**Правительство Российской Федерации**

**Федеральное государственное автономное образовательное учреждение высшего профессионального образования**

**"Национальный исследовательский университет "Высшая школа экономики"**

***Отделение программной инженерии***

***Кафедра Управления разработкой программного обеспечения***

***ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА***

|  |  |
| --- | --- |
| На тему | «Библиотека программ для обеспечения взаимодействия агентов в |
| распределённых системах на основе обмена сообщениями» |

|  |  |
| --- | --- |
| Студента группы № | 271мУРПО |
| Лидского Георгия Алексеевича |
| (Ф.И.О.) |
| Научный руководитель |
| Д.ф.-м.н., профессор |
| (должность, звание) |
| Ломазова Ирина Александровна |
| ( Ф.И.О.) |

Москва, 2013

# Аннотация

В данной работе рассматривается реализация программного продукта на языке Java, предназначенного для координации распределённых процессов посредством обмена сообщений. Главной целью работы является создание надёжного каркаса, включающего в себя набор верифицированных распределённых алгоритмов обхода сети и выбора лидера. Данная программа может пригодиться в любой распределённой системе, в которой необходимо использование распределённых алгоритмов.

 Ключевые слова: распределённый алгоритм, распределённая система верификация, волновой алгоритм, обход сетей, поиск лидера, координирование.

 Работа состоит из шестидесяти шести страниц, тридцати двух глав, тринадцати иллюстраций, трёх таблиц, трёх приложений и десяти использованных источников.

# Оглавление

[Аннотация 2](#_Toc358491108)

[Оглавление 3](#_Toc358491109)

[Введение 5](#_Toc358491110)

[Распределённые системы 8](#_Toc358491111)

[Распределённые алгоритмы 9](#_Toc358491112)

[Алгоритмы обхода 11](#_Toc358491113)

[Алгоритм Тарри обхода связных сетей 13](#_Toc358491114)

[Алгоритмы избрания лидера 13](#_Toc358491115)

[Алгоритм Ле-Ланна выбора лидера 14](#_Toc358491116)

[Описание программной системы «Хореография» 15](#_Toc358491117)

[Обмен информацией с узлами распределённой системы 16](#_Toc358491118)

[Архитектура Хореографии 18](#_Toc358491119)

[Агенты 18](#_Toc358491120)

[Представители 20](#_Toc358491121)

[Сети 21](#_Toc358491122)

[Группы 21](#_Toc358491123)

[Алгоритмы 22](#_Toc358491124)

[Выполнения алгоритмов 23](#_Toc358491125)

[Менеджеры 23](#_Toc358491126)

[Пример работы Хореографии 26](#_Toc358491127)

[Верификация Хореографии 30](#_Toc358491128)

[Обзор существующих инструментов для проверки моделей 32](#_Toc358491129)

[Верифицируемые модули 33](#_Toc358491130)

[Свойства для верификации распределённых алгоритмов 34](#_Toc358491131)

[Свойства для верификации алгоритма обхода Тарри 35](#_Toc358491132)

[Настройка Java Pathfinder 36](#_Toc358491133)

[Верификация алгоритма обхода Тарри 39](#_Toc358491134)

[Результаты верификации 46](#_Toc358491135)

[Список использованных источников 49](#_Toc358491136)

[Приложение 1. Руководство пользователя 51](#_Toc358491137)

[Приложение 2. Код настроек JPF для верификации алгоритмов 61](#_Toc358491138)

[Приложение 3. Код программной системы 66](#_Toc358491139)

#

# Введение

В последнее время распределённые системы приобретают всё большую популярность среди компьютерных систем. Это могут быть как многопоточные программы, выполняющиеся на одном компьютере, так и гигантские системы, состоящие из большого множества независимых компьютеров, собранных в одну сеть для решения определённой задачи.

Главными преимуществами распределённых систем являются повышение производительности за счёт параллельного вычисления и повышение надёжности за счёт дублирования. Но вместе с тем, распределённые системы более сложны, чем однопоточные системы, и при их проектировании и реализации возникают проблемы. К примеру, если узлы распределённой системы должны общаться друг с другом посредством передачи сообщений, то необходимо обеспечить надёжный канал обмена данных, потому что не исключено, что в процессе передачи сообщений, какая-либо часть системы может сработать неверно и сообщение не будет доставлено.

Также, одной из главных проблем в распределённых системах является задача координации процессов, заключающаяся в программировании узлов системы на выполнение нужных действий в нужный момент времени для решения общей для всех узлов задачи. Сложность этой задачи заключается в отсутствии у каждого процесса сведений о глобальном состоянии системы, в отсутствии глобального таймера и неопределённости в последовательности выполнения команд узлами системы.

Обычно при решении задач координации затрагиваются такие проблемы, как:

* обеспечение распространения информации по всем узлам сети;
* обеспечение синхронизации узлов;
* возможность избрания лидера;
* обнаружение завершения;
* предотвращение взаимных блокировок;
* обеспечение доступа к общедоступным ресурсам.

Некоторые из этих задач успешно решаются с помощью таких инструментов как Apache Zookeeper, Noah или Doozerd. Данные инструменты решают проблему координации узлов с помощью создания файловой системы, доступной каждому процессу распределённой системы и обеспечивающей механизм по синхронизированной записи и чтении данных. Использование подобной файловой системы позволяет выстраивать действия узлов в строгом порядке, а также позволяет добиться синхронизации и широковещательного распространения информации.

Возможен и другой подход к обеспечению координированных действий в распределённой системе: реализация распределённых алгоритмов. Например, алгоритмов обхода сети и выбора лидера, позволяющих решить большинство вышеизложенных задач распределённых систем. В данный момент существует большое количество распределённых алгоритмов, подходящих под нужды систем любой сложности. Например, в книге [1] дан подробный анализ многих алгоритмов. Тем не менее, при их реализации могут возникнуть определённые трудности.

Например, если распределённая система должна отвечать требованиям надёжности или безопасности, то реализация того или иного распределённого алгоритма будет довольно сложным и трудоёмким процессом, ведь выполнение алгоритма не должно приводить к ошибкам и дефектам системы. Ошибки могут возникнуть как в процессе внедрения алгоритма в систему, так и в процессе его проверки на корректность. Для поиска ошибок в реализованном алгоритме можно воспользоваться различными методами, такими как статический анализ кода, тестирование или верификация. Очевидно, что для успешного внедрения алгоритма в систему, необходимо обладать достаточными знаниями в области верификации распределённых систем.

Для того чтобы избежать типичных проблем, возникающих при реализации распределённых алгоритмов, и упростить процедуру координации распределённых процессов, можно воспользоваться программной системой «Хореография», описанной в данной работе.

Система «Хореография» выполняет следующие функции:

1. Она является контейнером распределённых алгоритмов, таким образом обеспечивая возможность повторного использования реализаций алгоритмов.
2. Она позволяет производить координацию узлов, подключенных к ней, посредством выполнения заложенных в ней распределённых алгоритмов и передачи сообщений через сеть по протоколу TCP. Для обеспечения координации распределённых процессов достаточно просто наладить процесс обмена сообщениями между распределённой системой и Хореографией.
3. Программа даёт гарантию надёжности своей работы и реализованных алгоритмов посредством проведения её верификации.

Хореография является бесплатным продуктом с открытым исходным кодом. Она написана на языке Java, поэтому может запускаться на любом устройстве с виртуальной машиной Java. Благодаря взаимодействию с распределёнными процессами через протокол TCP, клиенты Хореографии не привязаны к каким-либо конкретным языкам программирования: от них требуется лишь возможность передачи текстовых сообщений по сети.

# Распределённые системы

Для того чтобы понять, как работает Хореография, необходимо ознакомиться с такими понятиями как распределённые системы и распределённые алгоритмы.

Распределённая система – это взаимосвязанное множество автономных процессов, называемых узлами системы, объединённых в одну сеть обмена информацией. Необходимым условием независимости узлов является наличие у них собственного автономного модуля управления. Два узла, способные обмениваться между собой информацией, называются взаимосвязанными.

Узлы распределённой системы взаимодействуют друг с другом за счет доступа к общим ресурсам или посредством обмена сообщениями. Различают два способа передачи сообщений между узлами распределённой системы.

Первый способ называется синхронным обменом сообщениями и заключается в том, что процесс передачи сообщения является единым неделимым событием в системе. То есть, сообщение не будет отправлено до тех пор, пока получатель не будет готов его принять.

Второй способ называется асинхронным обменом сообщениями и означает, что события отправки и приёма сообщений являются различными событиями в системе. Таким образом, сообщение может быть отправлено даже в том случае, если получатель не готов его принимать. Другими словами, отправленное сообщение будет находиться в сети до тех пор, пока получатель его не примет.

В данной работе будут рассматриваться только те системы, в которых узлы взаимодействуют друг с другом посредством асинхронного обмена сообщениями.

Для изучения распределённых алгоритмов, необходимо ввести понятие системы переходов распределённой системы.

Системой переходов называется тройка $S=\left(C,\rightarrow ,I\right)$*,*

где *С* – это множество конфигураций (возможных состояний системы),

→ – отношение переходов на *С* (действий, которые система может совершать в этих в этих состояниях),

*I* – подмножество множества *С*, элементы которого называются начальными конфигурациями системы.

Как уже говорилось, распределенная система состоит из совокупности процессов. Каждый из процессов по отдельности также является системой переходов с тем условием, что он может взаимодействовать с коммуникационной системой сети.

Помимо обычных событий для взаимодействия с коммуникационной подсистемой у каждого процесса есть события отправления и события приема сообщений. Символом *М* обозначается множество всех возможных сообщений. Запись М(*М*) обозначает совокупность всех мультимножеств с элементами из *М*.

## Распределённые алгоритмы

Локальным алгоритмом процесса называется пятерка

$(Z, I,⊢^{i},⊢^{s},⊢^{r})$,

где Z – множество состояний,

I – начальные состояния,

$⊢^{i}$ – переходы между состояниями, связанные с внутренними событиями (отношение на множестве *Z x Z),*

$⊢^{s},⊢^{r}$ – переходы между состояниями, связанные с событиями передачи и приёма сообщений (отношения на множестве *Z x M x Z).*

Переход между двумя состояниями определяется следующим соотношением:

$$с⊢d ⇔ \left(c,d\right)\in ⊢^{i} ∪ ∃m\in M((c,m,d)\in ⊢^{s} ∪ ⊢^{r})$$

Выполнениями процесса называют выполнения системы переходов

(*Z,*$ ⊢$*, I* ).

Коммуникационная система координирует выполнения отдельных процессов в каждом выполнении системы в целом. Для описания этой координации, распределенная система рассматривается как система переходов, в которой множество конфигураций, отношение переходов и начальные состояния формируются из соответствующих компонентов процессов.

Распределенным алгоритмом для семейства процессов $P=\left\{p\_{1}…p\_{N}\right\}$ будет называться совокупность локальных алгоритмов, каждый из которых соответствует одному процессу из $P$.

В распределенных алгоритмах допускается большое многообразие возможных вариантов вычислений как в самих процессах, так и в подсистеме коммуникаций. Любое вычисление распределённого алгоритма – это множество событий $C$. Количество событий в вычислении $C$обозначается как $\left|C\right|$, а подмножество событий, произошедших в процессе $p$*,* обозначается как $C\_{p}$*.*

Пусть в процессе могут существовать внутренние события специального типа, которые называются событиями принятия решения и представляются оператором $decide$. Тогда волновым алгоритмом называется распределенный алгоритм, который удовлетворяет следующим требованиям:

1. Каждое вычисление конечно: $∀C:\left|C\right|<\infty .$
2. Каждое вычисление содержит хотя бы одно событие принятия решения:

$$∀C: ∃e\in C:e является событием решения decide.$$

1. В любом вычислении всякому событию решения предшествует в причинно-следственном отношении хотя бы одно событие в каждом из процессов:

$$∀C: ∀e\in C:(e- событие решения decide)⇒∀q\in C:∃f\in C\_{q}:f\leq e.$$

Процесс называется инициатором, если он запускает выполнение своего локального алгоритма самопроизвольно, то есть алгоритм запускается по некоторому условию внутри процесса. Неинициатор вовлекается в распределенный алгоритм только тогда, когда в ходе вычисления поступает сообщение, которое запускает выполнение его алгоритма. Первое событие инициатора – это внутреннее событие или отправка сообщения, первое событие неинициатора – это прием сообщения.

## Алгоритмы обхода

Алгоритмы обхода – это волновые алгоритмы, в которых все события в ходе вычисления линейно упорядочены по отношению причинно-следственной зависимости, причем последнее событие происходит в том же процессе, что и первое.

Каждый алгоритм обхода обладает следующими свойствами:

1. В каждом вычислении есть единственный инициатор, который, отправляя ровно одно сообщение, запускает выполнение алгоритма.
2. Любой процесс после получения сообщения либо отправляет ровно одно сообщение, либо принимает решение.

Из первых двух свойств следует, что в каждом вычислении ровно один процесс принимает решение. Говорят, что выполнение алгоритма завершает работу в том процессе, который принимает решение.

1. Алгоритм завершает работу в инициаторе, и к тому времени как это происходит, каждый процесс хотя бы один раз отправил сообщение.

Из вышеперечисленных свойств можно заключить, что в каждой достижимой конфигурации алгоритма обхода есть либо ровно одно сообщение, находящееся на этапе пересылки, либо ровно один процесс, который принял решение и еще не успел отправить сообщение. Поэтому, все сообщения в вычислении алгоритма обхода можно рассматривать как один объект, называемый маркером и представляющийся оператором $tok$, который переходит от одного процесса к другому и посещает, таким образом, все процессы.

Алгоритмы обхода могут успешно применяться в следующих ситуациях:

1. широковещательное распространение информациивсем процессам с тем условием, что некоторые выделенные процессы должны получить подтверждение о завершении выполнения широковещательной связи;
2. реализация глобальной синхронизации между процессами;
3. вычисление функции, значение которой существенно зависит от входных данных каждого процесса.

Несколько примеров применения алгоритмов обхода:

1. Процесс в пиринговой сети сообщает остальным процессам о том, что он является ведущим процессом. Сообщение должны получить все процессы, поэтому ведущий процесс пользуется алгоритмом обхода.
2. Инициатор предупреждает другие процессы о том, что сейчас запустится алгоритм, который требует, чтобы все процессы находились в определённом состоянии. К моменту принятия решения, все процессы в сети получат сообщение и перейдут в состояние готовности к запуску алгоритма.
3. Инициатор запускает алгоритм обхода, в котором каждый узел при получении сообщения должен выполнить команду ping по отношению к заданному адресу и записать в маркер минимальное и максимальное время, полученное командой ping. Таким образом, перед окончанием алгоритма инициатор получит сообщение, в котором будет содержаться информация о лучшем и худшем результате ping для сети.

## Алгоритм Тарри обхода связных сетей

Алгоритм обхода Тарри является первым реализованным алгоритмом в системе «Хореография».

Алгоритм Тарри предназначен для обхода произвольных сетей с двусторонней связью между узлами и определяется следующими правилами:

1. Процесс не передает маркер по одному и тому же каналу дважды.
2. Неинициатор передает маркер своей родительской вершине (процессу, от которого он впервые получил маркер) только если его невозможно передать по другим каналам согласно первому правилу.

## Алгоритмы избрания лидера

В задаче об избрании лидера требуется, начав вычисление из конфигурации, в которой все процессы находятся в одинаковом состоянии, прийти к такой конфигурации, в которой ровно один процесс находится в особом состоянии *leader*, в то время как остальные процессы находятся в состоянии  *lost.* Каждый инициатор вычисляет отличительные признаки всех инициаторов, после чего лидером избирается инициатор с наименьшим (или с наибольшим) признаком.

Алгоритм избрания лидера обладает следующими свойствами:

1. Вычисление может быть инициировано произвольным непустым множеством процессов. То есть, у вычисления может быть несколько инициаторов, в отличие от алгоритмов обхода, в которых должен быть только один инициатор.
2. Заключительная конфигурация достигается в каждом вычислении, и в каждой заключительной конфигурации существует ровно один процесс, который находится в состоянии *leader*, а все остальные процессы при этом находятся в состоянии *lost.*

Примеры использования алгоритмов избрания лидера:

1. После сбоя системы, процессы должны договориться, кто из них будет заниматься восстановлением системы. То есть из всего множества активных процессов должен быть выбран только один, который будет руководить процессом восстановления системы.
2. Поступающие в сеть данные должны распределяться по наиболее производительному процессу.
3. Для соединения двух сетей, в каждой сети выбираются такие процессы, у которых время доставки сообщений между сетями наименьшее.

## Алгоритм Ле-Ланна выбора лидера

Алгоритм Ле-Ланна для выбора лидера работает только на неориентированных кольцевых сетях. Каждый инициатор отправляет по кольцу маркер $tok$, в который вложен отличительный признак инициатора (вес инициатора), и все процессы передают далее этот маркер. Предполагается, что в каналах соблюдается очередность сообщений, и каждый инициатор должен отправить свой маркер, прежде чем получит маркер от любого другого инициатора. Когда инициатор *р* получает свой собственный маркер обратно, маркеры всех инициаторов уже прошли через *р,* и  *р* будет избран лидером в том и только в том случае, если *р* – наименьший (или наибольший) процесс среди всех инициаторов.

Алгоритм Ле-Ланна является вторым алгоритмом, реализованным в системе «Хореография».

# Описание программной системы «Хореография»

Хореография представляет собой каркас, хранящий в себе реализации распределённых алгоритмов и предоставляющий сервис по координации узлов распределённой системы с помощью реализованных алгоритмов. Это достигается путём генерации алгоритмами сообщений и отправки их узлам системы. Главной особенностью программы является её повышенная надёжность, обеспеченная с помощью применения к ней верификации моделей (model checking). Таким образом, инженеру, реализующему координацию узлов распределённой системы посредством алгоритмов можно положиться на Хореографию и не вникать в сложности реализации распределённых алгоритмов.

Хореография выполнена на языке Java, поэтому она может быть запущена с любого устройства, на котором установлена виртуальная машина Java, поддерживающая язык Java шестой версии и выше.

Важной особенностью Хореографии является то, что узлы распределённой системы (её клиенты) не ограничены каким-либо языком программирования. Такое послабление достигается за счёт того, что клиенты могут взаимодействовать с программой посредством передачи текстовых сообщений по протоколу TCP, используя сокеты. В этом случае, программа выступает в роли сервера, к которому подключаются клиенты – узлы системы. Но это вовсе не означает, что узлы обязаны всегда взаимодействовать с программой только по протоколу TCP. Хореография предоставляет интерфейс системы обмена данными с клиентами, так что инженеру, внедряющему программу в распределённую систему, достаточно реализовать методы из интерфейса, чтобы добиться необходимого способа передачи сообщений между программой и клиентами.

## Обмен информацией с узлами распределённой системы

Как было указано выше, Хореография взаимодействует с узлами распределённой системы посредством передачи текстовых сообщений. Исследуя возможные варианты реализации протокола для обмена сообщениями между Хореографией и агентами, я пришёл к выводу, что лучшим вариантом будет использование известного стандарта по взаимодействию интеллектуальных агентов ­­– FIPA (The Foundation for Intelligent Physical Agents). В частности, FIPA задаёт язык и описывает структуру сообщений, которыми должны обмениваться агенты для передачи друг другу информации. Этот язык называется ACL (Agent Communication Language) и он должен быть понятен каждому узлу в сети для взаимодействия с Хореографией.

 Главным плюсом использования сообщений на языке ACL является то, что Хореография может быть без дополнительных усилий внедрена в существующую многоагентную систему, поддерживающую стандарт FIPA, например, в систему, основанную на каркасе JADE (Java Agent Development Framework). Многоагентные системы являются частным случаем распределённых систем, узлами которых являются интеллектуальные агенты – процессы, выполняющие назначенное им задание самостоятельно, в течение длительного времени.

В многоагентных системах, как в распределённых системах, соблюдаются три правила:

1. Автономность – узлы системы могут работать независимо от других узлов.
2. Неопределённость состояния – у узлов нет информации о полном состоянии системы в конкретный момент времени.
3. Децентрализация – нет такого узла, который бы явно управлял действиями других узлов.

На рисунке 1 изображён пример общения двух агентов посредством обмена сообщениями на языке ACL: агент «А» просит агента «Б» выполнить отправку груза «box017» из локации «12» в локацию «19».



Рисунок 1. Пример общения двух агентов

Насколько видно из этого простого примера, ACL-сообщение состоит из набора заголовков (они выделены жирным шрифтом на рисунке) и их значений, некоторые из которых заключены в скобки или двойные кавычки. Сообщения ACL не ограничивается только вышеизложенными заголовками, но данный набор является минимальным обязательным набором заголовков, необходимым для передачи информации между Хореографией и клиентами.

Первый заголовок определяет тип сообщения: REQUEST обозначает, что отправитель сообщения запрашивает у получателя произвести какое-либо действие. Заголовок SENDER указывает отправителя и обязательно должен содержать имя агента, отправившего сообщение. Для того чтобы Хореография различала агентов, каждый агент должен обладать уникальным именем в пределах сервера, с которого он производит подключение.

Заголовок RECEIVER содержит список адресатов сообщения и должен содержать имена всех агентов, которым предназначено это сообщение. Заголовок CONTENT является телом сообщения, то есть его текстом. Заголовки ONTOLOGY и LANGUAGE указывают, какому протоколу общения принадлежит данное сообщение.

Все сообщения, передаваемые между узлами системы и Хореографией, должны иметь подобный вид, то есть следовать стандарту FIPA ACL.

## Архитектура Хореографии

 Так как Хореография представляет собой сервер, в котором выполняются распределённые алгоритмы, то она разработана таким образом, чтобы максимально реалистично эмитировать внутри себя поведение распределённой системы. С этой целью большая часть элементов программы по умолчанию являются отдельными процессами, и каждый процесс выполняется независимо от других в отдельной нити (thread). Это необходимо для того, чтобы во время выполнения распределённого алгоритма не терялись главные свойства распределённой системы: независимость, неопределённость состояния и децентрализация. Если бы программа была однопоточной, то реализованные распределённые алгоритмы таковыми не являлись бы, потому что каждое их выполнение проходило бы по одному и тому же пути.

 Далее описаны главные элементы системы.

## Агенты

 Как и в распределённой многоагентной системе, ключевым классом в программе является Агент (Agent). Агенты работают в отдельных процессах и общаются с другими агентами посредством передачи сообщений ACL. У агента обязательно должно быть уникальное имя, состоящее из адреса узла и символьного представления имени.

Каждый агент может находиться в трёх состояниях:

1. Агент остановлен и готов к запуску (STOPPED) – агент находится в этом состоянии сразу же после создания. Он ещё не может принимать и отправлять сообщения. Чтобы запустить агента, нужно выполнить функцию startup(). В этой функции агент обычно обрабатывает входящие параметры, а потом стартует.
2. Агент работает (ALIVE) – агент выполняет заданные функции и может обмениваться информацией с другими работающими агентами.
3. Агент завершил работу (DEAD) – каждый агент проходит через это состояние, если его работа полностью завершена. Из завершённого состояния агент не может стартовать заново.

У каждого агента есть почтовый ящик, в который поступают сообщения от других агентов. Так как каждый агент выполняется в отдельной нити, то процесс передачи сообщений получается асинхронным. При чём, в нём соблюдается очередность обработки поступающих сообщений. При появлении сообщения в ящике, срабатывают обработчики сообщений, которых у агента может быть произвольный набор, заданный при запуске агента. Каждый обработчик сообщения вначале проверяет, подходит ли сообщение под его задачи, и если подходит, то выполняет необходимые действия от имени агента. Обработка сообщений из почтового ящика выполняется в том же процессе, в котором работает агент, поэтому все сообщения обрабатываются последовательно, один за другим. Пока одно сообщение не будет полностью обработано, следующее сообщение не будет обрабатываться.

Сообщения в программе представлены классом ParamValueMessage и являются производным от класса ACLMessage, поставляемым вместе с библиотекой FIPA JADE. Таким образом, агенты обмениваются сообщениями на языке ACL. Отличие ParamValueMessage от родительского класса состоит в том, что текст их тела записывается в виде “ПАРАМЕТР\_1=ЗНАЧЕНИЕ\_1 & ПАРАМЕТР\_2=ЗНАЧЕНИЕ\_2…” То есть тело сообщений имеет вид, схожий с записью параметров в строке браузера. При этом заголовок LANGUAGE для таких сообщений имеет значение “parameter-value”. Клиенты должны учитывать эту структуру при отправке и принятии сообщений от Хореографии. Сообщения с другим языком будут отклонены системой.

## Представители

Так как распределённые алгоритмы выполняются внутри Хореографии, то для выполнения каждого алгоритма необходима сеть из элементов, выполняющих роль узлов распределённой системы, запросивших выполнение данного алгоритма. Элементами такой сети являются агенты, однозначно связанные с узлами внешней распределённой сети (то есть, одному узлу распределённой системы соответствует ровно один агент). Эти агенты называются представителями (Representative), и каждому узлу внешней распределённой системы после подключения к Хореографии выделяется представитель этого узла внутри программы. Представители носят точно такое же имя, как и привязанные к ним узлы, так что все сообщения, отправленные внешним узлом в систему, внутри сети алгоритма будут отправляться от имени представителя. Соответственно, все сообщения, отправляемые представителю другими представителями, будут переданы внешнему агенту.

Таким образом, все события, происходящие с представителем внутри программы во время выполнения алгоритма, передаются в виде сообщений внешнему узлу. С помощью такого механизма и обеспечивается координация агентов распределённой системы. Получается, что узлы распределённой системы имеют полное представление о состоянии вычисления алгоритма, но от них не требуется совершать каких-либо действий, чтобы поддерживать работу алгоритма, потому что эту работу за них делает Хореография.

К примеру, если в Хореографии выполняется алгоритм обхода, то, как только маркер посещает представителя, в тот же момент узлу распределённой системы передаётся сообщение с содержанием этого маркера. Узел, получив маркер, становится обладателем той информации, которую должен донести инициатор алгоритма до всех узлов системы. Далее, от данного представителя маркер передаётся следующему представителю и его узел получает сообщение, и так далее до тех пор, пока алгоритм не завершится. При завершении алгоритма, узлу-инициатору передаётся сообщение принятия решения.

## Сети

Так как каждому распределённому алгоритму нужна сеть из узлов распределённой системы, то в Хореографии присутствует класс «Сеть» (Network). Сеть состоит из представителей (вершин) и каналов связи между представителями (рёбер). Два представителя, связанные общим каналом связи называются соседями. Каждый представитель знает своих соседей и в любой момент времени может послать им сообщение. В текущей версии реализована только кольцевая двунаправленная и неориентированная сети.

## Группы

Когда в распределённой системе требуется выполнить некоторую задачу посредством запуска распределённых алгоритмов, в Хореографии для этой цели создаётся одна или несколько групп (Group), к которым подключаются узлы, участвующие в выполнении задачи. Каждая группа предназначена для запуска необходимого алгоритма и имеет уникальное имя. Алгоритм, для которого создаётся группа, называется главным алгоритмом группы. Главный алгоритм может запускать дополнительные алгоритмы во время вычисления. Также, при создании группы указываются такие параметры, как предназначение группы (обход или выбор лидера), топология сети и условие для старта алгоритма в группе.

В упрощённом виде выполнение алгоритма в группе происходит по следующей схеме:

1. Узел подключается к Хореографии и сообщает ей название искомой группы. Если группа не найдена, то Хореография попытается её создать.
2. Для узла выделяется представитель, если его ещё нет. Затем группа добавляет представителя в свою сеть, на которой будет выполняться распределённый алгоритм. Узлу отправляется сообщение с подтверждением подключения к группе и номер предстоящего вычисления алгоритма. Этот номер нужен для различия разных запусков выполнения алгоритма в группе.
3. Выполнение распределённого алгоритма начинается по заранее определённому условию, которое можно задать при создании группы. Запуск производится стартером (ExecutionStarter), который отслеживает основные события, происходящие в группе, и запускает её, когда заданное условие становится истинным. Таким условием может быть, например, количество агентов в сети, получение группой специального стартового сообщения или истечение определённого времени. Как только алгоритм начинает выполнение, последующие заявки на вступление в данную группу приводят к подключению к новой сети. Таким образом, во время выполнения алгоритма, состав сети не меняется.
4. Во время выполнения алгоритма узлы распределённой системы получают те же сообщения, которые получают Представители в Хореографии от своих соседей.

## Алгоритмы

Основной класс алгоритмов – “Algorithm”. Этот класс представляет собой локальный алгоритм, хранящийся у каждого представителя, участвующего в вычислении распределённого алгоритма. Алгоритм хранит в себе реализацию распределённого алгоритма и, как только представитель получает алгоритмическое сообщение от своего соседа, анализирует сообщение и управляет дальнейшими действиями представителя.

Чтобы представитель понимал, какому именно вычислению алгоритма принадлежит полученное сообщение, у каждого вычисления есть уникальный идентификатор. Уникальность идентификатора задаётся названием группы, номером запуска распределённого алгоритма в этой группе и номером алгоритма в запуске группы.

При получении сообщения, в алгоритме происходят следующие действия:

1. Если это первое полученное сообщение в рамках текущего вычисления для локального алгоритма, то выполняется инициализация локального алгоритма, общая для инициаторов и обычных узлов.
2. Если узел подключился к сети как инициатор алгоритма и алгоритм ещё не инициализирован (то есть инициатор ещё не делал никаких действий), то сначала выполнятся первые шаги инициатора, и только после этого входящее сообщение обрабатывается стандартными действиями алгоритма.
3. Если узел подключился как неинициатор, то выполнятся стандартные действия алгоритма при получении сообщения
4. Узел распределённой системы получает сообщение, которое только что обработал алгоритм

При этом алгоритм может обработать сообщение только в том случае, если он активен. Если алгоритм завершился нормально или с ошибкой, то он больше не сможет обработать никакие сообщения.

## Выполнения алгоритмов

Для каждого вычисления алгоритма создаётся объект «Выполнение алгоритма» (AlgorithmRun). С помощью выполнения алгоритма представители получают локальные алгоритмы данного вычисления. Так же, этот объект следит за ходом вычисления алгоритма и посылает группе сообщение в случае успешного завершения выполнения. Иными словами, выполнение алгоритма – это оболочка вокруг локальных алгоритмов, принадлежащих одному вычислению.

## Менеджеры

В Хореографии есть несколько классов, которые управляют работой других классов. Эти классы называются менеджерами.

Менеджер сервера обеспечивает сервис обмена сообщениями между Хореографией и распределённой системой. Менеджер сервера задаётся во время настройки Хореографии. По умолчанию в параметрах задан менеджер, предоставляющий доступ протоколу TCP. Но в случае необходимости можно разработать свой менеджер, который позволит подключать узлы распределённой системы любым другим способом. Я выбрал протокол TCP по той причине, что он гарантирует целостность передаваемых данных. Это позволяет снизить сложность реализации распределённых алгоритмов. Менеджер сервера является первым агентом, который получает входящие сообщения от узлов распределённой системы. Далее, сообщения отправляются менеджеру групп.

Менеджер групп контролирует операции по созданию, поиску и удалению групп. Менеджер вступает в процесс обмена сообщениями, когда узел пытается создать группу или подключиться к существующей группе. Также, он служит контейнером всех групп, поэтому распределяет алгоритмические сообщения по группам, указанным в этих сообщениях.

Менеджер представителей обеспечивает каждый узел распределённой сети представителем, а также является справочником по всем представителям, заведённым в системе. Данный менеджер может найти представителя по его идентификатору.

Менеджер сообщений проверяет входящие сообщения на соответствие заданным правилам и структуре и указывает менеджеру сервера, какому классу принадлежит входящее сообщение. Все сообщения, которые не проходят проверку менеджера, фильтруются и не попадают далее в систему.

Обзорная схема классов приведена на рисунке 2.



Рисунок 2. Обзорная схема классов Хореографии

## Пример работы Хореографии

Предположим, что имеется система с распределённым компилятором, которую нужно координировать. Каждый процесс компилятора должен по определённой команде начать компиляцию своей части продукта, расположенного на указанном адресе.

Для решения этой задачи потребуется создать группу для обхода сети, в которой инициатором будет выступать программист, отправляющий продукт на компиляцию.

После настройки и запуска Хореографии (см. Приложение 1 для получения подробных инструкций по работе с Хореографией), каждый процесс распределённого компилятора должен подключиться к группе с заранее известным названием. Допустим, что группу заранее создаёт администратор системы. Это делается с помощью отправки Хореографии следующего сообщения:

(REQUEST

 :sender ( agent-identifier :name Admin :addresses (sequence http://127.0.0.1 ))

 :content "SUBJECT = CREATE\_GROUP & GROUP\_NAME = COMPILE & STARTER\_CLASS = ru.hse.choreography.groups.SimpleMessageStarter & GROUP\_TYPE = TRAVERSAL"

 :language parameter-value :ontology GROUPS

)

В сообщении говорится о том, что узел Admin просит создать группу с названием “COMPILE” для выполнения алгоритма обхода, причём запуск алгоритма должен производиться с помощью отправки специального сообщения. Надо заметить, что в данном сообщении в заголовке content между параметрами и их значениями поставлены пробелы для удобства чтения, но в реальных сообщениях пробелы ставить не нужно.

Если Хореография создаст требуемую группу, то она отправит администратору сообщение со следующим содержанием:

(INFORM

 :sender ( agent-identifier :name "Group Manager" :addresses (sequence localhost ))

 :receiver (set ( agent-identifier :name Admin :addresses (sequence http://127.0.0.1:50498 )) )

 :content "SUBJECT = GROUP\_CREATED & GROUP\_NAME = COMPILE"

 :language parameter-value :ontology GROUPS

)

 Сообщение гласит о том, что группа создалась успешно. После принятия такого сообщения можно начинать подключение процессов компилятора к этой группе с помощью следующего сообщения:

(REQUEST

 :sender ( agent-identifier :name Compiler\_one :addresses (sequence http://127.0.0.1 ))

 :content "SUBJECT=CONNECT&GROUP\_NAME=COMPILE"

 :language parameter-value :ontology GROUPS

)

В сообщении находится запрос на подключение к группе. После успешного подключения, процессу приходит следующее сообщение, в котором говорится, что процесс был подключен к выполнению алгоритма №1:

(INFORM

 :sender ( agent-identifier :name "Group Manager" :addresses (sequence localhost ))

 :receiver (set ( agent-identifier :name Compiler\_one :addresses (sequence http://127.0.0.1:50579 )) )

 :content "SUBJECT=CONNECT&GROUP\_NAME=COMPILE&EXECUTION\_ID=1"

 :language parameter-value :ontology GROUPS

)

 Допустим, мы подключили пять процессов. Теперь нужно подключить программиста, который должен сообщать всем процессам о начале компиляции продукта, исходный код которого лежит в указанном месте. Обратите внимание, что в тексте запроса присутствует параметр IS\_INITIATOR, сообщающий Хореографии о том, что подключающийся агент выступает в роли инициатора алгоритма. Также, во время подключения нужно задать текст, который будет содержаться в маркере обхода. Сообщение подключения программиста, выглядит следующим образом:

(REQUEST

 :sender ( agent-identifier :name Programmer :addresses (sequence

http://127.0.0.1 ))

 :content "SUBJECT = CONNECT & IS\_INITIATOR & GROUP\_NAME = COMPILE & TOK\_TEXT = http://sources\_are\_here.com"

 :language parameter-value :ontology GROUPS

)

 Теперь осталось только отправить сообщение, сигнализирующее начало выполнения распределённого алгоритма обхода:

(REQUEST

 :sender ( agent-identifier :name Programmer :addresses (

sequence http://127.0.0.1 ))

 :receiver (set ( agent-identifier :name COMPILE ) )

 :content "SUBJECT = START & EXECUTION\_ID = 1"

 :language parameter-value :ontology GROUPS

)

Во время работы алгоритма каждый процесс получит сообщение-маркер следующего содержания:

(INFORM

 :sender ( agent-identifier :name Compiler\_three :addresses (sequence http://127.0.0.1:50644 ))

 :receiver (set ( agent-identifier :name Compiler\_one :addresses (sequence http://127.0.0.1:50579 )) )

 :content "ALGORITHM\_CLASS = ru.hse.choreography.algorithms.traversal.TarryTraversal & MESSAGE\_TYPE = TOK & GROUP\_NAME = COMPILE & ALGORITHM\_ID = 0 & EXECUTION\_ID = 1& TOK\_TEXT = http://sources\_are\_here.com "

 :language parameter-value :ontology ALGORITHMS

)

 Как видно из сообщения, это маркер (MESSAGE\_TYPE = TOK) алгоритма обхода Тарри с текстом “http://sources\_are\_here.com”. Инициатору же после выполнения алгоритма придёт сообщение с принятием решения (MESSAGE\_TYPE = DECISION):

RECEIVE: (INFORM

 :sender ( agent-identifier :name Programmer :addresses (

sequence http://127.0.0.1:50731 ))

 :receiver (set ( agent-identifier :name Programmer :addresses (

sequence http://127.0.0.1:50731 )) )

 :content "ALGORITHM\_CLASS = ru.hse.choreography.algorithms.traversal.TarryTraverSal &

MESSAGE\_TYPE = DECISION & GROUP\_NAME = COMPILE & ALGORITHM\_ID = 0 & EXECUTION\_ID = 1"

 :language parameter-value :ontology ALGORITHMS )

# Верификация Хореографии

Для того чтобы создать не просто работающую программу, но надёжную основу, на которую смогут положиться разработчики распределённых систем, необходимо провести её верификацию. То есть, нужно теми или иными способами подтвердить, что при любом возможном выполнении, программа будет работать корректно. В данной главе описан процесс поиска методов верификации программной системы, применения этих методов и результаты верификации.

Задача верификации распределённых систем в последнее время приобретает всё большую популярность. Разрабатываются различные методики и программы, позволяющие проводить верификацию более точно и быстро. Так, например, в работах [2, 3] обсуждаются проблемы, возникающие при верификации распределённых систем, пути их решения и наиболее известные программы для верификации.

Для решения задачи верификации применяются формальные методы проверки моделей (model checking). Главным отличием проверки моделей от тестирования или статического анализа кода является то, проверка моделей охватывает всевозможные состояния и пути выполнения программы. Таким образом, если хотя бы в одном из всех возможных состояний программа не удовлетворяет некоторому свойству, то проверка моделей, при должной реализации, обнаружит это состояние. В работе [4] проводится сравнение качества верификации и статического анализа.

Проверка моделей играет очень важную роль в верификации распределённых систем, так как, в таких системах, в отличие от однопроцессовых систем, возможен не один путь выполнения программы. Это связано с независимостью выполнения узлов системы. Чем больше узлов, тем больше всевозможных состояний и путей выполнения приобретает программа.

Для проведения верификации системы нужно сначала построить математическую модель верифицируемой системы. Модель должна представлять собой конечный автомат, то есть должна иметь конечный набор состояний и переходов между ними:

$M=(S, I,s\_{0},δ)$,

где $S=\left\{s\_{0},…,s\_{k}\right\}$ – конечное множество состояний системы,

$I=\left\{i\_{0},…,i\_{j}\right\}$ – конечный входной алфавит,

$s\_{0}\in S$ – начальное состояние системы,

$δ:S×I\rightarrow S$ – функция переходов между состояниями системы.

 Так как такая модель обладает конечным набором состояний, то модель можно описать следующими наглядными способами:

1. Построить диаграмму состояний и переходов. Такая диаграмма записывается в виде ориентированного графа, узлы которого являются состояниями системы, а дуги – переходами между состояниями. У каждой дуги должна быть уникальная метка, которая служит для различения разных переходов.
2. Построить таблицу переходов. В строках такой таблицы записываются всевозможные состояния системы, а в столбцах – всевозможные входные символы из алфавита. На пересечении состояния и входного символа (то есть в ячейке) записывает состояние, в которое перейдёт система из текущего с учётом поступления заданного входящего символа.

Понятно, что с ростом возможных состояний и входного алфавита, размер модели может резко возрасти. Эта проблема называется комбинаторным взрывом (state explosion или combinatorial explosion). Очень часто эта проблема возникает при верификации моделей реальных систем. Несмотря на то, что есть некоторое количество способов борьбы с этой проблемой, таких как, например, символическое выполнение [5] или редукция частичных порядков [6], но в данный момент проблема остаётся актуальной. Особенно актуальна проблема комбинаторного взрыва для распределённых систем из-за большого количества всевозможных состояний процессов.

## Обзор существующих инструментов для проверки моделей

С возрастанием популярности распределённых систем, повысилась необходимость их верификации. Поэтому, в последнее время стали активно развиваться способы и инструменты построения и верификации моделей.

Для построения и верификации моделей можно воспользоваться множеством программ, в частности в работе [2] ведётся обзор наиболее популярных программ для верификации.

Можно воспользоваться инструментом CPN Tools для построения модели в виде сетей Петри, а затем проверить на ней выполнение необходимых свойств. Эта задача описана в статье [7]. Сети Петри являются распространённым и гибко настраиваемым инструментом для верификации распределённых систем.

Вторым популярным средством для проверки моделей является программа Spin. Spin отлично подойдёт для задачи верифицирования распределённых алгоритмов [8], но при его использовании для верификации реализаций этих алгоритмов на языке Java, могут возникнуть проблемы.

При верификации модели с использованием вышеуказанных инструментов, есть вероятность, что результаты окажутся неверными, так как модель может оказаться некорректна. То есть, найдётся свойство, которое будет выполняться на модели, но не будет выполняться в реальной системе. Обычно это происходит из-за чрезмерной абстракции модели, потому что, во-первых, в модели очень сложно учесть всю специфику языка, на котором написана система, а во-вторых, модель не должна быть слишком детальной, чтобы не возникла проблема комбинаторного взрыва.

В связи с этой проблемой, были разработаны различные программные продукты, способные строить модель автоматически по исполняемому коду верифицируемой программы. Такие инструменты способны строить очень точную модель программы, обходя стороной проблему корректности модели. Но, как показала практика, эти программы без должной настройки приводят к быстрому росту пространства состояний модели и, следовательно, к комбинаторному взрыву.

Из всех программ для проверки моделей по коду я выбрал Java Pathfinder (JPF в дальнейшем), так как этот проект до сих пор активно развивается, требует минимальных вмешательств в код моделируемой программы , обладает множеством дополнений и очень гибок в настройке. К тому же, на его основе разработан верификатор Basset для многоагентных систем, с помощью которого в будущем можно организовать верификацию Хореографии. В статье [9] говорится о том, что Basset справляется с многоагентными системами гораздо эффективней, чем Java Pathfinder. Главной сложностью, не позволившей мне выбрать Basset, является то, что верифицируемая программа должна быть построена поверх каркаса ActorFoundry, либо написана на языке Scala с поддержкой агентов.

Java Pathfinder является виртуальной машиной, которая создаёт пространство состояний по байт-коду верифицируемой программы. При этом он обладает некоторыми техниками уменьшения возможных состояний для ухода от комбинаторного взрыва. Например, с помощью редукции частичных порядков (Partial Order Reduction), Java Pathfinder добивается значительного уменьшения пространства состояний (около 70 процентов, как заявлено на сайте JPF). Благодаря множеству дополнений, JPF поддерживает различные методы верификации: проверку моделей через свойства линейной темпоральной логики (Linear Temporal Logic – LTL), символьную абстракцию величин, верификацию клиент-серверных приложений с помощью алгоритма кеширования входящих сообщений и др.

## Верифицируемые модули

В целях уменьшения лишних состояний модели, которые привели бы к комбинаторному взрыву, я решил разделить процесс верификации на несколько частей: верификацию реализованных алгоритмов, и верификацию корректности взаимодействия управляющих элементов Хореографии (то есть, менеджеров).

Первая часть затрагивает только верификацию реализованных распределённых алгоритмов. При верификации алгоритмов очень важно проверить, что алгоритм правильно решает задачу, для которой он создан. Например, в алгоритме Тарри для обхода сетей, важно проверить, что при получении сообщения алгоритм либо отправит одно сообщение, либо примет решение, и к моменту принятия решения все узлы хотя бы один раз отправят сообщение.

Вторая часть верификации затрагивает всё, что находится снаружи алгоритмов. Эта часть представляет собой каркас, благодаря которому происходит взаимодействие между распределёнными алгоритмами, внутренними агентами программы и узлами распределённой системы. В ней заложена логика программы, без которой выполнение алгоритмов, а, следовательно, и координация распределённой системы, невозможна. Большая часть проверки каркаса состоит из верификации межпроцессного взаимодействия внутренних агентов системы, а также из верификации протокола обмена информацией с клиентами.

В данной работе будет рассмотрен процесс верификации распределённого алгоритма обхода Тарри. Я выбрал этот алгоритм потому, что, с точки зрения верификации, он является самым сложным компонентом из всей системы и освещает большую часть сложностей и решений в верификации через Java Pathfinder.

## Свойства для верификации распределённых алгоритмов

Для того чтобы понять, какие свойства нужно выбрать для проверки распределённого алгоритма, необходимо понимать, какую задачу должен выполнять алгоритм. После нахождения свойств необходимо доказать, что алгоритм удовлетворяет этим свойствам во всевозможных состояниях вычисления алгоритма.

В данный момент в программе реализован алгоритм обхода Тарри и алгоритм поиска лидера Ле-Ланна. Чтобы верифицировать их, необходимо сначала определить задачи, выполняемые этими алгоритмами.

## Свойства для верификации алгоритма обхода Тарри

Реализованный в программе алгоритм обхода Тарри должен выполнять следующие задачи.

Во-первых, алгоритм должен быть волновым алгоритмом, то есть он должен удовлетворять всем условиям из определения волнового алгоритма:

1. Каждое вычисление конечно
2. Каждое вычисление содержит хотя бы одно событие принятия решения
3. В любом вычислении каждому событию принятия решения предшествует хотя бы одно событие в каждом из процессов

Во-вторых, алгоритм должен быть алгоритмом обхода:

1. В каждом вычислении должен быть только один инициатор, который запускает алгоритм, посылая одно сообщение
2. Каждый процесс после получения сообщения должен либо принять решение, либо отправить одно сообщение
3. Алгоритм завершает работу в инициаторе и к этому моменту каждому процессу удалось хотя бы один раз отправить сообщение

В-третьих, алгоритм должен отвечать особенностям алгоритма обхода Тарри:

1. Процесс не передаёт сообщение по одному и тому же каналу дважды
2. Неинициатор передаёт сообщение соседу, от которого он впервые получил маркер, только в том случае, когда его невозможно передать по другим каналам согласно первому правилу

Данные свойства описаны на неформальном языке и должны быть формализованы для проверки через Java Pathfinder. Очень важно провести формализацию корректно, потому что от этого процесса зависит дальнейшие результаты верификации. Сложность заключается в преобразовании свободно написанного текста, значение которого можно истолковать по-разному, в систему однозначных утверждений. К примеру, если взять первое правило, гласящее, что всякое вычисление конечно, то сразу возникают мысли по поводу того, каким объектом в системе представляется вычисление алгоритма и каким образом в системе считается, что вычисление окончено? Может быть, вычисление – это объект AlgorithmRun, который хранит в себе все локальные алгоритмы вычисления и считается завершённым тогда, когда инициатор присылает этому объекту сигнал принятия решения. А может быть, вычисление – это совокупность всех локальных алгоритмов представителей и оно считается завершённым только тогда, когда все локальные алгоритмы перейдут в состояние FINISHED. Проблеме формализации посвящена работа [10].

Таким образом, определив задачи, которые должен решать реализованный алгоритм, были найдены свойства, которые необходимо верифицировать. Верификацию с помощью Java Pathfinder можно выполнить различными способами. Я выбрал два наиболее подходящих способа.

Первый способ состоит в том, что на каждом шаге работы программы Pathfinder сообщает о текущем действии и состоянии системы своим наблюдателям (design pattern Observer). Таким образом, если создать необходимый наблюдатель, который будет получать полную информацию о состоянии системы, то он сможет проверить, выполняется ли необходимое свойство в этом состоянии или нет. Второй способ, используемый мной в работе, заключается в составлении и дальнейшей проверки LTL-формул.

## Настройка Java Pathfinder

Вначале проводил верификацию алгоритмов без отрыва от внешней части. То есть, Java Pathfinder проверял всевозможные состояния системы с момента получения Хореографией сообщения с запросом на подключение группы до момента завершения работы системы. Этот подход не зарекомендовал себя, так как привёл к огромному количеству состояний и переходов между ними. Даже при посылке одного единственного сообщения через сервер, Pathfinder не закончил верификацию через двенадцать часов работы. Комбинаторный взрыв происходил из-за того, что менеджеры и представители выполняются каждый в своём процессе, поэтому JPF пытался найти всевозможные комбинации чередований активных процессов. Так как агенты в Хореографии должны выполняться в разных процессах и эту логику нельзя менять, то задача уменьшения пространства состояний должна сводиться не к изменению кода Хореографии в угоду JPF, а к более точной настройке Pathfinder’а и к нахождению оптимального варианта для входных данных.

Мне удалось повысить скорость верификации благодаря настройке JPF на верификацию только локальных алгоритмов и их выполнения (Algorithm и AlgorithmRun соответственно). Так как алгоритм не может выполняться в пустоте, для него необходимо настроить среду: создать менеджеров, группу и подключить узлы распределённой системы. При этом главной задачей оставалось сделать так, чтобы JPF был неактивен всё это время. То есть, Pathfinder не должен был строить пространство состояний до того, как выполнение алгоритма пошлёт инициатору сигнал запуска. Мне удалось реализовать эту идею с помощью наблюдателя, который предотвращал выбор всевозможных комбинаций переключений процессов до тех пор, пока не начнётся работа алгоритма. Таким образом, начало процесса верификации начинается с отправки сообщения инициатору о начале запуска алгоритма.

Более того, я настроил Pathfinder на игнорирование каких-либо состояний после принятия инициатором решения. Это ограничение, как и предыдущее, реализовывается с помощью наблюдателя, обрезавшего всевозможные пути выполнения после того, как инициатор совершает решение. Таким образом, я добился того, что и после завершения алгоритма Pathfinder не будет генерировать всевозможные комбинации состояний. Получилось, что JPF проводит построение пространства состояний только от начала запуска алгоритма и до его завершения (считается, что оно наступает после принятия инициатором решения). Такой подход позволил в разы сократить время верификации одного алгоритма.

Но этого было недостаточно, так как даже для небольшого количества агентов, верификация выполнялась довольно долго (около часа для трёх узлов). Я ввёл в JPF следующее ограничение: как только алгоритм получает сообщение для обработки, никакой другой процесс не может перехватить выполнение до тех пор, пока алгоритм не завершит обработку сообщения. Это решение запретит JPF создавать промежуточные состояния со всевозможными переходами во время обработки одним из агентов алгоритмического сообщения. Такое решение гораздо упростит пространство состояний и, к тому же, не повредит процессу верификации, так как:

1. У алгоритмов, принадлежащих разным узлам сети, нет доступа к общим ресурсам, затрагивающим работу алгоритма. Другими словами, алгоритму совершенно неважно, в каком состоянии находятся соседи и что они делают во время обработки текущего сообщения. Это обеспечивается благодаря тому, что представители, внутри которых выполняются алгоритмы, могут взаимодействовать друг с другом только посредством обмена сообщениями.
2. Обмен сообщениями происходит в асинхронном режиме, и обработчик сообщений у каждого агента выполняется в отдельном процессе. Таким образом, пока одно сообщение не будет полностью обработано, никакие другие сообщения, попавшие в почтовый ящик агента, не будут обрабатываться одновременно с данным сообщением.
3. Состав сети и узлов не меняется на протяжении работы алгоритма. Это контролируется группой и, так как является частью каркаса, не относящегося к алгоритмам, считается всегда верным утверждением.

В связи с размером кода наблюдателей, реализация настроек Java Pathfinder для верификации распределённых алгоритмов приведена в приложении 2.

Хочу добавить, что, несмотря на то, что я добился значительного уменьшения времени верификации, пространство состояний слишком быстро растёт с увеличением размера сети. Для одного узла верификация заняла секунду, для двух узлов заняла одну минуту, а для трёх узлов – двадцать минут. То есть, проблема комбинаторного взрыва осталась в силе даже при таких значительных послаблениях. Это является доказательством того, что программы, строящие модель по выполняемому коду, необходимо очень точно и долго настраивать под конкретную систему, иначе они приведут к проблемам увеличенного потребления ресурсов и времени из-за слишком детальной модели.

## Верификация алгоритма обхода Тарри

Далее приведены примеры реализации проверки алгоритма обхода Тарри с помощью Java Pathfinder. Некоторые свойства алгоритма выполнимы только при условии, что внешняя часть системы (каркас) работает корректно, поэтому, если для выполнения свойства алгоритма требуется выполнение какого-либо свойства каркаса, это свойство считается истинным во время верификации алгоритма.

Далее приведён список свойств каркаса, всегда считающихся истинными на время верификации распределённого алгоритма.

1. Передача сообщений между представителями всегда происходит в асинхронном режиме. Каждое отправленное сообщение обязательно будет получено и обработано адресатом.
2. В рамках одного выполнения алгоритма сообщения могут передаваться только между участниками этого выполнения
3. Инициализация любого централизованного алгоритма начинается с того, что только инициатор получает сообщение. Если инициатора в сети нет, то алгоритм не начнёт работу.
4. В сети нет недоступных узлов: от каждого узла сети можно найти путь до любого другого узла. При этом каналы передачи сообщений являются двунаправленными
5. Принятие решения означает окончание выполнения алгоритма.
6. Каждый узел может обрабатывать одновременно не больше одного сообщения.

Теперь по порядку запишем свойства, выбранные выше для верификации алгоритма Тарри, формализуем их, а затем запустим Java Pathfinder на тестовом примере.

Первым свойством, которое необходимо проверить, является свойство конечности каждого вычисления.

За вычисление я принимаю объект AlgorithmRun, в котором есть два метода: start() и markAsFinished(), отвечающие, соответственно, за начало выполнения алгоритма и за его завершение. Метод start() вызывается группой в тот момент, когда срабатывает условие запуска алгоритма и сигнализирует инициатору алгоритма о том, что нужно начать вычисление. Метод markAsFinished() для всех алгоритмов обхода вызывается каждый раз, когда инициатор сообщает о решении.

Таким образом, надо проверить, что если был вызыван метод start(), то обязательно должен быть вызван метод markAsFinished(). Это легко решается с помощью расширения JPF-LTL, позволяющего проводить проверку моделей через формулы линейной темпоральной логики (Linear Temporal Logic). А именно, нужно добавить аннотацию с LTL-формулой к проверяемому классу:



Рисунок 3. LTL-формула для свойства №1 алгоритма Тарри.

 Такой записью мы хотим, чтобы Java Pathfinder проверил, что условие $□\left(startup()\rightarrow ◊markAsFinished()\right)$ будет выполняться всегда. На языке человека эта формула читается следующим образом: «всегда должно выполняться следующее: если в каком-либо состоянии системы был запущен метод startup(), значит, что когда-нибудь запустится метод markAsFinished()».

Далее проверяется свойство, заключающееся в том, что каждое вычисление содержит хотя бы одно событие принятия решения.

 Данное условие, так же, как и предыдущее, проверяется через внедрение LTL-формулы в класс AlgorithmRun. Но на этот раз нужно проверять, что после начала выполнения алгоритма обязательно выполнится метод addDecision(), добавляющий событие решения в выполнение алгоритма. Так как один класс не может содержать нескольких LTL-аннотаций, то необходимо к существующей аннотации добавить новое условие:



Рисунок 4. LTL-формула для свойств 1 и 2 алгоритма Тарри

Далее проверяется свойство, заключающееся в том, что алгоритм завершает работу в инициаторе и к этому моменту каждому процессу удалось хотя бы один раз отправить сообщение.

 Данное свойство является более строгим вариантом свойства 3 волнового алгоритма, поэтому мы сразу рассмотрим его и проигнорируем более слабое свойство волнового алгоритма. Это свойство проверяется с помощью доказательства выполнения других свойств:

1. При получении сообщения каждый узел либо отправляет ровно одно сообщение, либо принимает решение. Это свойство №2 алгоритма обхода. Данное свойство проверяется с помощью предположений (функции assert).



Рисунок 5. Проверка свойства через предположение

При получении сообщения, счётчик выполненных действий сбрасывается в ноль (это выполняет метод resetNumberOfActions). Затем, алгоритм производит обычные шаги при получении сообщения, после чего проверяется, что количество выполненных действий равно одному. То есть представитель либо отправил сообщение, либо принял решение. Увеличение счётчика происходит в методах отправки сообщения или принятия решения:





Рисунок 6. Увеличение счётчика при совершении действия

 Таким образом, Java Pathfinder проверит, что при любой конфигурации системы, каждое полученное сообщение повлечёт за собой ровно одно действие. Это свойство очень важно для предупреждения блокировок вычисления, когда представитель получает сообщение, но ничего с ним не делает, и, следовательно, зависает всё выполнение алгоритма.

1. У каждого агента есть массив с соседями, в котором отслеживается, было ли сообщение отправлено тому или иному соседу. В начале выполнения алгоритма в этом массиве должны быть записаны все соседи узла и у каждого соседа должен стоять флаг не отправленного сообщения.

Массив, в котором хранится информация о том, каким из соседей уже было передано сообщение, называется msgIsSentToNeighbour. Необходимо убедиться, что в начале работы алгоритма размер этого массива равен размеру массива соседей представителя. К тому же, в качестве первоначальных значений в массиве должны находиться булевы переменные false, что означает, что сообщение ещё не передано ни по одному из каналов соседям.

Данная проверка изображена на следующем рисунке:



Рисунок 7. Проверка массива с соседями

Здесь надо отметить, что функция commonInitializationStep выполняется только один раз во время первичной инициализации вычисления.

Проверить это можно с помощью следующей LTL-формулы (в формуле функция commonInitializationStep сокращена до common):

$$□\left(\left(receive\rightarrow ○\left(¬receive Ucommon\right)\right)∧\left(common\rightarrow ○□¬common\right)\right)$$



Рисунок 8. Проверка выполнения шага инициализации

1. При отправке сообщения какому-либо узлу, в массиве соседей для этого узла ставится флаг отправки сообщения. При этом невозможна ситуация, когда сообщение отправляется узлу, у которого уже стоит флаг отправки сообщения.

Это свойство проверяется с помощью функции assert:



Рисунок 9. Проверка работы массива соседей

1. Процесс не может послать сообщение своему родителю (тому, от кого он в первый раз получил сообщение) до тех пор, пока в массиве соседей есть хотя бы один флаг неотправленного сообщения у неродительского узла. Это условие проверяется через assert, который располагается после assert’ов из предыдущего свойства.



Рисунок 10. Проверка отправки сообщения родителю

1. Инициатор не может принять решение, если в его массиве соседей есть флаг неотправленного сообщения. Если при принятии сообщения инициатор больше не может отправить сообщение одному из соседей, он принимает решение.



Рисунок 11. Проверка принятия решения инициатором

1. Только инициатор может принимать решение. Данная проверка добавляется после проверки из предыдущего свойства.



Рисунок 12. Проверка, что только инициатор принимает решение

Далее проверяется следующее свойство: алгоритм завершает работу в инициаторе и к этому моменту каждому процессу удалось хотя бы один раз отправить сообщение. Это свойство можно доказать с помощью следующих утверждений и вышеизложенных свойств:

1. Каждый сосед инициатора получит сообщение к моменту принятия решения. Если бы один из соседей инициатора не получил от него сообщения, то инициатор не смог бы принять решения, так как в его массиве соседей был бы флаг неотправленного сообщения.
2. Каждый сосед инициатора отправит ему сообщение к моменту принятия решения. Так как передаваемое сообщение в каждый момент времени является единственным сообщением в сети (это следует из правила, что при получении сообщения узел может отправить не больше одного сообщения), то для его отправки необходимо его сначала принять. Пусть у инициатора *N* соседей. Тогда для отправки сообщения всем своим соседям, он должен *N* раз получить сообщение. Первый раз инициатор получает сообщение от системы во время запуска алгоритма. Для принятия решения инициатору необходимо отправить сообщение всем своим соседям, а затем отправить сообщение с принятием решения. Таким образом, инициатору нужно отправить *N* сообщений соседям и одно сообщение системе (принятие решения). Чтобы отправить *N* сообщений, инициатор должен получить от своих соседей $N-1$ сообщение. Но чтобы впоследствии принять решение, инициатор должен получить ещё одно сообщение. Таким образом, инициатор должен получить от соседей *N* сообщений. С учётом того, что каждый сосед может отправить инициатору не больше одного сообщения, то к моменту принятия решения все соседи отправят инициатору сообщение.
3. Предыдущее правило можно расширить на любой узел сети, если считать что родитель для неинициатора играет ту же роль, что и система для инициатора: первое сообщение узел получает именно от этого узла и последнее сообщение должен этому узлу отправить.
4. Таким образом, к моменту отправки сообщения родителю, узел должен получить сообщение от всех своих соседей. Этот момент происходит до принятия решения, так как доказано, что принятие решения является завершающим состоянием выполнения алгоритма.
5. Таким образом, каждый узел сети к моменту принятия инициатором решения хотя бы один раз отправит сообщение своим соседям.

Таким образом, было получено формализованное представление требуемых свойств, и теперь осталось запустить Java Pathfinder на тестовом примере, в котором создаётся одна группа для выполнения алгоритма обхода Тарри.

## Результаты верификации

В результате выполнения проверки, были выявлены ошибки в работе алгоритма, которые могли привести к зависанию и неполучению узлами сообщений. Например, была выявлена серьёзная ошибка, которая могла привести к ситуации, когда агент, получив сообщение, не передаёт его дальше и не принимает решение. К тому же, в Java Pathfinder по умолчанию встроены мониторы гонок данных и взаимных блокировок. Результат проверки показал, что в системе имеются ситуации с возможными гонками данных. Благодаря этим результатам мне удалось улучшить архитектуру программы, а также немного ускорить её работу.

**Заключение**

Проделанная работа доказывает, что создание надёжных распределённых систем – очень сложное и хлопотливое занятие. Распределённые системы более подвержены ошибкам, которые зачастую невозможно найти и исправить с помощью тестирования или статического анализа кода. Эти ошибки чаще всего связаны с взаимодействием нескольких независимых процессов и могут возникнуть только за пределами среды разработки. Данная работа подтвердила это утверждение тем, что проведённая верификация выявила несколько очень серьёзных недочётов в работе алгоритмов, которые могли бы возникнуть в ходе работы Хореографии в реальной системе, но которые не были найдены во время тестирования и отладки.

Верификация посредством программы Java Pathfinder показала себя сильным инструментом, способным строить более точные и безошибочные модели системы, нежели методы проверки моделей, строящие пространство состояний не по байт-коду программы. Так, например, модель алгоритма обхода Тарри, построенная с помощью сетей Петри оказалась некорректна, так как не обнаружила некоторых ошибок, обнаруженных с помощью Java Pathfinder.

Метод верификации по байт-коду программы сильно снижает вероятность получения некорректной модели, но в то же время, он очень требователен к компьютерным ресурсам. Для верификации одного алгоритма с сетью всего из трёх агентов потребовалось двадцать минут и свыше гигабайта памяти. Понятно, что для более серьёзной верификации, покрывающей большее количество элементов и возможных ситуаций работы Хореографии, необходимо иметь доступ к гораздо большему количеству ресурсов.

Проблема чрезмерной детальности модели может быть решена с помощью точной настройки средства верификации. В данной работе процесс настройки Java Pathfinder занял основную долю затраченного на верификацию времени и способствовал заметному уменьшению времени верификации.

Разработанная программная система позволит осуществлять широковещательное распространение информации, синхронизацию, вычисление некоторой функции, зависящей от каждого узла сети; а также производить выбор лидера в распределённых системах. Внедрение Хореографии в распределённую систему позволит разработчикам повысить надёжность системы, избежав многих проблем, возникающих при координировании узлов. При этом для внедрения Хореографии необходимо запрограммировать узлы распределённой системы на получение и отправку текстовых сообщений, написанных на языке ACL известного стандарта FIPA. Особенно эффективным будет внедрение Хореографии во многоагентную систему, в которой уже налажен механизм обмена сообщениями ACL.

В качестве дальнейшей работы может быть предложен поиск средств, позволяющих уменьшить время верификации программной системы. Это может быть достигнуто как более точной настройкой Java Pathfinder, так и средствами применения того или иного расширения JPF, позволяющего более эффективно провести верификацию в конкретных случаях. Так же, дальнейшим направлением разработки программы является реализация в ней большего набора алгоритмов и верификация их на сетях с произвольной топологией и любым количеством узлов.

# Список использованных источников

[1] Тель Ж. Введение в распределённые алгоритмы/ Ж. Тель; пер. с англ. В.А. Захарова. –М.: МЦНМО, 2009. – 616с.: ил.

[2] Program Model Checking - A Practitioner’s Guide [Электронный ресурс]/ Masoud Mansouri-Samani, Peter C. Mehlitz, Corina S. Pasareanu, John J. Penix, Guillaume P. Brat, Lawrence Z. Markosian, Owen O’Malley, Thomas T. Pressburger, Willem C. Visser. – Электрон. текстовые дан. и граф. дан. – Ames Research Center Moffett Field, California, 94035-1000, 2008. – Режим доступа: http://sarpresults.ivv.nasa.gov/DownloadFile/59/17/Practitioner%27s%20Guidebook\_20070427%20\_final.pdf, свободный (дата обращения 05.06.2013).

[3] Konnov I. Who is afraid of Model Checking Distributed Algorithms [Электронный ресурс]/ Igor Konnov, Helmut Veith, Josef Widder. – Электрон. Текстовые дан. – Unpublished contribution to: CAV Workshop (EC)^2, 2012. – Режим доступа: http://forsyte.at/wp-content/uploads/2012/07/ec2-konnov.pdf, свободный (дата обращения 05.06.2013).

[4] Vorobyov K. Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches [Электронный ресурс]/ Kostyantyn Vorobyov, Padmanabhan Krishnan. – 5th International Workshop on Systems Software Verification, 2010. – Режим доступа: http://www.usenix.org/event/ssv10/tech/full\_papers/Vorobyov.pdf, свободный (дата обращения 05.06.2013).

[5] Khurshid S. Generalized Symbolic Execution for Model Checking and Testing [электронный ресурс]/ S. Khurshid, C. Pasareanu and W. Visser. – 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems, 2003. – Режим доступа: http://www.ece.utexas.edu/~khurshid/testera/GSE.pdf, свободный (дата обращения 05.06.2013).

[6] Lauterburg S. Evaluating Ordering Heuristics for Dynamic Partial-order Reduction Techniques [электронный ресурс]/ Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. – Fundamental Approaches to Software Engineering, 2010. – Режим доступа: http://faculty.salisbury.edu/~stlauterburg/publications/fase2010\_final.pdf, свободный (дата обращения 05.06.2013).

[7] Dedova A. From Code to Coloured Petri Nets: Modelling Guidelines [электронный ресурс]/ Anna Dedova and Laure Petrucci. – International Workshop on Petri Nets and Software Engineering, 2012. – Режим доступа: http://ceur-ws.org/Vol-851/paper8.pdf, свободный (дата обращения 05.06.2013).

[8] John A. Towards Modeling and Model Checking Fault-Tolerant Distributed Algorithms / Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith and Josef Widder // Model Checking Software: 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings / Ezio Bartocci, C. R. Ramakrishnan. – Springer, 2013. – стр. 209-226.

[9] Lauterburg S. A Framework for State-Space Exploration of Java-based Actor Programs / Steven Lauterburg, Mirco Dotta, Darko Marinov, Gul Agha // ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering / IEEE Computer Society, 2009. – стр. 468-479.

[10] John A. Starting a Dialog between Model Checking and Fault-tolerant Distributed Algorithms [электронный ресурс]/ Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, Josef Widder. – ArXiv e-prints, 2012. – Режим доступа: http://arxiv.org/pdf/1210.3839v1, свободный (дата обращения 05.06.2013).

# Приложение 1

**Руководство пользователя**

**Системные требования**

Программа запускается на любом компьютере с виртуальной машиной Java, поддерживающей версию 6 или выше.

**Содержание пакета с программой**

В установочный комплект входят два файла:

1. choreography.jar – дистрибутив с программой, который необходимо запустить в виртуальной машине Java
2. choreography.conf – файл с настройками программы

**Установка**

Скопируйте файлы choreography.jar и choreography.conf в любое подходящее место. Если вы скопировали эти два файла в разные директории, то вам необходимо установить системную переменную CHOREOGRAPHY\_CONF\_PATH, указывающую пусть к файлу с настройками.

**Настройка**

Программа может запуститься с параметрами по умолчанию без предварительной настройки. Настройки программы заданы в файле choreography.conf, которые пользователь может изменять под свои нужды. В файл настроек можно добавлять свои параметры по мере необходимости.

Далее описаны параметры в файле настроек по умолчанию:

SERVER\_HOST – адрес, на котором TCP-сервер будет ожидать подключения от узлов. Если этот параметр оставить пустым, то сервер будет ожидать подключения на любом возможном адресе.

SERVER\_PORT – порт, на котором TCP-сервер будет ожидать подключения от узлов.

SERVER\_BACKLOG – размер очереди на подключение к TCP-серверу.

TRAVERSAL\_DEFAULT\_ALG – алгоритм обхода по умолчанию, который выбирается для новой группы обхода, если он не задан в параметрах запроса создания группы.

TRAVERSAL\_DEFAULT\_NET – класс сети по умолчанию, который выбирается для новой группы обхода, если она не задана в параметрах запроса создания группы.

ELECTION\_DEFAULT\_ALG – алгоритм поиска лидера по умолчанию, который выбирается для новой группы поиска лидера, если он не задан в параметрах запроса создания группы.

ELECTION\_DEFAULT\_NET – класс сети по умолчанию, который выбирается для новой группы поиска лидера, если он не задан в параметрах запроса создания группы.

STARTER\_CLASS – класс стартера группы по умолчанию, который выбирается для новой группы, если он не задан в параметрах запроса создания группы.

**Запуск**

Для запуска программы необходимо выполнить следующую команду:

java -Dfile.encoding=UTF-8 -jar путь\_к\_choreography.jar

После запуска программы отобразится информация о том, что сервер ожидает подключения на определённом адресе. Теперь клиенты могут подключаться к программе.



Рисунок 13. Пример запуска программы

**Завершение**

Для завершения программы достаточно закрыть окно, в котором программа была запущена.

**Начало работы**

Для того чтобы начать взаимодействовать с программой, узел распределённой сети должен подключиться по протоколу TCP к адресу, указанному в настройках.

После подключения узел и программа могут обмениваться текстовыми сообщениями. Обратите внимание, что программа может обрабатывать только сообщения, построенные по стандарту FIPA ACL. Соответственно, к клиенту она отправляет сообщения, составленные по тому же стандарту.

При отправке и принятии сообщений от программы клиент должен следовать следующим базовым правилам:

1. При отправке сообщения клиент должен вначале послать программе длину сообщения в символах. Длина должна быть представлена в виде беззнакового числа, занимающего размер двух байт. После длины сообщения клиент может послать само сообщение.
2. При получении сообщения клиент должен руководствоваться аналогичному правилу. Первые два байта полученного сообщения – это длина тела сообщения.

**Создание группы**

Для того чтобы выполнить какой-либо распределённый алгоритм, необходимо, чтобы все узлы, которые должны участвовать в вычислении данного алгоритма, подключились к программе и присоединились к одной и той же группе, в которой этот алгоритм будет выполняться.

Группу можно рассматривать как аналог комнаты чата или видеоконференции. К ней подключаются узлы для выполнения наперёд известного распределённого алгоритма, который необходим для решения известной всем её участникам задачи.

Для того чтобы подключиться к группе, нужно её сначала создать. Для этого клиенту необходимо послать Хореографии сообщение, содержание которого изложено в следующих двух таблицах.

Таблица 1

Заголовки сообщения создания группы

|  |  |  |
| --- | --- | --- |
| **Заголовок** | **Значение** | **Описание** |
| Тип сообщения | REQUEST | Обозначает, что отправитель просит адресата выполнить определённое действие. |
| SENDER NAME | Имя узла-отправителя | Необходимо, чтобы у двух разных узлов распределённой сети были разные пары данных заголовков. |
| SENDER ADDRESS | Адрес узла-отправителя (необязательно) |
| LANGUAGE | parameter-value | Определяет язык, на котором написано сообщение. Язык parameter-value является обязательным для всех сообщений Хореографии. |
| ONTOLOGY | GROUPS | Определяет протокол общения. GROUPS означает, что сообщения предназначено для действий с группами. |
| SUBJECT | CREATE | У каждого запроса REQUEST должен быть параметр SUBJECT в теле сообщения. Этот параметр указывает, какова тема запроса. |

Таблица 2

Тело сообщения создания группы

|  |  |  |
| --- | --- | --- |
| **Параметр** | **Значение** | **Описание** |
| GROUP\_NAME | Название группы | Название для создаваемой группы. Если группа с таким названием уже существует, то новая группа не будет создана. |
| GROUP\_TYPE | Назначение группы | Указывает, для какого семейства алгоритмов будет создана группа. Возможные варианты – TRAVERSAL для алгоритмов обхода и ELECTION для алгоритмов выбора лидера. Этот выбор влияет на значения по умолчанию для необязательных параметров. |
| ALGORITHM\_CLASS | Класс распределённого алгоритма | Необязательный параметр. Если не задан, то значение берётся из параметра \*DEFAULT\_ALG, который задаётся в choreography.conf. |
| NETWORK\_CLASS | Класс сети | Необязательный параметр. Если не задан, то значение берётся из параметра \*DEFAULT\_NET в файле choreography.conf |
| STARTER\_CLASS | Класс стартера группы | Необязательный параметр. Указывает на класс, который будет отвечать за запуск группы. Если не задан, то значение берётся из параметра STARTER\_CLASS в файле choreography.conf |
| CONNECT\_AFTER\_CREATE | Подключиться после создания группы | Указывает Хореографии, что узел хочет подключиться к группе сразу же после её создания. Если группа с таким именем уже существует, узел будет подключен к существующей группе. |
| IS\_INITIATOR | Флаг инициатора | Если этот параметр присутствует в запросе, то узел будет подключен к группе как инициатор алгоритма. Этот параметр имеет значение, если выставлен параметр CONNECT\_AFTER\_CREATE |
| Остальные параметры | Остальные нефиксированные параметры | Среди таких параметров могут быть, например, параметры для алгоритма (вес в выборе лидера) или для стартера (количество агентов для запуска). |

В данный момент в программе доступны три стартера на выбор:

1. ru.hse.choreography.groups.TimerStarter запускает выполнение группы по истечении определённого времени. Время в миллисекундах указывается в параметре MILLIS\_TO\_START при создании группы.
2. ru.hse.choreography.groups.SimpleMessageStarter запускает выполнение группы по принятии специального сообщения StartRequest. Состав этого сообщения описан в таблице 3.
3. ru.hse.choreography.groups.AgentNumberStarter запускает выполнение группы при наборе необходимого количества участников. При этом, среди них должен быть хотя бы один инициатор. Количество, необходимое для старта группы, задаётся параметром NUMBER\_TO\_START при создании группы.

Во всех последующих таблицах с сообщениями будут опущены заголовки SENDER и LANGUAGE, так как они остаются неизменными для одного узла в течение всего времени общения с программой.

Таблица 3

Сообщение старта группы

|  |  |
| --- | --- |
| **Заголовок** | **Значение** |
| Тип сообщения | REQUEST |
| ONTOLOGY | GROUPS |
| SUBJECT | START |
| RECEIVER NAME | Название запускаемой группы |
| Тело: EXECUTION\_ID | Номер выполнения, которое необходимо запустить в группе. Номер выполнения можно узнать из оповещения о подключении к группе. |

Пример запроса создания группы:

(REQUEST

 :sender ( agent-identifier :name Agent\_name)

 :content "SUBJECT = CREATE\_GROUP & GROUP\_NAME = COMPILE & STARTER\_CLASS = ru.hse.choreography.groups.SimpleMessageStarter & GROUP\_TYPE = TRAVERSAL"

 :language parameter-value :ontology GROUPS )

В случае успешного создания группы, агенту отсылается сообщение следующего вида:

(INFORM

 :sender ( agent-identifier :name "Group Manager" :addresses (sequence localhost ))

 :receiver (set ( agent-identifier :name %Agent\_name% :addresses (sequence %Agent\_address% )) )

 :content "SUBJECT=GROUP\_CREATED&GROUP\_NAME=%Group\_name%"

 :language parameter-value :ontology GROUPS )

В случае отказа, агенту отсылается сообщение следующего вида:

(REFUSE

 :sender ( agent-identifier :name "Group Manager" :addresses (sequence localhost ))

 :receiver (set ( agent-identifier :name %Agent\_name % :addresses (sequence %Agent\_address% )) )

 :content " SUBJECT = CREATE\_GROUP & REASON = GROUP\_ALREADY\_EXISTS & GROUP\_NAME = %Group\_name% &

GROUP\_TYPE = TRAVERSAL & ALGORITHM\_CLASS = ru.hse.choreography.algorithms.traversal.TarryTraversal & NETWORK\_CLASS = ru.hse.choreography.groups.networks.CircleNetwork & STARTER\_CLASS = ru.hse.choreography.groups.SimpleMessageStarter"

 :language parameter-value :ontology GROUPS )

В данном сообщении говорится о том, что создание группы отклонено по причине существования группы с таким же названием. Далее приведены параметры имеющейся группы.

Если в теле запроса на создание группы не указан параметр CONNECT\_AFTER\_CREATE, то для подключения к группе необходимо послать Хореографии запрос на подключение к группе:

(REQUEST

 :sender ( agent-identifier :name %Agent\_name%)

 :content "SUBJECT=CONNECT&GROUP\_NAME=%Group\_name%”

 :language parameter-value :ontology GROUPS )

В данном сообщении, как и в запросе создания группы, можно указать параметры для выполнения алгоритма. Например, если агент подключается к группе обхода как инициатор, то он может указать текст сообщения, передаваемого узлам сети во время обхода через параметр TOK\_TEXT. Если агент подключается к группе выбора лидера как инициатор, то он должен указать свой вес с помощью параметра WEIGHT. Этот параметр должен принимать целочисленные значения.

После успешного подключения к группе, агенту приходит сообщение, указывающее выполнение (EXECUTION\_ID), к которому его подключили:

(INFORM

 :sender ( agent-identifier :name "Group Manager" :addresses (sequence localhost ))

 :receiver (set ( agent-identifier :name %Agent\_name%) )

 :content "SUBJECT = CONNECT & GROUP\_NAME = %GROUP\_NAME% & EXECUTION\_ID=1"

 :language parameter-value :ontology GROUPS )

В случае неуспешного подключения, агенту отсылается отказ. В подключении может быть отказано по причине того, что агент уже подключен к текущему выполнению группы или если указанная группа не найдена. Возможные причины отказа не ограничиваются этими двумя ошибками. В каждом конкретном случае текст ошибки будет указан в параметре REASON тела отказа.

Во время выполнения алгоритмов, узлам распределённой сети могут отправляться алгоритмические сообщения, например сообщение TOK или DECISION.

Сообщение TOK является маркером в алгоритмах обхода и выбора лидера и имеет следующий вид:

(INFORM

 :sender ( agent-identifier :name Initiator

 :receiver (set ( agent-identifier :name %Agent\_name%))

 :content "ALGORITHM\_CLASS = ru.hse.choreography.algorithms.traversal.TarryTraversal &

MESSAGE\_TEXT = %Hi\_there% & MESSAGE\_TYPE = TOK &

GROUP\_NAME = COMPILE & ALGORITHM\_ID = 0 & EXECUTION\_ID = 1"

 :language parameter-value :ontology ALGORITHMS )

Сообщение DECISION указывает на то, что инициатор произвёл решение в алгоритме:

RECEIVE: (INFORM

 :sender ( agent-identifier :name Initiator )

 :receiver (set ( agent-identifier :name Initiator))

 :content "ALGORITHM\_CLASS = ru.hse.choreography.algorithms.traversal.TarryTraversal &

MESSAGE\_TYPE = DECISION &

GROUP\_NAME = COMPILE & ALGORITHM\_ID = 0 & EXECUTION\_ID = 1"

 :language parameter-value :ontology ALGORITHMS )

# Приложение 2

**Код настроек JPF для верификации алгоритмов**

Далее приведён код наблюдателей Java Pathfinder, необходимый для уменьшения времени верификации алгоритмов.

Код прослушивателя, удаляющего состояния после окончания работы алгоритма приведён ниже:

package ru.hse.choreography.jpf.listener;

import gov.nasa.jpf.PropertyListenerAdapter;

import gov.nasa.jpf.jvm.JVM;

import gov.nasa.jpf.jvm.MethodInfo;

import gov.nasa.jpf.jvm.bytecode.INVOKESTATIC;

import gov.nasa.jpf.jvm.bytecode.Instruction;

public class NiceFinisher extends PropertyListenerAdapter

{

 @Override

 public void instructionExecuted ( JVM vm )

 {

 final Instruction lastInstruction = vm.getLastInstruction ();

 if ( lastInstruction instanceof INVOKESTATIC )

 {

 final INVOKESTATIC staticMethod = ( INVOKESTATIC ) lastInstruction;

 final MethodInfo invokedMethod = staticMethod.getInvokedMethod ();

 if ( isShutdown ( invokedMethod ) ) { vm.ignoreState ( true ); }

 }

 }

 private boolean isShutdown ( MethodInfo invokedMethod )

 {

 return invokedMethod != null &&

 "ru.hse.choreography.core.Choreography.shutdown".equals (

 invokedMethod.getBaseName () );

 }

}

Далее приведён код прослушивателя, удаляющего варианты переключения до начала работы алгоритма и во время обработки алгоритмом сообщения:

package ru.hse.choreography.jpf.listener;

import gov.nasa.jpf.PropertyListenerAdapter;

import gov.nasa.jpf.jvm.ChoiceGenerator;

import gov.nasa.jpf.jvm.JVM;

import gov.nasa.jpf.jvm.MethodInfo;

import gov.nasa.jpf.jvm.StackFrame;

import gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet;

public class AlgorithmThreadFixer extends PropertyListenerAdapter

{

 private static final String RECEIVE\_METHOD = "ru.hse.choreography.algorithms.Algorithm.receive";

 private SingleThreadCG lastSet;

 private boolean executionIsStartedUp = false;

 @Override

 public void choiceGeneratorRegistered ( JVM vm )

 {

 final ChoiceGenerator<?> cg = vm.getNextChoiceGenerator ();

 if ( cg != lastSet && cg instanceof ThreadChoiceFromSet )

 {

 final ThreadChoiceFromSet threadCG = (ThreadChoiceFromSet) cg;

 if ( !executionIsStartedUp ) { changeCG ( threadCG, vm ); }

 else

 {

 for ( StackFrame stackFrame : vm.getCurrentThread ().getInvokedStackFrames () )

 {

 if ( stackFrame.getMethodInfo ().getBaseName ().equals ( RECEIVE\_METHOD ) )

 {

 changeCG ( threadCG, vm );

 break;

 }

 }

 }

 }

 }

 @Override

 public void methodEntered ( JVM vm )

 {

 if ( !executionIsStartedUp )

 {

 final MethodInfo method = vm.getLastMethodInfo ();

 if ( method != null && method.getBaseName ().equals (

 "ru.hse.choreography.groups.GroupExecution.startup" ) )

 { executionIsStartedUp = true; }

 }

 }

 private void changeCG ( ThreadChoiceFromSet threadCG, JVM vm )

 {

 threadCG.setDone ();

 lastSet = new SingleThreadCG ( threadCG );

 vm.setNextChoiceGenerator ( lastSet );

 }

}

package ru.hse.choreography.jpf.listener;

import gov.nasa.jpf.jvm.ThreadInfo;

import gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet;

public class SingleThreadCG extends ThreadChoiceFromSet

{

 public SingleThreadCG ( String id, ThreadInfo[] set, boolean isSchedulingPoint )

 {

 super ( id, new ThreadInfo[] { set[0] }, isSchedulingPoint );

 }

 public SingleThreadCG ( ThreadChoiceFromSet threadCG )

 {

 this ( threadCG.getId () + "SingleThreadCG", threadCG.getChoices(),

 threadCG.isSchedulingPoint() );

 }

 public SingleThreadCG ( String id, ThreadInfo thread, boolean isSchedulingPoint )

 {

 this ( id, new ThreadInfo[] { thread }, isSchedulingPoint );

 }

}

# Приложение 3

**Код программной системы**

В связи с большим объёмом, код программной системы находится на электронном носителе, приложенном к данной работе.