3-бөлім Информатика

Раздел 3 Информатика

Section 3 Computer science

УДК 681.3.06

Бектемесов А.Т.[1*], Бурлибаев А.Ж.[1**], Иляетдинов Ф.А.[1***]

[1] Казахский национальный университет имени аль-Фараби, Республика Казахстан, г. Алматы
E-mail: *amanzhol.bektemessov@kaznu.kz, **aimurat.burlibaev@kaznu.kz
***iliyletdinov.farhad@kaznu.kz

Распределенные алгоритмы и их верификация с помощью Byzantine model checker

Каждый алгоритм системы является главным узлом для построения надежных распределенных систем. Для того, чтобы быть уверенным в том, что эти алгоритмы делают систему более надежной, мы должны гарантировать, что предлагаемые алгоритмы работают правильно. Однако, проверка на модели внедренной отказоустойчивости распределенных алгоритмов [1] в действительности возможно использовать только для очень маленьких систем, чтобы в конечном итоге иметь возможность автоматически проверять отказоустойчивость распределенных алгоритмов на больших системах. В этой статье мы рассмотрим моделирование и проверку алгоритма "Parsing"методом Broadcast вещания [2] с помощью Spin и ByMC (Byzantine Model Checker [3]). Предлагаемые свойства обеспечивают безопасность и живучесть на LTL (Linear-temporal logic).
Ключевые слова: распределенные алгоритмы, параллельная программа, ByMC, Model Checking, верификация.

Bektemessov A.T., Burlibaev A.Zh., Iliyaletdinov F.A.

Distributed algorithms and their verification with Byzantine model checker

Every system algorithms are the main nodes for the construction of reliable distributed systems. To ensure that these algorithms make the system more reliable, we have to ensure that the proposed algorithms are working properly. However, check the model implemented fault tolerance of distributed algorithms [1] in real can be used for very small systems only. To finally, be able to automatically check the fault-tolerant distributed algorithms on large systems. Since they are aimed at improving the reliability of computer systems, it is important that these algorithms are correct and fully satisfy their requirements. Due to different sources of non-determinism is easy to fail in the arguments of the correctness of distributed algorithms on the level of temporality. Therefore, they are unreliable material for model checking. Nevertheless, the model checking method for distributed fault-tolerant algorithm is extremely difficult work. In this article we will discuss the modeling and verification of algorithm Broadcast "Parsing"by broadcasting [2] with Spin and ByMC (Byzantine Model Checker [3]). The proposed properties are safety and liveness in the LTL(Linear-temporal logic).
Key words: Distributed algorithms, Parallel program, ByMC, Model Checking, Verification.

Бектемесов А.Т., Бурлибаев А.Ж., Илялетдинов Ф.А.

Үлестірілген алгоритмдер және оларды Byzantine моделді тексеріс арқылы верификациялау

Жүйенің әрбір алгоритмдері сенімді үлестірілген жүйелерді құруда негізгі түйіндері болып табылады. Бұл алгоритмдер жүйенің сенімді болуына кепілдік беру үшін, біз осы алгоритмдер дұрыс жұмыс жасауына кепілдік беруіміз қажет. Алайда, бекемдікпен негізделген үлестірілген алгоритмдерге моделді тексеріс орындау тек кішігірім жүйелер үшін ғана болуы мәлім [1]. Соңғы нәтижесінде, үлкен жүйедегі бекемделген үлестірілген алгоритмдерді автоматты түрде тексеру қажет. Бұл есептеу жүйелерінің сенімділігін арттыруға бағытталғандықтан ол алгоритмдер талаптарына сай дұрыс жұмысын жасауы өте маңызды. Әр түрлі детерминисттік емес түйіндері үшін темпоралды деңгейде үлестірілген алгоритмдердің корректілік аргументінде қате кету өте оңай болмақ. Сондықтан олар модель тексеруге сенімді материалдар бола алмайды. Алайда, үлестірілген алгоритмдердің сенімділігіне модельді тексеру әдісі өте қиын.
Бұл мақалада, Spin және ByMC (Byzantine Model Checker [3]) көмегімен "Parsing"алгоритмін Broadcast хабар таратушы [2] әдісімен моделдеп тексеруін қарастырамыз. Ұсынылатын LTL(Linear-temporal logic) қасиеттер қауіпсіздік және өмірсүргіштік.
Түйін сөздер: үлестірілген алгоритмдер, паралелльді программа, ByMC, модельді тексеріс, верификация.

Введение

Применение распределенных вычислений является способом решения трудоемких вычислительных задач с использованием нескольких компьютеров, объединенных в параллельную вычислительную систему. Распределенные вычисления применимы также в распределенных системах управления таких как Hadoop, MapReduce и MPJ [4]. Последовательные вычисления в распределенных системах выполняются с учетом одновременного решения многих задач. Особенностью распределенных многопроцессорных вычислительных систем является возможность неограниченного наращивания производительности за счет масштабирования, но исключение составляют локальные суперкомпьютеры. Для гарантированной работы между несколькими компьютерами репликация представляет собой классический подход, это значит, что компьютерная система является отказоустойчивой, то есть по-прежнему правильно выполняет свою задачу, даже если некоторые компоненты справляются не в полной мере. Основная идея репликации состоит в подтверждении дублирующихся компьютерных координат при работе нескольких компьютеров. Например, в случае репликации баз данных, происходит согласование на хранение одинаковой информации. При анализе параллельных процессов исключено делать предположений об относительных скоростях выполнения процессов или стратегии планировщика процессов. Большинство ошибок в параллельных программах – из-за непредвиденных перекрытий операций параллельных процессов.

```
byte state = 1;
proctype A() {byte tmp; (state==1) -> tmp = state; tmp = tmp+1; state = tmp}
proctype B() {byte tmp; (state==1) -> tmp = state; tmp = tmp -1; state = tmp}
init { run A(); run B() }
```

Если какой-нибудь процесс завершится до того, как другой процесс выполнит проверку state==1, то «запоздавший» процесс будет навсегда блокирован. Если проверка условия выполнится процессами до того, как другой процесс завершится, то оба процесса

завершатся, но значение переменной state будет непредсказуемым – она может принять любое значение: 0, 1 или 2. Более подробно исследовано в работе [5].

1 Постановка задачи

Согласование всех компьютеров на хранение одной и той же информации является нетривиальным из-за нескольких источников недетерминированности, что приведет к неопределенным задержкам сообщений и асинхронным шагам вычислений. Однако решение таких проблем как отказоустойчивость состояния репликации машин было рассмотрено ранее [6]. Так как они направлены на улучшение надежности вычислительных систем, очень важно, чтобы эти алгоритмы являлись правильными и полностью удовлетворяли их требованиям. Из-за различных источников недетерминированности легко потерпеть неудачу в аргументах корректности распределенных алгоритмов на уровне темпоральности. Как следствие они являются ненадежным материалом для проверки модели. Тем не менее, метод проверки на модели отказоустойчивых распределенных алгоритмов является чрезвычайно сложной работой [7], при этом возникают следующие проблемы:

- Параллелизм и выбор недетерминированности. Отказоустойчивые распределенные алгоритмы страдают от комбинаторного взрыва в пространстве состояний, а также от количества поведений.
- Корректность и решаемость проблем, в частности, степень параллелизма, задержки сообщений, а также сбой модели. Например, невозможно гарантировать правильное выполнение, если нет ограничений на количество неисправных компонентов в системе.
- Нет общей согласованной распределенной вычислительной модели, но существует довольно много вариантов, которые различаются тонкими деталями, такими как атомарность вычислительного шага.
- Распределенные алгоритмы обычно описываются псевдокодом, как правило, с использованием различных языковых псевдокодов, которые запутывают отношение формальных методов к псевдокодам, которые описывают достижение к основной цели модели.

2 Метод решения

2.1 Модельная проверка на безопасность

При разработке ПО необходимы свои разделы прикладной математики – это формальные методы, на которых основывается верификация. Используя разделы и методы, в конечном итоге мы должны гарантировать правильность поведения созданных нами систем. Требования к поведению систем для полной верификации необходимы для создания формальной спецификации системы. Одним из подходов к решению проблемы автоматизации отладки и проверки правильности программ является Model Checking. Для заданной анализируемой программы строится ее абстрактная формальная модель. Проверяемое свойство или требование выражается на формальном математическом языке в виде логической формулы:

$$M \models \phi \tag{1}$$

где некоторая булева формула удовлетворяет модели M. Рассмотрим множество ато-

марных высказываний AP. Пространство состояний моделируемой программы или программного комплекса можно формализовать, как модель Крипке (структуру Крипке). Моделью Крипке M над множеством атомарных высказываний AP называют четверку (S, S_0, R, L), где:

S - конечное множество состояний;

$S_0 \in S$ - множество начальных состояний;

$R \in S \times S$ - отношение переходов, которые обязаны быть тотальным, т.е. для каждого состояния $s \in S$ должно существовать такое состояние $s' \in S$, что имеет место $R(s, s')$;

$L : S \to 2^A P$- функция, которая помечает каждое состояние множеством атомарных высказываний, истинных в этом состоянии.

Путь в модели M из состояния s - это бесконечная последовательность состояний $\pi = s_0 s_1 ...$, такая, что $s_0 = s$ и для всех $i \geq 0$ выполняется $R(s_i, s_{(i+1)})$.

Моделируемый программный модуль на каждом состоянии выявляются множеством значений переменных $V = v_0, v_1, ...$, принимающих значения на конечном множестве D и определяющих отдельные компоненты и выполнение взаимодействие между ними. Множество AP состоит из $v_i = d_i$, где $d_i =\in D$. Таким образом, что каждое состояние s в M представляет $V \to D$.

Отношение R определяется следующим образом. Пусть имеются два состояния, s_1 и s_2. Если в s_1 имеется компонент, который может выполнить атомарный переход, в результате выполнения которого система будет находиться в состоянии s_2, тогда состояния s_1 и s_2 связаны отношением перехода: $(s_1, s_2) \in R$. В случае, если нет такого состояния s_2, для которого бы выполнялось $R(s_1, s_2)$, полагается $R(s_1, s_1)$, т.е. "тупиковое"состояние, связанное отношением перехода в рекурсиях [8]. Проверка живучести двух процессов в критических интервалах $@crit1$ и $@crit2$:

$$AG(@crit1 \wedge @crit2) \quad (2)$$

Таким образом, верификация программы сводится к проверке выполнимости формализованного требования спецификации на абстрактной модели программ.

2.2 Обеспечение безопасности распределенных вычислений или алгоритм византийского соглашения

В вычислительной технологии эксперимент призван показать проблему синхронизации состояния систем в случае, когда коммуникации являются надежными, а процессы могут быть дефектными и ненадежными. Часть процессов, включая главный процесс, могут быть противниками. Нужно построить единую стратегию действий, которая будет выигрышной для не дефектных процессов. Алгоритм решения можно представить следующим образом:

Шаг 1: Каждый из процессов посылает остальным сообщение, где указывает значение состояния. Дефектные процессы могут указать различные значения в разных сообщениях, а правильные указывают верные значения. Процесс p_1 указал p_1', процесс p_2 - p_2', процесс p_3 который является дефектным, соответственно указал трем остальным процессам неверные значения x, y, z, процесс p_4 - p_4'.

Шаг 2: Каждый из процессов вычисляет свой вектор из полученной информации. Получается: $vect_1(p_1, p_2, x, p_4)$, $vect_2(p_1, p_2, y, p_4)$, $vect_3(p_1, p_2, p_3, p_4)$, $vect_4(p_1, p_2, z, p_4)$.

Шаг 3: Процессы посылают свои вектора другим процессам. Процесс p_3 вновь посылает произвольные значения (Таблица 1).

Шаг 4: Каждый из процессов проверяет каждый элемент в полученных векторах. В том случае, если одно значение совпадает как минимум в двух векторах, оно помещается в результирующий вектор, в противном случае, соответствующий элемент помечается как «неисправен». В итоге все процессы получат один вектор (p'_1, p'_2, неисправен $, p'_4$). Следовательно, согласие на решения задач достигнуто. Для n=3 и m=1 согласие не достигнуто.

Таблица 1 – Результаты векторов от всех процессов

p_1	p_2	p_3	p_4
(p'_1, p'_2, y, p'_4)	(p'_1, p'_2, x, p'_4)	(p'_1, p'_2, x, p'_4)	(p'_1, p'_2, x, p'_4)
(a, b, c, d)	(e, f, g, h)	(p'_1, p'_2, y, p'_4)	(p'_1, p'_2, y, p'_4)
(p'_1, p'_2, z, p'_4)	(p'_1, p'_2, z, p'_4)	(p'_1, p'_2, z, p'_4)	(i, j, k, l)

Исходя из примера, мы понимаем, что алгоритм византийского соглашения позволяет нам произвести модельную проверку распределенных алгоритмов среди неисправных процессов [9].

3 Алгоритм Parsing

Рассмотрим распределенную работу алгоритма Parsing. Этот алгоритм предназначен для обработки больших данных. Работа с большими данными требует достаточно много ресурсов. Рассматриваемый алгоритм является очень простым, но распределение при параллелизме требует тщательного анализа для выявления тонких ошибок. Допустим, обработка на Hadoop 209k pdf текста для 16 узлов системы займет около 26 часов [10]. Но, к сожалению, неполадки внутри алгоритма вернут все труды назад.

Пошаговое выполнение алгоритма Parsing можно проиллюстрировать семью шагами:

1. Программа обработки находит первое предложение и переходит в пункт 2.
2. Зафиксирует предложение. Берет первое слово и переходит в пункт 3.
3. Сохраняет полученное слово.
4. Считает все вхождения этого слова в тексте.
5. Если предложение не закончено, добавляет к слову следующее слово и переходит в пункт 4. Если предложение закончилось, переходит в пункт 6
6. Если слово, с которого начиналось словосочетание, не последнее в предложении, то зафиксирует следующее слово в этом предложении и переходит в пункт 3. Если слово последнее, то переходит в пункт 7.
7. Если текст не закончился, переходит на следующее предложение и дальше в пункт 2.

На первый взгляд алгоритм очень простой, конечно, если учитывать саму разработку, то прилагается не малый труд. Есть некоторые подводные камни, например, в ходе тестирования и анализа алгоритма Parsing разработчики выявили, что не всегда

бывают идеально написанные статьи в файле формата pdf. Дублированные и лишние ключевые символы приведут программу к сбою или внесут некорректную информацию в базу.

3.1 Проверка алгоритма Parsing через Spin

Для алгоритма Parsing была создана модель на языке Promela (Process Meta Language), что можно сформировать и на ByMC:

```
active [N] proctype p(){
if :: BUF == 0 -> BUF = 1 - BUF; pro1=true;
```

Конечно же, запускаем исключительный подбор всех состояний процессов:

```
 do ::(i < num) -> j=0;
   do ::(j< num) ->
if ::(filenm[i] != Chfilenm[j]) -> Chfilenm[j] = filenm[i];
goto q1;
   :: else -> skip;
fi; j++;
   ::else -> break;
   od;
   :: else -> break;
q1:i++;
od;
BUF = 1 - BUF; pro2=false;
```

где filenm[] - модель обрабатываемых файлов, Chfilenm[] - буфер проверки файлов на дублирование. Цифры использованы вместо файлов и текстов в качестве моделей. Подобраны несколько LTL спецификаций на корректность. Требования оказались положительным для следующих формул безопасности на Spin:

$$FG(pr1\ AND\ pr2) \tag{3}$$

$$GF(pr1\ AND\ pr2) \tag{4}$$

где $pr1(pro1 == true)$ и $pr2(pro2 == true)$

Выявлена ошибка взаимного исключения - дедлок(Рисунок 1).

Проблема заключалась в атомарности оператора BUF = 1 - BUF. Так как, оператор верхнего порога делится без атомарности на несколько операций, вычислительная машина сотни миллисекундах не разбирает привилегии в простых арифметических операциях. То есть, если в процессе p_0 обрабатывается операция присвоения BUF = 1, чтобы вычесть 1 - BUF. И в этих миллисекундах процесс с номером p_1 может выполнить проверку для входа в критический интервал, и он входит, потому что значение BUF еще не успело поменяться на 1, а p_0 еще не вышел из критического интервала. Если алгоритм требует улучшения по времени и обработки больших данных, можно предугадать ошибки параллельно-модернизированного алгоритма.

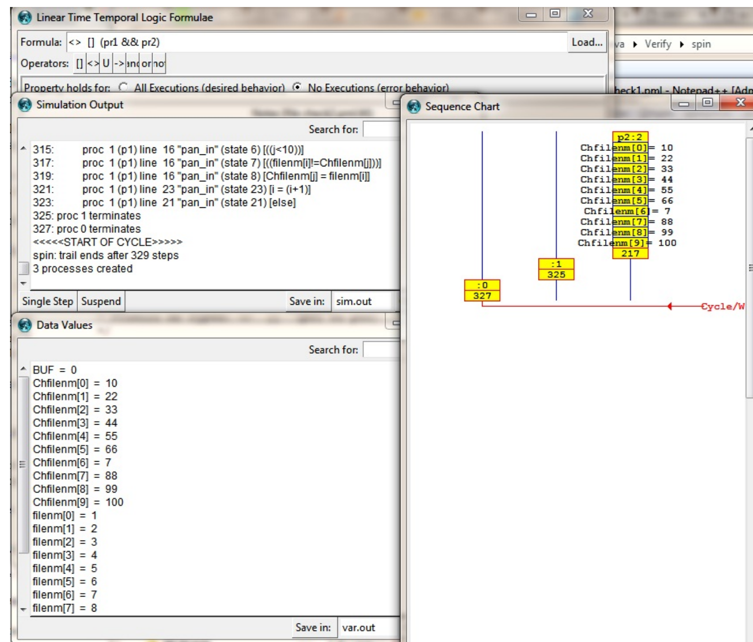

Рисунок 1 –Верификация параллельного алгоритма "Parsing"в системе Spin

Параллельные программы могут годами сохранять ошибки, проявляющиеся после долгой эксплуатации как реакция на возникшую специфическую комбинацию многочисленных факторов, в частности, непредсказуемых скоростей выполнения отдельных процессов в параллельных программах.

При параллельном программировании обычно имеем дело с такой концепцией как рандеву, но серьезной проблемой при разработке адекватного кода бывает координация доступа к данным, совместно используемым в нескольких потоках управления. Итогами попыток обеспечения для этого взаимного исключения при слишком слабой или слишком высокой синхронизации являются конфликты при доступе к данным, тупик синхронизации (deadlock) и невысокая масштабируемость.

Полученный результат показывает, что требования (3,4) нарушились, и система остановила дальнейшую проверку на состоянии 327. Автомат бесконечных выполнений обнаружил изменения направления на замке never claim. И тем самым предоставил контрпример траектории нарушения.

Однако, надо учитывать, что проверялись только некоторые спецификации требований по свойству алгоритма. Проблема автоматизации применения требований еще не решилась сфере верификации параллельных и распределенных систем. Поэтому, утверждаем, что верификация проводилась не в полной мере. С другой стороны, в количестве 9 процессов, система вышла с аварийным предупреждением: out of memory, что является результатом комбинаторного взрыва.

3.2 Проверка алгоритма Parsing через ByMC.

ByMC включает в себе алгоритм Византийского соглашения и совокупность различных методов верификации, таких как, модифицированный Spin с входным языком Promela и Yices, который, контролирует параметризацию абстракции модели и выполне-

```
sakura
    12  S{bymc_kP={0,0,0,0,2,0,0,0,1,1,2,0,0,0,0,0},nsnt=2}
    13  S{bymc_kP={0,0,0,0,2,0,0,0,1,2,2,0,0,0,0,0},nsnt=2}
    14  S{bymc_kP={0,0,0,0,2,0,0,0,1,2,2,0,0,0,0,0},nsnt=2}
    15  S{bymc_kP={0,0,0,0,1,0,0,0,1,2,2,0,0,0,0,0},nsnt=2}
    16  S{bymc_kP={0,0,0,0,0,0,0,0,1,2,2,0,0,0,0,0},nsnt=2}
    17  S{bymc_kP={0,0,0,0,0,0,0,0,0,3,2,0,0,0,0,0},nsnt=2}
    18  S{bymc_kP={0,0,0,0,0,0,0,0,0,3,2,0,0,0,0,0},nsnt=2}
    19  S{bymc_kP={0,0,0,0,0,0,0,0,0,3,3,0,0,0,0,0},nsnt=2}
    20  S{bymc_kP={0,0,0,0,0,0,0,0,0,2,3,0,0,0,0,0},nsnt=2}
    21  S{bymc_kP={0,0,0,0,0,0,0,0,0,1,3,0,0,0,0,0},nsnt=2}
    22  S{bymc_kP={0,0,0,0,0,0,0,0,0,0,3,0,0,0,0,0},nsnt=2}
    23  S{bymc_kP={0,0,0,0,0,0,0,0,0,0,3,0,0,0,0,1},nsnt=2}
    24  S{bymc_kP={0,0,0,0,0,0,0,0,0,0,3,0,0,0,0,1},nsnt=2}
        <<<<<START OF CYCLE>>>>>
    25  S{bymc_kP={0,0,0,0,0,0,0,0,0,0,3,0,0,0,0,1},nsnt=2}
    26  S{bymc_kP={0,0,0,0,0,0,0,0,0,0,3,0,0,0,0,1},nsnt=2}

  (status trace-refined)
  (status trace-refined)
Refinement step #9.
Converting the spec: !((([](<>( ! in_transit_A))) && ([]( !
( ! ex_acc_E) || (<>all_acc_A))))...
Generating pan...
+ spin -a -N relay.never abs-counter.prm
```

Рисунок 2 –Верификация параллельного алгоритма "Parsing"в системе ByMC

ния контрпримера с входной формулой SMT. Для автоматизации абстрагирования модели выполняется компонент технологии CEGAR[11]. ByMC считается единой системой с несколькими верификаторами для полной верификации распределенных алгоритмов.

Рассмотрим модельную проверку алгоритма Parsing на верификаторе ByMC. Поскольку установка и обучение системе требует отдельного труда, мы решили показать только результаты, полученные при верификации ByMC. Реализация и внедрение алгоритма Parsing на данной нетривиальной системе привела алгоритм к изменению. В алгоритм внедрялись: пороговая граница для процессоров, система отслеживания искусственного контрпримера и т.п. Поскольку, распределенные алгоритмы требует высокого контроля в плане абстракции, модель уменьшилась до пороговой границы. Генерация осуществилась в ходе верификации через Yices. Были внесены дополнительные параметры $N; T; F$, которые реализуют тестирование отказоустойчивости данного алгоритма, где N - лояльные процессоры, T - пороговая граница, F - дефектные процессы. Для этих параметров существует предкомпиляционная область определение $assume(N > 3 * T\ AND\ T >= 1\ AND\ 0 <= F\ AND\ F <= T)$.

Утверждение $\exists i\ rcvd_i < nsnt$ описывает глобальное состояние, в котором сообщения еще находятся в пути. Отсюда следует, что формула ϕ определяется как

$$GF(rcvd_i < nsnt) \tag{5}$$

где, $rcvd_i$ - принятые локальные переменные, $nsnt$ - общие переменные для каждого процессора. По результатам верификации алгоритма Parsing на ByMC (Рисунок 2), выявлены одни и те же результаты, что и при верификации в Spin. Однако, если учитывать в плане автоматизации абстрагирование и недетерминированный подбор требований на несколько тысяч вариантов (что составляет в сумме 5936 [12]), система ByMC превосходит по качеству и количеству проверяемых процессов на аппаратном и программном уровне. В Spin количество процессов было ограничено до 9, на ByMC количество процессов динамически менялось и было доведено до 16.

Заключение

Было проведено сравнительное исследование верификации параллельного алгоритма Parsing на верификаторе Spin и модернизированного варианта Spin - ByMC. ByMC показал лучшие результаты в плане верификации больших систем и по количеству проверяемых процессов на распределенном алгоритме Parsing. Работа выполнена при поддержке грантового финансирования научно-технических программ и проектов Комитетом науки МОН РК, грант № 5033/ГФ4*

Литература

[1] *Lamport L.* A new solution of Dijkstra's concurrent programming problem. // Commun. ACM 17(8), - 1974. - P. 453–455.

[2] *Srikanth T., Toueg S.* Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. // Distributed Computing 2, - 1987. - P. 80–94.

[3] *John A., Konnov I., Schmid U., Veith H., Widder J.* Counter Attack on Byzantine Generals: Parameterized Model Checking of Fault-tolerant Distributed Algorithms. // Cornell University Library. Ithaca. NY. - V2. - 2013. - P. 56-75.

[4] *Miner D., Shook A.* MapReduce Design Patterns. Building Effective Algorithms and Analytics for Hadoop and Other Systems. // Ottawa. Canada. ACM. NY. - 2012. – P. 204–213.

[5] *Ахмед-Заки Д.Ж., Бектемесов А.Т.* Симмуляция распределенных программ при использовании транзакционной памяти. // Вестник КазНУ. Алматы – 2014. – С. 26-33.

[6] *Charron-Bost B., Pedone F., Schiper A.* Replication: Theory and Practice // Lecture Notes in Computer Science. Springer. - V 5959. - 2010. - P. 156-167.

[7] *Bokor P., Kinder J., Serafini M., Suri N.* Efficient model checking of fault-tolerant distributed protocols. // In: DSN. - 2011. - P. 73–84.

[8] *Clarke E. M., Grumberg O., Peled D.A.* Model Checking // The MIT Press. London. England. - 1999. –330 p.

[9] *John A., Konnov I., Schmid U., Veith H., Widder J.* Starting a dialog between model checking and fault-tolerant distributed algorithms // arXiv CoRR abs/1210.3839. - 2012.

[10] *Aubakirov S., Trigo P. and Ahmed-Zaki D.* Comparison of Distributed Computing Approaches to Complexity of n-gram Extraction // Agent and Systems Modeling, Portugal Proceedings of DATA, - 2016. –P. 25–30.

[11] *Clarke E., Grumberg O., Jha S., Lu Y., Veith H.* Counterexample-guided abstraction refinement for symbolic model checking. // J.ACM 50(5). - 2003. - P. 752–794

[12] Benchmarks Spin2013 ByMC 0.9.5: Byzantine model checker, http://forsyte.tuwien.ac.at/software/bymc/ (Application date: 05.08.2016)

References

[1] *Lamport L.* A new solution of Dijkstra's concurrent programming problem. // Commun. ACM 17(8), - 1974. - P. 453–455.

[2] *Srikanth T., Toueg S.* Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. // Distributed Computing 2, - 1987. - P. 80–94.

[3] *John A., Konnov I., Schmid U., Veith H., Widder J.* Counter Attack on Byzantine Generals: Parameterized Model Checking of Fault-tolerant Distributed Algorithms. // Cornell University Library. Ithaca. NY. - V2. - 2013. - P. 56-75.

[4] *Miner D., Shook A.* MapReduce Design Patterns. Building Effective Algorithms and Analytics for Hadoop and Other Systems. // Ottawa. Canada. ACM. NY. - 2012. – P. 204–213.

[5] *Akhmed-Zaki D.Zh., Bektemessov A.T.* Simulation of distributed programm with transactional memory. // Buletin KazNU. Almaty – 2014. – P. 26-33.

[6] *Charron-Bost B., Pedone F., Schiper A.* Replication: Theory and Practice // Lecture Notes in Computer Science. Springer. - V 5959. - 2010. - P. 156-167.

[7] *Bokor P., Kinder J., Serafini M., Suri N.* Efficient model checking of fault-tolerant distributed protocols. // In: DSN. - 2011. - P. 73–84.

[8] *Clarke E. M., Grumberg O., Peled D.A.* Model Checking // The MIT Press. London. England. - 1999. –330 p.

[9] *John A., Konnov I., Schmid U., Veith H., Widder J.* Starting a dialog between model checking and fault-tolerant distributed algorithms // arXiv CoRR abs/1210.3839. - 2012.

[10] *Aubakirov S., Trigo P. and Ahmed-Zaki D.* Comparison of Distributed Computing Approaches to Complexity of n-gram Extraction // Agent and Systems Modeling, Portugal Proceedings of DATA, - 2016. –P. 25–30.

[11] *Clarke E., Grumberg O., Jha S., Lu Y., Veith H.* Counterexample-guided abstraction refinement for symbolic model checking. // J.ACM 50(5). - 2003. - P. 752–794

[12] Benchmarks Spin2013 ByMC 0.9.5: Byzantine model checker, http://forsyte.tuwien.ac.at/software/bymc/ (Application date: 05.08.2016)