1} \in \{(s_v^t, I_v^t, \cdot, s_v^{t+1}) \in E_v\}, \\ \omega_{I,v}^t \in \Delta_{I_v^t, v}(\omega^t), \\ \omega_{O,v}^t = \varnothing \end{cases} \tag{2}$$

$$\theta^t(v) \wedge \neg\phi^t(v) \quad \Rightarrow \quad \begin{cases} \theta^{t+1}(v), \\ s_v^{t+1} = s_v^t, \\ \pi_v^{t+1} = \pi_v^t, \\ \omega_{I,v}^t = \omega_{O,v}^t = \varnothing \end{cases} \tag{3}$$

$$\theta^t(v) \wedge \phi^t(v) \wedge \zeta^t(v) \quad \Rightarrow \quad \begin{cases} \xi^{t+1}(v), \\ (s_v^t, \cdot, O_v^t, s_v^{t+1}) = \pi_v^t, \\ \omega_{O,v}^t = \Delta_{O_v^t, v}, \\ \omega_{I,v}^t = \varnothing \end{cases} \tag{4}$$

$$\theta^t(v) \wedge \phi^t(v) \wedge \neg\zeta^t(v) \quad \Rightarrow \quad \begin{cases} \chi^{t+1}(v), \\ \pi_v^{t+1} = \pi_v^t, \\ \omega_{O,v}^t = \omega_{I,v}^t = \varnothing \end{cases} \tag{5}$$

$$\chi^t(v) \wedge \zeta^t(v) \quad \Rightarrow \quad \begin{cases} \xi^{t+1}(v), \\ (s_v^t, \cdot, O_v^t, s_v^{t+1}) = \pi_v^t, \\ \omega_{O,v}^t = \Delta_{O_v^t, v}, \\ \omega_{I,v}^t = \varnothing \end{cases} \tag{6}$$

$$\chi^t(v) \wedge \neg\zeta^t(v) \quad \Rightarrow \quad \begin{cases} \chi^{t+1}(v), \\ s_v^{t+1} = s_v^t, \\ \pi_v^{t+1} = \pi_v^t, \\ \omega_{O,v}^t = \omega_{I,v}^t = \varnothing \end{cases} \tag{7}$$

$$\omega^{t+1} = \omega^t \setminus (\cup_{v \in B_c} \omega_{I,v}^t) \cup (\cup_{v \in B_c} \omega_{O,v}^t), \tag{8}$$

$$s^{t+1} = (s_{v_1}^t, \ldots, s_{v_m}^t) \tag{9}$$

Работу каждого блока можно схематично изобразить в виде автомата, см. рисунок 3.

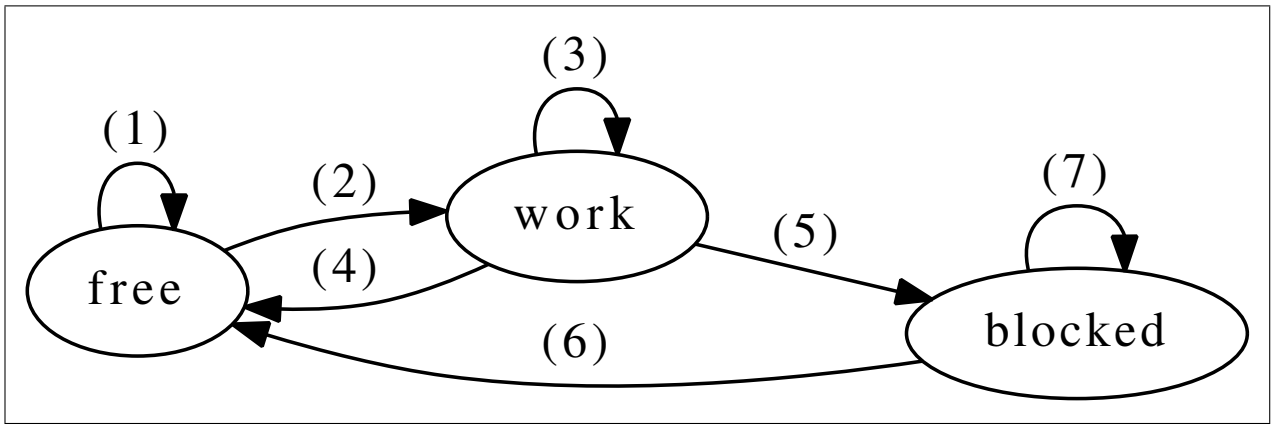



Рис. 3: Схема работы блока

Поясним эти правила. Каждое правило (1) — (7) описывает условие перехода в конечном автомате на рисунке 3.

Правила (1) и (2) говорят о поведении свободного блока — если блок может запуститься из его текущего состояния $s_v^t$ (то есть хватает данных на портах для какого-либо перехода), то он поглощает данные $\omega_{I,v}^t$ и начинает некий переход $\pi_v^{t+1}$ из текущего состояния, иначе остается свободным дальше.

Тройка правил (3), (4) и (5) описывают поведение работающего блока $v$. Если множество $\phi^t$ не содержит рассматриваемый блок, он продолжает работу. Иначе, возможны два варианта: либо блок может завершиться и испустить данные $\zeta^t(v)$ (то есть ребра $\Delta_{O_v^t,v}$, связанные с испускаемыми портами $O_v^t$, свободны), то блок благополучно завершает работу $\xi^{t+1}(v)$ и испускает данные $(\omega_{O,v}^t)$, либо блок переходит в состояние блокировки до момента, когда требуемые ребра освободятся, которое описывается правилами (6) и (7).

Заметим, что мы специально различаем состояния блокировки и работы, хотя можно было бы считать, что заблокированный блок просто работает дополнительное время. Состояние блокировки отличается тем, что блок в этом состоянии не требует вычислительных ресурсов, что будет важно в дальнейшем.

Заметим, что в правилах поведения потока данных сразу видны все неопределенности, о которых говорилось ранее — перед запуском блока, определяется переход, который будет совершать блок, вместе с набором поглощаемых портов и ребра с которых эти данные будут поглощены. Подробнее мы будем рассматривать эти неопределенности далее.

### 1.4.2 Начало и завершение работы потока данных

Если рассматривать представление потока данных в виде графа, то начало и завершение потока данных связанно с дополнительными или «фиктивными» блоками SOURCE и STOCK: их задачи сформировать начальную и поглотить конечную активные волны, соответственно. Исходя из правил, можно

записать начальные условия:

$$\omega^0 = \varnothing, \tag{10}$$

$$s_v^0 = s_{0,v}, \forall v \in B_c, \tag{11}$$

$$\theta^0 = \{\text{SOURCE}\}, \tag{12}$$

$$\xi^0 = B_c \setminus \{\text{SOURCE}\}, \tag{13}$$

$$\chi^0 = \varnothing, \tag{14}$$

$$\omega^1 \neq \varnothing \tag{15}$$

Формирование начальной волны заложено в принципе работы блока SOURCE. Так как блоки SOURCE и STOCK являются вспомогательными, то нет смысла рассматривать их автоматы Мили, так как автомат для блока SOURCE не может иметь ни одного перехода из-за отсутствия входных портов, а автомат для блока STOCK будет создавать дополнительные неопределенности при анализе потока данных. Будем считать, что работа этих блоков определяется следующим образом: SOURCE на первом шаге испускает некий непустой набор входных данных — начальную активную волну; блок STOCK поглощает все данные на его портах, если никакие другие блоки больше не могут запуститься. Именно запуск блока STOCK мы будем считать завершением работы всего потока данных, иными словами, поток данных завершает работу на шаге $t = T_{end}$, если:

$$\theta^{T_{end}} = \varnothing \wedge \Psi_c(\omega^{T_end}, s^{T_end}) = \{\text{STOCK}\} \tag{16}$$

Завершение работы потока данных связанно с понятием траектории запуска потока данных и более подробно будет рассмотрено ниже.

# 2 Анализ модели

## 2.1 Построение конечного автомата Мили по программе блока

Введенная выше модель оперирует с упрощенными описаниями алгоритмов блоков — каждый алгоритм заменяется на описывающий его качественное поведение автомат Мили. Каждое состояние конечного автомата соответствует группам внутренних состояний алгоритма блока. Разбиение на группы можно производить с большой свободой. Сразу же заметим, для дальнейшего анализа будет полезно выделить особое состояние конечного автомата Мили для блока — начальное состояние (далее будет обозначаться словом INITIAL), соответствующее начальным значениям внутренних переменных. Это состояние обладает значительной особенностью, а именно, блок находящийся в нем, фактически не занимает ресурсов памяти вычислительного узла, поэтому может быть просто перенесен на другой вычислительный узел без дополнительных затрат на копирование внутреннего состояния с одного вычислительного узла на другой. Процедуры перемещения состояния алгоритма блока на другой вычислительный узел играют важную роль в управлении запуском потока данных в распределенной среде (более подробно запуск потока данных в распределенной среде будет рассмотрен ниже), так как ожидающий данных блок будет заблокирован (или по, крайней мере, может быть дополнительно ограничен в вычислительных ресурсах), если на вычислительном узле на момент поступления данных первому блоку выполняется алгоритм другого блока, но может быть инициализирован на свободном (если таковые имеются) вычислительном узле, если блок находится в начальном состоянии.

Построение автомата Мили для алгоритма блока — неоднозначная задача, требующая понимания качественного поведения алгоритма, а значит участия человека. В данной работе подробно не рассмат-

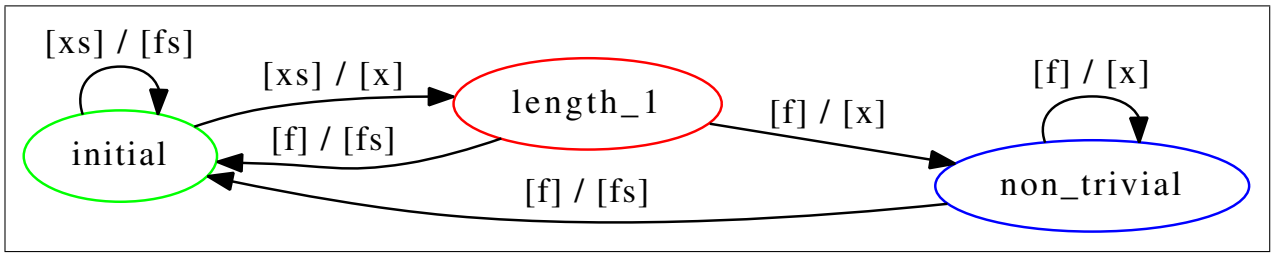



Рис. 4: Избыточный конечный автомат Мили для шаблона блока *For-Loop*

риваются алгоритмы (которые, по мнению автора, будут бесполезны на практике в силу огромных вычислительных затрат) и методики построения автоматов Мили для алгоритмов блоков. Ограничимся лишь рассмотрение показательного примера с шаблоном блока *For-Loop*. В зависимости от реализации, блок *For-Loop* принимает на вход данные о количестве необходимых операций, для определенности будем рассматривать шаблон блока, реализующий аналог функции *map* популярной в функциональных языках программирования, которая поэлементно применяет переданную ей в качестве аргумента функцию к списку значений и выдает список результатов. Типичный пример составного блока с блоком шаблона $For-Loop$ показан на рисунке 2a (сам блок шаблона *For-Loop* именуется *map*), а автомат Мили шаблона блока *For-Loop* показан на рисунке 5b. Сам блок шаблона *For-Loop* требует внешнего блока-функции, который подключен к портам $x$ и $f$. Между вызовами (то есть испусканием очередного элемента списка) блока-функции, блок шаблона *For-Loop* имеет внутреннее состояние, в котором хранит, как минимум, не обработанную часть списка и частичный результат преобразования. Поэтому в конечном автомате Мили шаблона блока *For-Loop* имеются два состояния: *initial* — начальное состояние ожидания данных, *non_ trivial* — состояние, в котором результирующий список находится в стадии формирования.

Фактически, для любого алгоритма можно построить автомат Мили с не более, чем двумя состояниями: начальным (или тривиальным) и не тривиальным (с отличным от начального внутренним состоянием) состояниями, однако, в некоторых случаях автомат Мили будет вносить дополнительную неопределенность в поведение блока, не связанную с неопределенностью входных данных при анализе. Возможен и обратный случай — в предыдущем примере с шаблоном блока *For-Loop* можно ввести отдельное состояние, отвечающее за списки длины 1. В данном случае автомат Мили будет выглядеть, как показано на рисунке 4, дополнительное состояние именуется *length_ 1*. Это дополнительное состояние, конечно, несет новую информацию о текущем состоянии блока, но она абсолютно бесполезно для анализа схемы потока данных.

## 2.2 Эквивалентность траекторий запуска потока данных

Рассмотрим подробнее правила работы потока данных. В правилах помимо неопределенностей в выборе вариантов поведения, присутствует множество $\phi^t$, задаваемое извне. Фактически это множество определяет время работы каждого запущенного блока и введено для возможности описания любой возможной конфигурации времен работы алгоритмов блоков. Однако при анализе потока данных эти времена неизвестны и здесь не будут рассматриваться их оценки. Для анализа большее значения играет не сами времена работы блоков, а зависимости типа "запуск блока $A$ был вызван завершением работы блоков $B_1, B_2, \ldots, B_l$". В дальнейшем, для выяснения всевозможных поведений потока данных в терминах этих зависимостей, мы будем эмитировать запуск потока данных по правилам (1)-(9), и, в частности, положим времена работы всех блоков равными единице. Модельное время в данном случае отражает лишь один из возможных случаев запуска потока данных и служит только для удобства

понимания. Вначале определим понятие траектории запуска потока данных и введем на множестве траекторий запуска отношение эквивалентности.

**Определение 10.** *Траекторию запуска потока данных* $\Lambda = \{\Lambda^t\}_{t=0}^{\infty}$ *мы определим как семейство всех множеств упомянутых в правилах работы потока данных* (1)-(9) *и описывающих состояние потока данных, то есть* $\Lambda^t = (\omega^t, \{\omega^t_{I,v} | v \in B_c\}, \{\omega^t_{O,v} | v \in B_c\}, \pi^t, s^t, \xi^t, \theta^t, \chi^t$.

Введем также определения конечной и успешной траекторий и времени работы потока данных на этой траектории.

**Определение 11.** *Траектория запуска потока данных* $\Lambda = \{\Lambda^t\}_{t=0}^{\infty}$ *называется конечной, если* $\exists T_{end} : \forall t \geq T_{end} : \theta^t = \varnothing$. *Минимальное* $T_{end}$ *называется временем работы потока данных.*

**Определение 12.** *Траектория запуска потока данных* $\Lambda = \{\Lambda^t\}_{t=0}^{\infty}$ *называется успешной, если* $\exists T_{end} : \forall t \geq T_{end} : \xi^t = B_c \wedge \omega^t = \varnothing$.

Понятно, что минимальные $T_{end}$ из определений 11 и 12 совпадают (если оба существуют), и более того, равны (с точностью до времени работы блока STOCK, которое, для удобства положим равным 0) $T_{end}$ в (16), так как при отсутствии работающих блоков фронт активной волны $\omega^t$ не может меняться (в силу правила работы потока данных (8)), а значит правило (6) не может быть выполнено для заблокированных блоков, так как оно полностью определяется фронтом активной волны. В дальнейшем мы будем предполагать, что все траектории рассматриваемого потока данных являются конечными.

Множества $\pi^t_v$, $\omega^t_{I,v}$ и $\omega^t_{O,v}$ полностью описывают отношение эквивалентности между траекториями.

**Определение 13.** *Для каждой конечной траектории запуска множества* $\pi^t_v$, $I^t_v$ *порождают граф причинности* $G = (S, E)$. *Для каждого блока пронумеруем все времена его запуска* $t^n_v$. *Тогда* $G$ *можно определить как:*

- $S = \{(v, n, \pi_v^{t_v^n})\}$,
- *из* $s_1 = (v, \cdot, \pi_1)$ *в* $s_2 = (v, \cdot, \pi_2)$ *есть ребро тогда и только тогда, когда часть поглощенной* $\omega^t_{I,v}$ *в результате перехода* $\pi_2$ *была испущена в результате перехода* $\pi_1$*; при этом ребро дополнительно помечается множеством соответствующих входных портов блока* $v$.

Отображение $G_\Lambda = G(\Lambda)$ индуцирует классы эквивалентности траекторий запуска потока данных, иными словами $\Lambda_1 \sim \Lambda_2 \Leftrightarrow G(\Lambda_1) = G(\Lambda_2)$. Отметим, что данное отношение эквивалентности имеет смысл только для успешных траекторий, так как в нем не учитываются остаточные активные волны и заблокированные блоки. Наличие не успешных траекторий потока данных мы будем рассматривать как ошибку проектирования схемы потока данных. Отметим, что это правило полностью соответствует практике.

Каждый граф причинности качественно описывает запуск потока данных.

**Пример 2.** *Рассмотрим один показательный пример нахождения графов причинности для потока данных, содержащего цикл (схема потока данных изображена на рисунке 2а). Для простоты, будем считать, что автомат Мили блока* $f$ *состоит из одного состояния и одного перехода* $\pi_f = (INITIAL, \{x\}, \{f\}, INITIAL)$. *Наличие ребра из* $s_1$ *в* $s_2$ *в графе причинности будем обозначать как* $s_1 \rightsquigarrow s_2$.

*Начальная вершина всех графов причинности —* $s_0 = (SOURCE, 0, \pi_{SOURCE})$. *Далее возможны варианты: либо начальная вершина вызовет переход* $\pi_{MAP,1} = (INITIAL, \{xs\}, \{fs\}, INITIAL)$ *(то*

*есть переход по пустому списку данных), либо* $\pi_{MAP,2} = (INITIAL, \{xs\}, \{x\}, NON\text{-}TRIVIAL)$. *Последний в свою очередь вызовет переход* $\pi_f$, *который, в свою очередь, может вызвать переходы* $\pi_{MAP,3} = (NON\text{-}TRIVIAL, \{f\}, \{x\}, NON\text{-}TRIVIAL)$ *и* $\pi_{MAP,4} = (NON\text{-}TRIVIAL, \{f\}, \{xs\}, INITIAL)$, *и так далее. Конечная вершина графа —* $s_{end} = (STOCK, T_end, \pi_{STOCK})$. *Получаем семейство графов причинности:*

$$\begin{aligned}
s_0 &\rightsquigarrow (MAP, t_1, \pi_{MAP,1}) \rightsquigarrow s_{end}, \\
s_0 &\rightsquigarrow (MAP, t_1, \pi_{MAP,2}) \rightsquigarrow (f, t_2, \pi_f) \rightsquigarrow (MAP, t_3, \pi_{MAP,4}) \rightsquigarrow s_{end}, \\
s_0 &\rightsquigarrow (MAP, t_1, \pi_{MAP,2}) \rightsquigarrow (f, t_2, \pi_f) \rightsquigarrow (MAP, t_3, \pi_{MAP,3}) \rightsquigarrow \\
&\rightsquigarrow (f, t_4, \pi_f) \rightsquigarrow \cdots \rightsquigarrow (MAP, t_n, \pi_{MAP,4}) \rightsquigarrow s_{end}
\end{aligned}$$

Этот пример демонстрирует необходимость следующего предположения: *будем считать, что любая циклическая зависимость конечна (то есть не будем рассматривать не конечные траектории, возникающие из-за этих зависимостей).*

## 2.3 Неопределенности в запуске потока данных

Как было упомянуто выше, в анализе схемы потока данных неизбежно возникают два рода неопределенностей: первый вызван отсутствием знания входных данных при анализе схемы, неопределенности второго рода связаны с самой схемой потока данных, которая может допускать такие неоднозначности как, например, несколько вариантов поглощения данных. Неоднозначности первого рода неизбежны, и поэтому следует обрабатывать все варианты порожденные ими. Все эти неоднозначности второго рода возникают как следствие состояний гонки (race condition), которых традиционно стараются избегать, так как результат выполнения всего потока данных становиться зависим от времен работы блоков, которые, в свою очередь, могут зависеть от данных и параметров вычислительных узлов, что говорит скорее о неправильном построении схемы потока данных.

Сформулируем теперь критерии неопределенностей первого и второго рода.

Любые неопределенности связаны с правилом запуска потока данных (2) — правило запуска блока, что видно из правил работы потока данных, так как, только это правило содержит включения, вместо строгих равенств.

**Определение 14.** *Будем говорить, что состояние* $s$ *автомата Мили* $FA_v = (S_v, \cdot, \cdot, \cdot, E_v)$ *блока* $v$ *допускает неопределенность первого рода, тогда и только тогда, когда:* $\exists I_v : |\{e \in E_v | e = (s, I_v, \cdot, \cdot)\}| > 1$, *или иными словами, существует набор входных портов* $I_v$, *такой что, из состояния* $s$ *возможны несколько переходов по входным портам* $I_v$.

Например, в автомате Мили шаблона блока *For-Loop* (рисунок 5b) каждое состояние допускает неопределенность первого рода. Причины возникновения этих неопределенностей были описаны выше.

**Определение 15.** *Блок* $v \in B_c$ *в состоянии* $\Lambda^t$ *траектории запуска потока данных на шаге* $t$ *допускает неопределенность второго рода, если:* $|\omega^t_{I,v}| > 1$, *где* $\omega^t_{I,v}$ *определена в правиле* (2) *и содержится в кортеже* $\Lambda^t$.

**Определение 16.** ***Корректным потоком данных*** *будем называть поток данных, любая возможная траектория которого не допускает неопределенностей второго рода.*

Заметим, что по автомату Мили можно сразу предсказать возможные неопределенности второго рода, например, если автомат Мили содержит переходы $\pi = (s, I, \cdot, \cdot)$ и $\pi' = (s, I', \cdot, \cdot)$, где $I \subseteq I'$: в этом

случае переход по $\pi'$ автоматически означает неопределенность второго рода, соответственно, наличие таких переходов автоматически означает неправильность построения автомата Мили или алгоритма блока.

Введем вспомогательное определение набора последовательностей времен работы блоков.

**Определение 17.** *Зафиксируем набор последовательностей времен работы* $\tau = \{\{\tau_v^n\}_{n=0}^{\infty} | v \in B_c\}$. *Траектория* $\Lambda$ *согласуется с набором последовательностей времен запуска, если время работы* $n$*-го запуска блока* $v$ $t_v^n = \tau_v^n$.

**Утверждение 1.** *В работе потока данных не существует других неопределенностей, кроме неопределенностей первого и второго рода, то есть, если зафиксировать набор последовательностей времен работы блоков* $\tau$*, то любое ветвление префиксного дерева траекторий, согласующихся с* $\tau$*, описывается только неопределенностями первого и второго рода.*

*Доказательство.* Доказательство напрямую следует из факта, что любые различия в траекториях запуска потока данных при фиксированных временах работы блоков определяются правилом (2) работы потока данных. □

Среди неопределенностей второго рода следует выделить важный случай, уже упоминавшийся выше — состояние гонки.

**Определение 18.** *Поток данных допускает состояние гонки, если существует некая последовательность времен запуска* $\tau$ *и класс эквивалентности* $\mathbb{K}$ *траекторий запуска потока данных, такие что траектории удовлетворяющие* $\tau$ *не лежат в классе* $\mathbb{K}$.

Поясним это определение. Из определения сразу же следует, что в отсутствии состояния гонки, каждый класс эквивалентности для любого набора последовательностей времен запуска содержит траекторию, согласующуюся с ней, что означает то, что результат не зависит от времен работы блоков (что соответствует классическим определениям).

## 2.4 Волновой алгоритм

Решение задачи нахождения классов эквивалентности траекторий запуска позволяет существенно упростить анализ потока данных, так как переход к графам причинности позволяет избавиться от модельного времени для корректных потоков данных.

Однако, даже для корректного потока данных, классов эквивалентности может быть бесконечное количество. Это утверждение было проиллюстрировано в примере 2. Одним из возможных решений этой проблемы — нахождение циклов и их замена на соответствующий составной блок. Однако, это не всегда удается, например, из-за того, что «тело» цикла может иметь связи с самим блоком цикла. Однако, если замена циклов удалась, то поток данных будет преобразован к ацикличному графу, работу которого заметно проще анализировать. В данной работе рассматривается иной метод избавления от циклических зависимостей путем неявного построения «внутреннего» автомата Мили, который можно известными из теории синтаксического анализа алгоритмами преобразовать к «внешнему» автомату Мили, который можно использовать для представления потока данных как единого составного блока в иерархическом потоке данных.

### 2.4.1 Волновой алгоритм для ацикличного корректного потока данных

Вначале рассмотрим самый простой случай — анализ ацикличного корректного потока потока данных. Задача волнового алгоритма перебрать все возможные классы эквивалентности траекторий запуска

потока данных, которую он выполняет путем перебора по единичным траекториям из каждого класса. Вначале докажем один простой факт, из которого вместе с 1 частично следует корректность алгоритма 1.

**Утверждение 2.** *Классы эквивалентности траекторий запуска ацикличного корректного потока данных образуют конечное множество.*

*Доказательство.* Докажем от противного. Если множество классов эквивалентности не конечно, то для любого $N \in \mathbb{N}$ существует блок в этом потоке данных и граф причинности, в котором этот блок был запущен более $N$ раз, так как количество блоков в любом потоке данных конечно. Так как поток данных по условию ацикличен, то любая волна данных не может вернуться к входным портам породившим ее блокам, а в силу конечности начальной волны любой блок будет запущен заведомо не более, чем $|E_c|$ раз, где $E_c$ — множество ребер рассматриваемого потока данных. Противоречие. □

Рассмотрим теперь подробно работу алгоритма 1. В строках 2-13 происходит формирование множества наборов возможных переходов. В эти наборы попадают переходы, которые могут запуститься ($\pi \in FA(s_v, I_v(\omega))$) и успешно излучить данные ($\zeta(\omega, \sigma)$). В строках 14-20 записано условие окончание работы потока данных в соответствии с определением 12. Заметим, что выполнение строки 18 означает, что траектория не является успешной, так как существует остаточная активная волна, которая не способна запустить ни одного блока. Выполнение строки 16 означает, что на вход подан завершившийся поток данных — функция возвращает граф уже построенный граф причинности. Далее в строках 22-41 происходить перебор всевозможных сочетаний переходов блоков, функция Samples (код которой не представляет интереса) должна вернуть эти всевозможные сочетания. В строках 34-37 происходит дополнения неполного графа причинности для текущего случая. Заметим, что выполнение строки 32 означает более одного варианта поглощения, что является неопределенностью второго рода и рассматривается как ошибка. Докажем корректность алгоритма.

**Утверждение 3.** *Функция* ***Wave*** *из алгоритма 1 при запуске* $Wave(c, (S_0, \omega_0), (\varnothing, \varnothing))$ *на корректном ацикличном потоке данных с возвращает множество всех возможных графов причинности.*

*Доказательство.* Для доказательства внимательно рассмотрим возвращаемые значения алгоритма. Вначале рассмотрим случай незавершенного состояния потока данных (то есть допускающего запуск блоков). Рассмотрим строки 14-20. Так как по условию поток данных корректный, а значит в частности не содержит состояний гонки, то запуск блока не в момент выполнения условия запуска 2, а в момент выполнения условий запуска и излучения эквивалентен изменению времени работы блока до момента выполнения условия излучения, и не влияет на граф причинности обрабатываемых классов эквивалентности. А значит перебор возможных переходов из данного состояния происходит корректно. В строке 40 происходит сбор результатов стартов алгоритма из новых начальных условий с дополненными графами причинности. В силу утверждения 2 и вышесказанного все рекурсивные вызовы функции закончатся на строке 16, вернув один из графов причинности завершенного потока данных. Так как любой поток данных стартует из одного состояния $((S_0, \omega_0))$ и на каждом шаге рассматриваются все возможные варианты запуска блоков (также можно показать, что все графы причинности различны), то вызов функции с начальными условиями вернет множество всех графов причинности, перебрав тем самым все классы эквивалентности. □

**Algorithm 1** Волновой алгоритм для случая ацикличного корректного потока данных

1: **function** WAVE($c = (B_c, E_c), S_c = (S_B, \omega), g_R = (V, E)$)
2: $\Pi \leftarrow$ empty
3: **for all** $v \in B_c, v \neq$ STOCK **do**
4: $\Pi_v \leftarrow \varnothing$
5: $s_v \leftarrow \text{StateOfBlock}(S_B, v)$
6: **for all** $\pi \in FA_v(s_v, I_v(\omega))$ **do**
7: $(\sigma, \lambda', q) \leftarrow \pi$
8: **if** $\zeta(\omega, \sigma)$ **then**
9: $\Pi_v \leftarrow \Pi_v \cup \{\pi\}$
10: **end if**
11: **end for**
12: $\Pi(v) \leftarrow \Pi_v$
13: **end for**

14: **if** $\Pi =$ empty **then**
15: **if** $\Delta_{I,\text{STOCK}}(\omega) = \{\omega\}$ **then**
16: **return** $\{g_R\}$
17: **else**
18: **error** "Ошибка, поток данных не является корректным"
19: **end if**
20: **end if**

21: $G_R \leftarrow \varnothing$
22: **for all** $\pi \in \text{Samples}(c, \Pi)$ **do**
23: $(V', E') \leftarrow g_R$
24: $(S'_B, \omega') \leftarrow S_c$
25: **for all** $v \in B_c, v \neq$ STOCK **do**
26: **if** $\pi(v) \neq$ empty **then**
27: $(\sigma, \lambda', q) \leftarrow \pi(v)$
28: $\Delta \leftarrow \Delta_{I,v}(v)$
29: **if** $|\Delta| = 1$ **then**
30: $\delta \leftarrow \Delta_1$
31: **else**
32: **error** "Ошибка, поток данных не является корректным"
33: **end if**
34: $V' \leftarrow V' \cup \text{Reasons}_c(S_c, \lambda')$
35: $E' \leftarrow E' \cup \{(r, \pi_r, v, \pi(v)) | (r, \pi_r) \in \text{Reasons}_c(\lambda')\}$
36: $\omega' \leftarrow (\omega' \setminus \delta) \cup \sigma$
37: $S'_B \leftarrow \text{SetStateOfBlock}(S'_B, v, q)$
38: **end if**
39: **end for**
40: $G_R \leftarrow G_R \cup \text{Wave}(c, (S'_B, \omega'), (V', E'))$
41: **end for**

42: **return** $W$
43: **end function**

### 2.4.2 Волновой алгоритм для корректного потока данных

Вернемся теперь к рассмотрению общего случая потоков данных, а именно к потокам с возможным наличием циклов. Как уже было показано множество графов причинности потока данных содержащего циклические зависимости может и не быть конечным. Заметим, что не всякая циклическая зависимость в графе потока данных эквивалентна циклу в классическом понимании. Чтобы убедиться в этом, достаточно изменить поток данных из примера 2a, убрав ребро (NON-TRIVIAL, NON-TRIVIAL) из автомата Мили шаблона блока map. Поэтому будет говорить, что в потоке данных присутствует цикл, если в нем существует циклическая зависимость, порождающая бесконечное множество графов причинности.

Не сложно убедиться, что подграфы графов причинности, соответствующие циклам (будем называть такой подграф подграфом цикла) в потоке данных, образуют регулярные цепочки. Заметим, что для анализа графов причинности достаточно указать графы причинности соответствующие каждой полной итерации. Это означает, что существует множество полных итераций — последовательностей переходов, которые вернут подграф цикла в исходное состояние. На обнаружении возвращений в исходное состояние подграфа цикла основана идея модификации волнового алгоритма 1. Для этого мы будем строить «внутренний» автомат всего потока данных.

**Определение 19.** ***Внутренним автоматом*** *потока данных будем называть граф* $FA_c = (S, E)$*, где:*

- $S = \Sigma \times 2^{E_c}$ *— множество состояний потока данных (то же, что и в определении потока данных);*
- $E \subseteq S \times \Pi \times S$ *— множество ребер «внутреннего» автомата Мили потока данных, где* $\Pi = \cup_{v \in B_c} \Pi_v$*,* $\Pi_v$ *— множество переходов автомата Мили блока* $v \in B_c$*.*

В отличии от алгоритма 1 в данном алгоритме мы будем делать по одному запуску блока за итерацию, причем отдавать предпочтение испущенным в результате предыдущего запуска данным.

Начальный запуск осуществляется как $\text{Wave}(c, S_0, \text{push}(\text{empty\_stack}, \omega_0))$. Вспомогательные функции push, delete и pop являются стандартными функциями для работы со стеком. Заметим, что алгоритм заведомо завершится за конечное количество шагов, так как единственный рекурсивный вызов в строке 19 осуществляется только при отсутствии рассматриваемой точки «внутреннего» автомата в множестве $S$, которое заведомо конечно. Доказательство корректности алгоритма повторяет доказательство корректности алгоритма 1, с той лишь разницей, что требуется заметить, что одиночный запуск блока за вызов функции Wave при отсутствии состояний гонки (что гарантируется корректностью потока данных) дает корректный автомат Мили.

Поясним теперь необходимость представления активной волны в виде стека. Для этого достаточно рассмотреть поток данных, в котором дополнительная ветка расположена параллельно циклу. Так как кортеж $S_B$ содержит состояния всех блоков, то для избежания лишних итераций алгоритма, благодаря стеку, выполнение параллельной ветки «остановится» на время выполнения расчета цикла.

Заметим, что полученный «внутренний» автомат позволяет довольно просто генерировать графы причинности и его можно неформально рассматривать как конечный автомат для регулярного «языка» графов причинности.

## 2.5 Анализ графов причинности

Переход к графам причинности является главным шагом к анализу запуска потока данных в распределенных средах. Фактически граф причинности полностью описывает стратегию запуска потока

**Algorithm 2** Волновой алгоритм для случая корректного потока данных

1: $S \leftarrow \varnothing$
2: $E \leftarrow \varnothing$
3: **function** WAVE($c = (B_c, E_c), S_B, Q_W$)
4: $S \leftarrow S \cup \{(S_B, Q_W)\}$
5: $\omega' \leftarrow \varnothing$
6: $Q'_W \leftarrow Q_W$
7: **while** $\forall v \in B_c : FA_v(s_v, \omega') = \varnothing \vee \neg\zeta(\sigma, \omega)$ **do**
8: $\omega' \leftarrow \omega' \cup \{\text{pop}(Q'_W)\}$
9: **end while**

10: **if** $|\{v \in B_c | FA_v(s_v, \omega') \neq \varnothing\}| = 1$ **then**
11: $v \leftarrow \{v \in B_c | FA_v(s_v, \omega') \neq \varnothing\}_1$
12: **else**
13: error "Поток данных не является корректным"
14: **end if**
15: $\Pi \leftarrow FA(stateOfBlock(S_B, v), I_v(\omega'))$
16: **for all** $\pi_v \in \Pi$ **do**
17: $(\sigma, \lambda, q) \leftarrow \pi$
18: $Q'_W \leftarrow \text{push}(\text{delete}(Q_W, \lambda), \sigma)$
19: $S'_B \leftarrow SetStateOfBlock(S_B, v, q)$
20: **if** $(S'_B, Q'_W) \notin S$ **then**
21: $S \leftarrow S \cup \{(S'_B, Q'_W)\}$
22: $E \leftarrow E \cup (S_B, Q_W, \pi_v, S'_B, Q'_W)$
23: Wave$(c, S'_B, Q'_W)$
24: **else**
25: $E \leftarrow E \cup (S_B, Q_W, \pi_v, S'_B, Q'_W)$
26: **end if**
27: **end for**
28: **end function**

данных. Все необходимые алгоритмы получения графов причинности по схеме потока данных были описаны выше. Здесь мы рассмотрим задачу нахождения наилучшего количества вычислителей для запуска потока данных в бесконечной однородной распределенной среде. Для этого достаточно рассмотреть эту задачу для одного графа причинности и объединить результаты.

Заметим, что бессмысленно в общем случае рассматривать какие-либо вероятностные меры на графах причинности. Фактически сопоставление графу причинности какой-либо вероятности означает приписывание каждым переходам автоматов Мили блоков вероятностей. Однако практически всегда качественное поведение блоков зависит от данных поступивших на вход. Для примера рассмотрим часто встречающийся на практике и уже упомянутый пример потока данных (схема потока данных изображена на рисунке 2а). Прежде всего нас будет интересовать вероятности переходов в автомате Мили блока map с шаблоном Loop. Сопоставляя вероятности переходам в автомате Мили шаблона Loop, как уже было упомянуто, мы делаем предположение о длине цикла (или длине списка значений).

$$\begin{aligned} P(\text{INITIAL} \to \text{INITIAL}) &= a; \\ P(\text{INITIAL} \to \text{NON-TRIVIAL}) &= 1-a; \\ P(\text{NON-TRIVIAL} \to \text{NON-TRIVIAL}) &= b; \\ P(\text{NON-TRIVIAL} \to \text{INITIAL}) &= 1-b; \end{aligned}$$

$$\begin{aligned} P(L=0) &= P(\text{INITIAL} \to \text{INITIAL}) = a; \\ P(L=1) &= P(\text{INITIAL} \to \text{NON-TRIVIAL} \to \text{INITIAL}) = \\ &= (1-a)(1-b); \\ P(L=2) &= (1-a)b(1-b); \\ &\dots \\ P(L=n) &= (1-a)b^{n-1}(1-b); \\ \mathbb{E}(L) &= (1-a)(1-b)\sum_{n=1}^{\infty} nb^{n-1} = \frac{1-a}{1-b} \end{aligned}$$

Количество итераций цикла $L$ напрямую определяет время работы всего потока данных: $LT_f = \frac{1-a}{1-b}T_f$, где $T_f = const$ — время одной итерации (работы блока f) для простоты положим равным константе. Из формулы 17 видно, что математическое ожидание количества итераций цикла (а значит оценку длины списка в реальных запусках) можно сделать любым, однако количество итераций в реальных задачах может варьироваться в пределах нескольких порядков.

Рассмотрим поток данных и его граф причинности $g_R$. Будем считать, что при запуске поток данных будет вести себя в соответствии с графом причинности $g_R$. Пусть также дана распределенная однородная среда бесконечного размера. Будем считать, что для выполнения одного перехода автомата Мили любого блока требуется один вычислитель. Пусть количество вычислителей, на который выполняется запуск потока данных $N$. Определим задачу оценки оптимального количества ресурсов, как задачу нахождения диапазона минимального диапазона $[N_{\min}, N_{\max}]$:

$$\forall \tau \in \mathbb{K}_{g_R}(\forall N < N_{min} T_{N,\tau} > T_{N_{\min},\tau}) \wedge (\forall N > N_{max} : T_{N,\tau} = T_{N_{\max},\tau}),$$

где $T_{N,\tau}$ — минимальное время работы потока данных на $N$ вычислителях при наборе времен работы блоков $\tau$. Иными словами, задача оценки оптимального количества ресурсов состоит в поиске таких количеств вычислителей $N_{\min}$ и $N_{\max}$, что беря количество вычислителей менее $N_{\min}$, время работы

потока данных будет гарантированно меньше, а добавление еще одного вычислителя после $N_{\max}$ не принесет результата.

Для удобства введем следующее определение.

**Определение 20.** *Множество $S_{g_R}$ будем называть* ***срезом графа причинности*** *$g_R = (V, E)$, если $S_{g_R} \subseteq V : \forall P \subseteq E —$ путь в графе $g_R : \nexists s_1, s_2 \in S_{g_R} : s_1 \xrightarrow{P} s_2$.*

Иными словами срез в один срез графа причинности могут входить только переходы, которые могут выполняться одновременно (то есть не зависят друг от друга).

**Утверждение 4.** *В любой срез любого графа причинности корректного потока данных не могут входить переходы одного и того же блока.*

*Доказательство.* Предположим обратное. Заметим, что если $\pi_{1,v}, \pi_{2,v} \in S$, то в потоке данных присутствует состояние гонки. Это напрямую следует из определения среза графа причинности, так как переходы $\pi_{1,v}$ и $\pi_{1,v}$ независимы, а значит существуют наборы времен выполнения переходов, такие, что эти переходы выполняются в разной последовательности, что и есть определение состояния гонки. Но поток данных по условию корректный. Противоречие. □

Рассмотрим нахождение $N_{\max}$. Заметим, что $N_{max}$ соответствует максимальной мощности среза графа причинности, так как фактически это максимальное количество одновременно выполняемых переходов или максимальное количество вычислителей которое можно задействовать одновременно.

Для определения $N_{\min}$ потребуется ввести дополнительные определения.

**Определение 21.** *Будем говорить, что срез графа причинности $g_R = (V, E)$ $S_2$ следует из среза $S_1$, если:*

$$S_1 \rightsquigarrow S_2 \Leftrightarrow S_1 \neq S_2 \wedge \{(e_1, e_2) | e_1 \in S_1, e_2 \in S_2\} \subseteq (V \cup \{(e, e) | e \in S_1 \cap S_2\})$$

Иными словами $S_1 \rightsquigarrow S_2$ означает, что вершины $S_1$ и $S_2$ находятся на расстоянии не более 1 шага и $S_1 \neq S_2$.

**Определение 22.** ***Эволюция графа причинности*** *— последовательность $\Delta = \{S^j\}_{j=0}^{J}$, где $S^0 = \{(\pi_{SOURCE}, 1)\}$, $S^J = \{(\pi_{STOCK}, 1)\}$ и $\forall J \geq j > 1 : S^{j-1} \rightsquigarrow S^j$.*

**Утверждение 5.** *Любой эволюции $\Delta = \{S^j\}_{j=0}^{J}$ графа причинности соответствует как минимум один набор времен работы блоков $\tau$, при котором множества работающих блоков запуска потока данных $\forall t \in [0, T_{end}] : \theta^t = S^t$.*

*Доказательство.* Построим требуемый набор времен работы блоков. Для этого достаточно положить время $n$-го перехода $\pi_v$ равным $\max\{j_1 - j_2 : \forall j \in [j_1, j_2] : (\pi_v, n) \in S_j\}$. Нетрудно показать, что такие времена работы блоков удовлетворяют условию утверждения. □

Несложно убедиться используя доказанное выше утверждение, что

$$N_{\min} = \min_{\Delta} \max_{S^j \in \Delta} |S^j|$$

Для единообразия переформулируем равенство для $N_{\max}$:

$$N_{\max} = \max_{S} |S| = \max_{\Delta} \max_{S^j \in \Delta} |S^j|$$

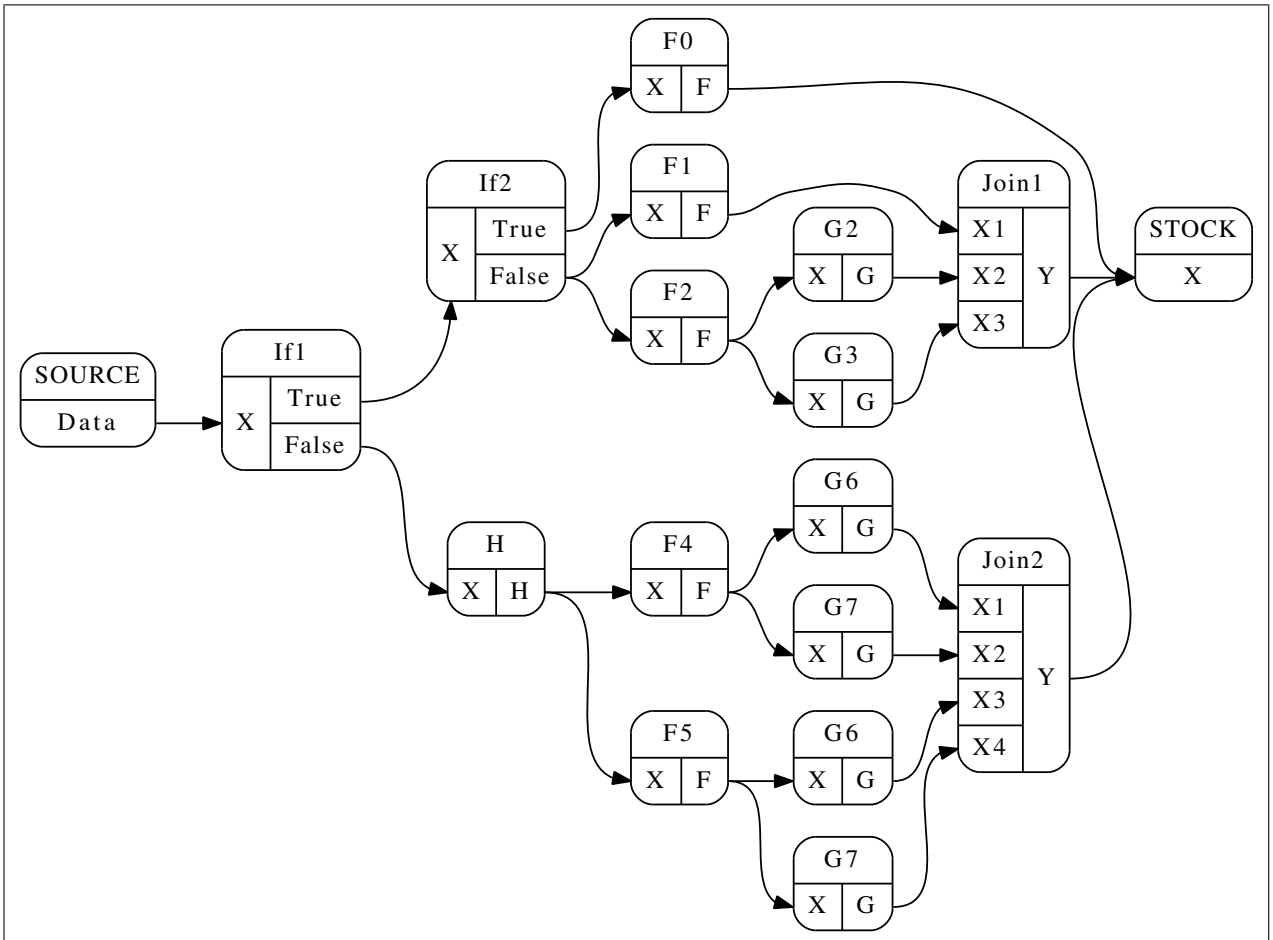
Для поиска оценок $N_{\max}$ и $N_{\min}$ существуют разнообразные модификации алгоритма HEFT [9], [10], позволяющие вычислять как точные, так и оценочные значения описанных величин.

### 2.5.1 Динамическая корректировка зарезервированных ресурсов

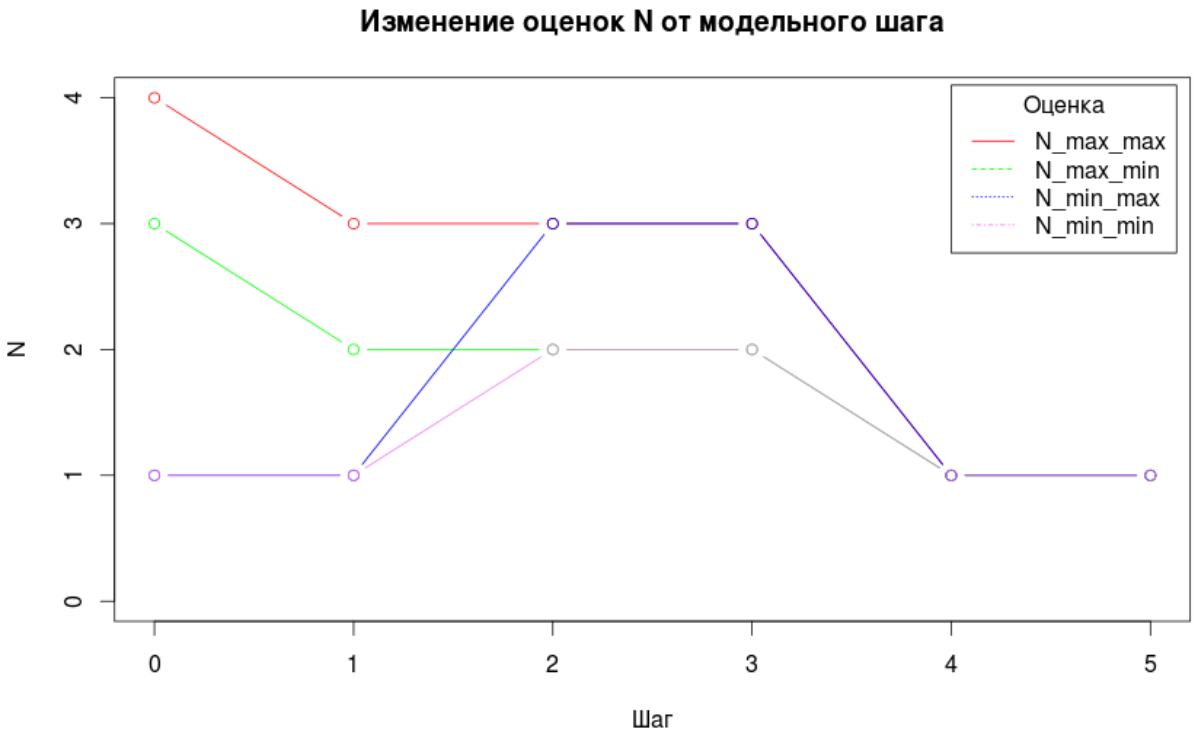
Для каждого графа причинности существуют две оценки $N_{\max}$ и $N_{\min}$ количества необходимых вычислителей. Основной проблемой является то, что точный граф причинности текущего запуска потока данных может быть неизвестен вплоть до конца работы. Однако можно заметить, что графы причинности образуют префиксное дерево (это видно уже из волнового алгоритма), поэтому возможно динамическое (runtime) уточнение множества возможных графов причинности. Заметим, что проверка, является ли граф причинности подграфом другого, осуществляется за линейное время, так как между вершинами графов причинности существует единственное соответствие. Построение же префиксного дерева неявно осуществляется во время работы волнового алгоритма.

Итак, пусть дано префиксное графов причинности. Сопоставим каждой вершине $g$ этого дерева четыре величины $\hat{N}_{\min,\min}(e)$, $\hat{N}_{\min,\max}(e)$, $\hat{N}_{\max,\min}(e)$, $\hat{N}_{\max,\max}(e)$: соответственно максимум/минимум максимумов/минимумов постфиксов графов причинности с префиксом $g$. Заметим, что некоторые модификации алгоритма HEFT позволяют быстро перевычислить оценки соответствующих величин при построении/уменьшении графа причинности. Каждая оценка несет свой смысл, например, $N_{\max,\max}$ — максимальное количество вычислителей, одновременно задействованных в «худшем» случае.

Пример динамического изменения оценок приведен на рисунке 5.




(a) Пример схемы шаблона потока данных




(b) Поведение оценок в зависимости от шага по префиксному дереву графов причинности в случае переходов If1 по ветке True (шаг 1), If2 по ветке False (шаг 2)

Рис. 5: Пример динамического изменения оценок числа $N$

## 2.6 Проверка корректности потока данных

Вернемся теперь к проверки корректности потока данных. Некорректное завершение (то есть наличие не успешных траекторий) легко определяется внутри волновых алгоритмов по наличию остаточных данных или отсутствию данных для выдачи результата потока данных. Соответствующие участки алгоритма указаны в пояснениях к алгоритмам.

Вернемся теперь к проблеме поиска неопределенностей второго рода и будем рассматривать поток данных лишь при одном предположении — все его траектории успешны.

Заметим, что неопределенности при поглощении данных, уже были частично обработаны в алгоритмах 1 и 2. Однако, волновые алгоритмы корректны только при отсутствии состояний гонки в потоке данных. Для обнаружения состояний гонок мы припишем каждому ребру активной волны ее историю. Историю начальной волны будем обозначать как $W_0$. Далее введем следующие правила.

- Если блок поглощает данные по ребрам с историями $W_1$, $W_2$, ..., $W_n$, то все результирующие данные считаются *коррелирующими со всеми волнами* $W_1$, ..., $W_n$. Будем записывать историю результирующих волн как $(W_1 + W_2 + \cdots + W_n)\pi_v$, где $\pi_v$ — соответствующий переход в блоке,
- при осуществлении блоком перехода будем записывать на всех портах историю волн.

Таким образом, для любых $w_1 \in \omega^{t_1}, w_2 \in \omega^{t_2}$ можно сказать, является ли одна из них потомком другой. В случае, если $w_2$ коррелирует с $w_1$, будем писать $w_1 \rightsquigarrow w_2$. Также будем считать, что $\forall w : \varnothing \rightsquigarrow w$. Заметим, что последовательное появление данных на некотором порту можно гарантировать тогда и только тогда, когда входящие волны являются коррелирующими. Можно считать, что все волны, полученные в результате одного излучения, коррелируют, однако для простоты мы будем считать их не коррелирующими.

Заметим, что для проверки на состояние гонки достаточно выполнить две проверки:

- проверка на корреляцию с последней прошедшей по порту волной,
- проверка на возможные комбинации с не коррелирующими волнами на других портах блоков.

Первая проверка позволяет исключить неучтенные в волновом алгоритмы варианты поглощения данных по одному и тому же набору портов. Вторая проверка исключает как раз само состояние гонки — возможность осуществления отличного от записанного в истории перехода рассматриваемого блока.

Приведем более детальный алгоритм проверки на неопределенности второго рода. На вход функции проверки подается история проверяемой волны, истории на портах текущего блока. Для определенности будем считать, что волна поглощается по порту с номером 1 (соответствует истории $H_1$).

Докажем корректность алгоритма.

**Утверждение 6.** *Алгоритм проверки 3 во время работы волнового алгоритма вернет ошибку тогда и только тогда, когда поток данных допускает состояние гонки.*

*Доказательство.* Рассмотрим вызов функции CheckRace во время работы волнового алгоритма, с условием, что ни одна функция проверки не выдала ошибки.

В строках 2-4 записана проверка на возможные варианты поглощения. Прохождение двух не коррелирующих волн через один порт, означает существование наборов времен работы блоков при котором волны пройдут в обратном порядке, что и есть состояние гонки.

Предыдущее означает, что через один порт может проходить только цепочка коррелирующих волн (в случае корректного потока).

**Algorithm 3** Функция проверки состояния гонки

```
1: function CheckRace(v, W, H_1, H_2, ..., H_n)
2:     if ¬top(H_1) ⇝ W then
3:         return Error, "Обнаружено состояние гонки"
4:     end if

5:     Ω ← ∅
6:     for i = 2, ..., n do
7:         if ¬top(H_i) ⇝ W then
8:             Ω ← Ω ∪ {top(H_i)}
9:         end if
10:    end for

11:    V ← classesOfCorrelation(Ω)
12:    for all p ∈ V do
13:        for all s ∈ states(V) do
14:            if FA_v(s, p ∪ {1}) ≠ FA_v(s, p) then
15:                return Error, "Обнаружено состояние гонки"
16:            end if
17:        end for
18:    end for
19:    return "Состояние гонки не обнаружено"
20: end function
```

Далее осуществляется проверка на другой тип состояния гонки — на осуществления различных переходов в зависимости от набора входных портов. Для этого отбираются наборы портов, через которые прошли не коррелирующие между собой и с рассматриваемой волной, после чего для каждого состояния, в котором возможно появление данного набора портов, осуществляется проверка на возможность других переходов из этого состояния с учетом текущей волны (по договоренности, она находится на порту 1). Если из некого состояние возможны другие переходы, нежели без учета рассматриваемой волны, значит присутствует состояние гонки, иначе текущая волна не могла привести к неопределенности в переходах блока.

Как уже было сказано, если в поток данных допускает состояние гонки, значит существуют как минимум два класса наборов времен при котором, некая пара волн придет на входные порты блока в разном порядке, вызвав при этом, разную последовательность переходов. Алгоритм рассматривает различные комбинации волн на входных портах (с учетом их корреляции), тем самым обнаруживает состояние гонки, если она присутствует. □

### 2.6.1 Детерминированные по входу автоматы Мили

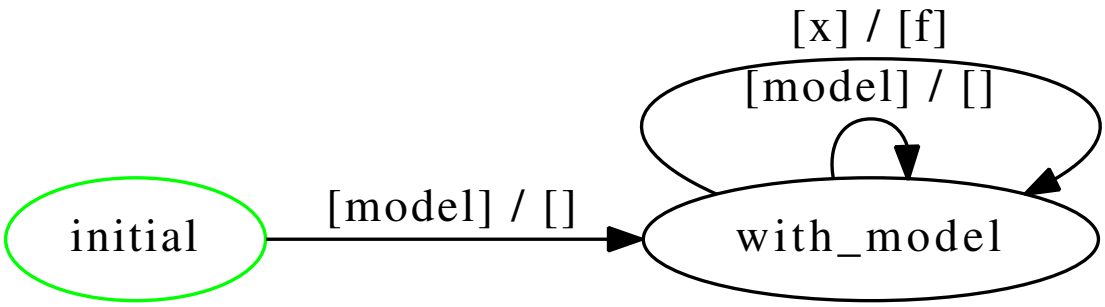
Особого рассмотрения в задаче поиска неопределенностей второго рода заслуживает специальный подкласс автоматов Мили.

**Определение 23.** ***Детерминированным по входу автоматом Мили*** *будем называть автомат Мили* $FA = (S, s_0, \Sigma, \Lambda, E)$*, для которого множество переходов подчиняется закону:*

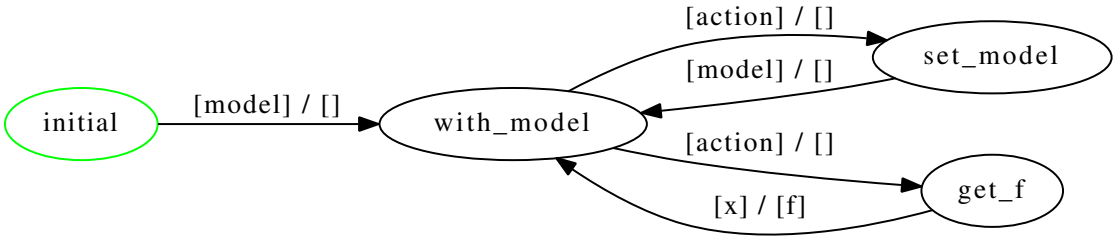
$$\forall s \in S : \forall \sigma, \sigma' : (s, \sigma, \cdot, \cdot) \in \Sigma \wedge (s, \sigma', \cdot, \cdot) \in \Sigma \Rightarrow \sigma = \sigma'$$

Иными словами, детерминированный по входу автомат Мили в любом состоянии принимает только одно входное множество.

Такой класс автоматов Мили автоматически исключает возможность второго типа состояний гонки, описанных в доказательстве алгоритма 3, так как в таких автоматах в любом состоянии возможно поглощение данных только из одного набора входных портов.




(a) Не детерминированный по входу автомат Мили блока аппроксимации




(b) Детерминированный аналог

Рис. 6: Пример не детерминированного по входу автомата Мили 6a и его детерминированного аналога 6b

Несмотря на это преимущество, такие автоматы Мили весьма неудобны при построении схемы потока данных. Рассмотрим пример потока данных, содержащий аппроксимацию. Процесс аппроксимаций обычно включает в себя два блока: первый по набору точек строит модель аппроксимации, второй принимает модель и, используя ее, рассчитывает аппроксимированные значения в новых точках. Не детерминированный по входу автомат Мили работает по простому принципу: вначале принимает модель, в дальнейшем либо рассчитывает новое значение в заданной точке, либо изменяет модель (рисунок 6a). Чтобы привести автомат к детерминированному виду, придется добавить два дополнительных состояния и порт, данные из которого определяют следующие действие (рисунок 6b). Таким образом все действия с блоком осуществляются в два шага: передача следующего действия, передача данных для этого действия. Наличие дополнительных ребер для передачи действия в схеме потока данных уменьшает наглядность схемы, однако заставляет обращать внимание на возможные состояния гонки.

# 3 Заключение

В ходе работы были рассмотрены различные варианты формального представления потоков данных. Среди всех вариантов было выбрано представление в виде графов. На основе этого представления была построена модель потоков данных, характерной особенностью которой являются:

- замена программы блока на конечный автомат Мили, показывающий качественное поведение программы,
- большая свобода в поведении блока.

Основными преимуществами построенной модели являются:

- интуитивная понятность,
- гибкость в построении схемы потока данных

Были исследованы свойства модели, сформулированы критерии корректности потока данных. Также были построены алгоритмы получения возможных стратегий поведения потока данных, основанные на волновом алгоритме с учетом особенностей рассматриваемой модели. Результатом алгоритма является описание всех графов причинности, которые являются эквивалентом графов задач, применяемых для планирования запуска задач в распределенных системах. Поэтому для получения оценок требуемых ресурсов для запуска потока данных предлагается использовать уже существующие алгоритмы на базе HEFT-алгоритма. Также был предложен метод позволяющий осуществлять динамическое корректирование оценок в процессе выполнения потока данных. Основой для метода является построение префиксных деревьев графов причинности и расчет оценок для вычислительных ресурсов в каждом узле дерева.

Также были рассмотрены возможные неопределенности в схеме потока данных. Они условно разделены на два типа: неопределенности первого и второго рода. Неопределенности первого рода являются следствием упрощений в модели и учитываются при работе предлагаемого волнового алгоритма, неопределенности второго рода указывают на ошибки в построении потока данных и зависимость результата работы потока данных от особенностей при запуске потока данных. Главным типом ошибок второго рода является состояние гонки. Для обнаружения состояний гонки была предложена процедура проверки, применяемая в процессе работы волнового алгоритма. Остальные неопределенности второго рода были учтены в волновом алгоритме.

В процессе работы был создан модуль на языке программирования Python, позволяющий создавать блоки и строить потоки данных в соответствии с нашей моделью. Также модуль содержит все описанные в работе алгоритмы анализа потока данных. С помощью этого модуля все результаты были проверены как на искусственных потоках данных, так и на аналогичных реальным инженерным.

# 4 Дальнейшая работа

В работе была предложена модель потоков данных, для развития которой возможны другие направлений исследований. В частности, во время написания работы были предложены некоторые идея дальнейший исследований:

- создание стохастических моделей поведения потока данных (простейшая модель была рассмотрена в данной работе),
- построение на их основе оценок времени работы потока данных,
- обработка историй запуска потоков данных методам анализа данных,
- алгоритмы кластеризации (создание иерархии) потоков данных,
- дальнейшая модификации предложенной модели, например, путем добавления элементов потоков управления,
- анализ свойств потоков данных, построенных на основе наборов атомарных блоков с определенными свойствами,
- автоматическое приведение схемы потока данных к схеме с детерминированными по входу автоматами Мили.

# Список литературы


[1] The LCG TDR Editorial Board. Lhc computing grid, technical design report. Technical report, CERN, 2005.

[2] Top 500 supercomputer sites. `http://www.top500.org/list/2013/06/`, 2013.

[3] Lhc@home project. `http://lhcathome.web.cern.ch/LHCathome/`.

[4] Thomas M. Parks Edward A. Lee. Dataflow process networks. In *IEEE*, May 1995.

[5] M. Ghanem V. Curcin. Scientic workow systems - can one size t all? In *IEEE, CIBEC'08*, 2008.

[6] Gregor von Laszewski Kaizar Amin. *GridAnt: A Grid Workflow System.*

[7] Natalia Sidorova Nikola Trecka, Wil van der Aalst. Analyzing control-flow and data-flow in workfow processes in a unifed way.

[8] Kees van Hee Wil van der Aalst. *Workflow Management. Models, Methods, and Systems.* The MIT Press Cambridge, Massachusetts London, England, 2002.

[9] Wei Zhang Dan Ma. A static task scheduling algorithm in grid computing. In *Grid and Cooperative Computing*, pages 153–156. Springer Berlin Heidelberg, 2004.

[10] J. Janecek T. Hagras. A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems. *Parallel Computing*, 31:653–670, July 2005.